Jowidgets Nutzerhandbuch

Michael Grossmann

02. Februar 2015

Inhaltsverzeichnis

Kapitel 1

Einführung

Dieses Nutzerhandbuch bietet eine Einführung in die Verwendung von jowidgets.

Das Dokument ist sowohl als PDF Version als auch online unter http://www.jowidgets.org/docu/ verfügbar.

Unter http://www.jowidgets.org/api_doc/ ist die jowidgets API Spezifikation zu finden.

1.1 Bachelorarbeiten

Im Rahmen der Jowidgets Entwicklung wurden mehrere Bachelorarbeiten zum Thema angefertigt. Die folgende Liste enthält eine Auswahl:

- Erweiterung von Layout-Manager Konzepten für Jo Widgets
- Entwicklung eines Prototyps einer Test-Bibliothek zum GUI Framework Jo Widgets für automatisierte Tests
- Evaluierung der Eignung von JavaFX 2 als UI Technologie für das GUI-Framework Jo Widgets anhand einer prototypischen Implementierung des Jo Widgets Service Provider Interfaces

1.2 Lizenz

Jowidgets steht unter der BSD Lizenz und kann somit sowohl für kommerzielle als auch für nicht kommerzielle Zwecke frei verwendet werden.

1.3 Motivation

Jowidgets ist eine API zur Erstellung von graphischen Benutzeroberflächen mit Java.

Der kritische Leser stellt sich nun eventuell die Frage: "*Warum noch ein UI Framework, gibt es da nicht schon genug?*"

Die Erfahrung bei der Entwicklung von graphischen Oberflächen in unterschiedlichen Unternehmen hat gezeigt, das für viele in Unternehmen typischen Anwendungsfälle oft keine Lösungen in Standard UI Frameworks existieren, und aus dieser Not heraus das *Rad* für diese immer wieder *neu erfunden* wird.

Hierzu ein kleines Beispiel. In einer Eingabemaske soll in ein Eingabefeld eine Zahl eingegeben werden. Die Eingabe soll validiert werden, und der Nutzer soll ein möglichst für ihn verständliches Feedback bekommen, wenn etwas falsch ist. Für die Eingabe bietet Swing ein `JTextField`. Dieses liefert aber einen String und keine Zahl zurück, das heißt der Wert muss erst konvertiert werden. Für die Anzeige des Validierungsfeedback könnte man in Swing ein `JLabel` verwenden. Da man Anwendungsfälle wie die Eingabe von Zahlen, Eingabe eines Datum, Anzeige von Validierungsfehlern, etc. in sehr vielen Softwarehäusern vorfindet, existieren vermutlich auch in jeder dieser firmeninternen UI Bibliothek Widgets wie: `InputNumberField`, `ValidatedInputNumberField`, `ValidatedDateField`, `ValidationLabel` usw.. Wer sich hier *wieder findet* oder wer vielleicht selbst schon mal solch ein Widget implementiert oder verwendet hat, für den könnte jowidgets möglicherweise genau *das richtige* UI Framework sein.

Im Vergleich zu herkömmlichen Technologien wie *Swing*, oder *JavaFx* liefert jowidgets kein eigenständiges Rendering für Basiswidgets [1], sondern lediglich Adapter, welche die jowidgets Widget Schnittstellen implementieren. Dadurch ist es möglich, UI Code, welcher gegen das jowidgets API implementiert wurde, mit quasi jedem Java UI Framework auszuführen. Derzeit existieren Adapter (SPI Implementierungen, siehe Architektur) für Swing, SWT und RWT. Die RWT Implementierung ermöglicht es somit sogar eine jowidgets Applikation als Webapplikation im Browser auszuführen. Eine *JavaFx* Adapter Implementierung wurde im Rahmen einer Bachelorarbeit prototypisch umgesetzt. Es existieren Bundle Manifeste für OSGi, so dass auch ein Einsatz in Eclipse RCP (siehe Jowidgets und RCP) ohne weiters möglich ist.

Aufbauend auf den Basiswidgets existieren wie bereits weiter oben angedeutet zusätzliche Composite Widgets und Features, welche sich in herkömmlichen UI-Frameworks nicht finden. Zwei davon sollen an dieser Stelle exemplarisch vorgestellt werden.

Beispielsweise gibt es ein generisches `InputField<VALUE_TYPE>`. Dieses hat Methoden wie `VALUE_TYPE getValue()` oder `setValue(VALUE_TYPE value)`. Für Standarddatentypen wie `Integer`, `Long`, `Double`, `Float`, `Date`, etc. existieren bereits Defaultimplementierungen. Im obigen Beispiel würde das Eingabefeld also bereits eine Zahl liefern. Mit Hilfe von Convertern kann aber auch jeder beliebige andere Datentyp unterstützt werden. Für ein `InputField` lassen sich auch beliebige Validatoren definieren. Zudem kann die UI an `ObservableValues` gebunden werden.

Für das nächste Feature betrachten wir noch einmal das vorige Beispiel mit dem Eingabefeld von Zahlen. Für die Validierungsausgabe könnte ein ValidationLabel Widget erstellt worden sein, welches Warnungen und Fehler mit entsprechenden Icons und / oder Fehlertext anzeigen kann. Dabei kann man sowohl die Icons als auch die Farbe konfigurieren, weil Kunde A Fehler nur mit *dezenten* Farben angzeigt bekommen möchte (laut Aussage seiner Mitarbeiter bekommt man von der standardmäßig verwendeten Farbe *Augenkrämpfe*), Kunde B kann aber Validierungsfehler, die nicht *fett* und *rot* angezeigt werden, gar nicht wahrnehmen, was die Applikation für ihn unbrauchbar macht. Stellen wir uns nun weiter vor, diese Eingabefelder und Validierungslabels sind in ein Widget mit dem Namen `GenericInputForm` eingebettet, welches sowohl von Kunde A als auch von Kunde B verwendet wird, und stellen wir uns weiter vor, dass die verwendeten Widgets noch weitere Konfigurationsattribute haben, dann müssten diese alle auch für das `GenericInputForm` konfigurierbar sein. Spätestens wenn dieses in ein weiteres Modul eingebettet ist, wird klar, dass dieses Vorgehen schnell unpraktikabel wird.

In jowidgets ist es u.A. für solche Anwendungsfälle möglich, für jedes beliebige Widget die Defaulteinstellungen global *umzudefinieren* (siehe dazu Widget Defaults). So könnte man im genannten Beispiel die gleiche Applikation einmal mit *roten* und einmal mit *grauen* Validierungslabels ausliefern, ohne dazu die eigentliche Applikation anpassen zu müssen. Mit der gleichen Methode könnte man zum Beispiel auch die Reihenfolge für Buttons in Eingabedialogen anpassen, definieren in welchem Format Datumswerte angezeigt werden sollen, definieren ob Buttons für Speichern und Abbrechen auf jedem Formular oder nur in der Toolbar vorhanden sind, uvm..

[1] Wie zum Beispiel Frame, Dialog, Composite, Button, TextField, ...

1.4 Architektur

Die folgende Abbildung zeigt die jowidgets Architektur.

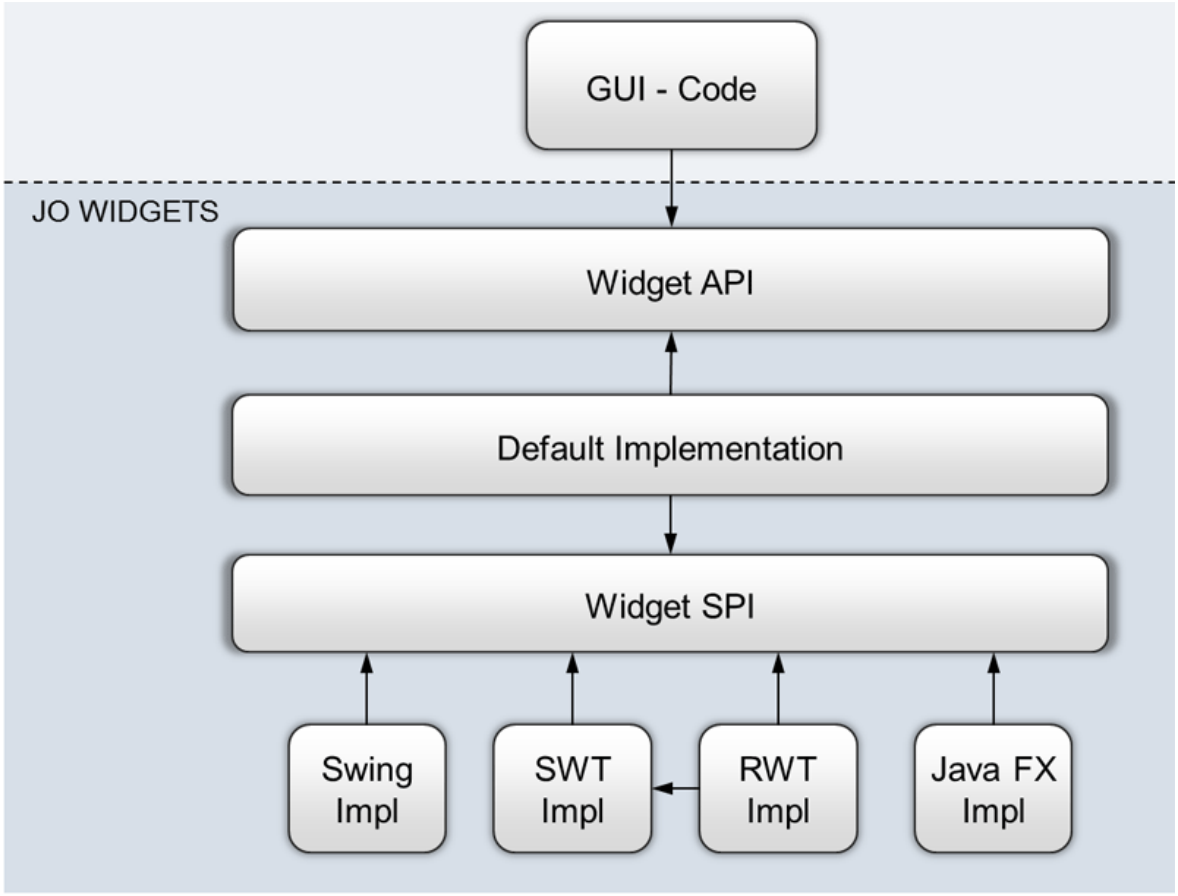


Abbildung 1.1: Architektur

UI Code wird gegen das Widget API implementiert. Die Default Implementierung von jowidgets verwendet das Widget SPI [2], um die Features der API zu implementieren. Für die SPI existieren derzeit Implementierungen für Swing, SWT und RWT. Die SPI Implementierungen enthalten Adapter, welche die SPI Widget Schnittstellen implementieren.

Die Schnittstellen der SPI Widgets wurden bewusst *schlank* gehalten, um das Hinzufügen einer neuen UI Technologie möglichst einfach zu gestalten, ohne dabei Abstriche bei der *Mächtigkeit* der API machen zu müssen.

Dies soll am Vergleich mit dem Eclipse Standard Widget Toolkit (SWT) erläutert werden. Dort findet man eine ähnliche Architekur. Es gibt eine API, welche für verschiedenen Platformen implementiert ist. Die Implementierungen stellen im Prinzip Adapter bereit, welche mittels JNI Methodenaufrufe an das jeweilige native UI Toolkit delegieren.

Wer SWT kennt, weiß jedoch, dass sich die Verwendung der API zum Teil sehr *low level anfühlt*. Will man hingegen eine API Implementierung machen (zum Bespiel für Swing oder JavaFX), sieht aus dieser Perspektive betrachtet die API sehr *mächtig* und komplex aus. Bei der Definition der

[2] Service Provider Interface

Schnittstellen musste immer ein Kompromiss aus einfacher Implementierbarkeit und komfortabler Nutzbarkeit gemacht werden.

Durch die Aufteilung in API und SPI kann die API *mächtige* Funktionen enthalten, welche aber nur ein mal implementiert werden müssen. Auf der anderen Seite ist eine SPI Implementierung für eine neue UI Technologie keine allzu komplexes Aufgabe, was die Option bietet, geschrieben UI Code auch für zukünftige UI Technologien wiederzuverwenden.

Im diesem Kontext liegt vielleicht der Vergleich nahe, die jowidgets SPI mit SWT zu vergleichen, und die jowidgets API mit JFace. Dieser Vergleich ist jedoch nicht ganz korrekt, denn:

- Die jowidgets SPI muss man nicht direkt verwenden, sondern man verwendet die *high level* Schnittstellen der API. Es gibt jedoch nicht für alle SWT Wigdets ein JFace *Pendant*.
- Die jowidgets API bietet Funktionen, welche JFace nicht bietet.[3]
- Jowidgets kann auch mit Swing oder zukünftigen Ui Technologien verwendet werden. Es gibt für SWT zwar auch eine Swing Implementierung, diese ist aber nicht vollständig und wird seit 2007 nicht mehr gepflegt. Für zukünftige UI Technologien ist eine jowidgets SPI Implementierung weniger aufwändig als eine SWT Implementierung. Code der gegen das jowidgets API implementiert wurde, ist dadurch in gewisser Hinsicht robust gegen technologische Neuerungen.

1.5 Widget Paradigma

In diesem Abschnitt soll der Begriff des *Widget* im Kontext von jowidgets definiert werden. Dazu wird zunächst ein kleiner Abstecher zur Entstehungsgeschichte gemacht.

In einem gemeinsamen Projekt zweier Unternehmen sollte ein bereits vorhandenes firmeninternes Framework neu entwickelt werden. Dieses Framework lieferte unter anderem Tabellen und Formulare für CRUD Anwendungen mit Datenbankanbindung sowie Sortierung und Filterung in der Datenbank (und nicht im Client, weil das für große Datenmengen nicht möglich ist). Bei der Neuentwicklung sollten bisherige Schwachpunkte wie eine fehlende 3 Tier Architektur, keine Security in der Service Schicht und eine nicht austauschbare Datenschicht optimiert werden. [4]

Die eine Firma setzte Eclipse RCP ein und wollte daran auch nichts ändern, die andere Swing. Die neu entwickelten Module sollten aber auch für die Erweiterung bisheriger (Swing) Applikation *kompatibel* sein und ob man zukünftig anstatt Swing SWT einsetzten wollte, war auch nicht geklärt.

So entstand die Idee, für die UI relevanten Anteile wie die `BeanTable` und das `BeanForm` Schnittstellen zu definieren. Die eine Firma implementierte dann das dazugehörige Widget für Swing, die andere für SWT. Schnell wurde klar, das diese Aufteilung zu *grob* war, denn es entstand hauptsächlich für das Formular viel redundanter Code. Es wuchs die Begehrlichkeit, auch für Eingabefelder, Comboboxen, etc. Schnittstellen zu definieren, um das `BeanForm` auf Basis dieser implementieren zu können. Das war die *Geburtsstunde* von jowidgets.

Man kam zu dem Schluss, dass jede Interaktion mit dem Nutzer (HCI) durch eine Java Schnittstelle spezifiziert werden kann und soll. Dies gilt für einfache Controls wie ein `Button`, eine `ComboBox` oder ein `Eingabefeld` genauso wie für zusammengesetzte Widgets wie Beispielsweise ein Formular oder ein Dialog zur Eingabe einer Person. Während das einfache Control vielleicht eine Zahl oder einen Text liefert, liefert ein `BeanForm` ein Bean und ein `PersonDialog` ein Personenobjekt.

[3] JFace bietet allerdings auch Funktionen, welche jowidgets *noch* nicht bietet.
[4] Dieses Framework ist auch unter BSD Lizenz veröffentlicht und findet sich hier jo-client-platform

Das API sollte nicht nur diese einfachen und zusammengesetzten Widgets Schnittstellen bereitstellen, sondern auch eine Platform bieten um selbst eigene Widget Bibliotheken nach dem gleichen Konzept und mit dem gleichen Benefit wie anpassbaren Defaultwerten, Dekorierbarkeit, Testbarkeit, etc. zu erstellen.

Jowidgets soll eine **offene**, **erweiterbare** API für Widgets in Java bieten. Der Name jowidges steht ursprünglich für *Java Open Widgets*.

> *Defintion: Ein Widget ist eine Schnittstelle für den Austausch von Informationen zwischen Nutzer und Applikation*

In jowidgets werden diese Schnittstellen immer durch Java Interfaces abgebildet. Es kann generell mehrere Implementierungen für die gleiche Schnittstelle geben. So könnte zum Beispiel die Eingabe von Personendaten auf einem Tablet anders aussehen als auf einem Desktop PC. Eine Widget Implementierung kann in der Generic Widget Factory überschrieben oder dekoriert (Decorator Pattern) werden. Zudem läßt sich das Default Setup eines jeden Widgets überschreiben.

1.6 Widget Hierarchie

Folgende Abbildungen zeigt die konzeptionelle (nicht vollständige) Widget Hierarchie von jowidgets. Widgets teilen sich auf oberster Ebene in Komponenten (Component) und Items mit Menüs auf.

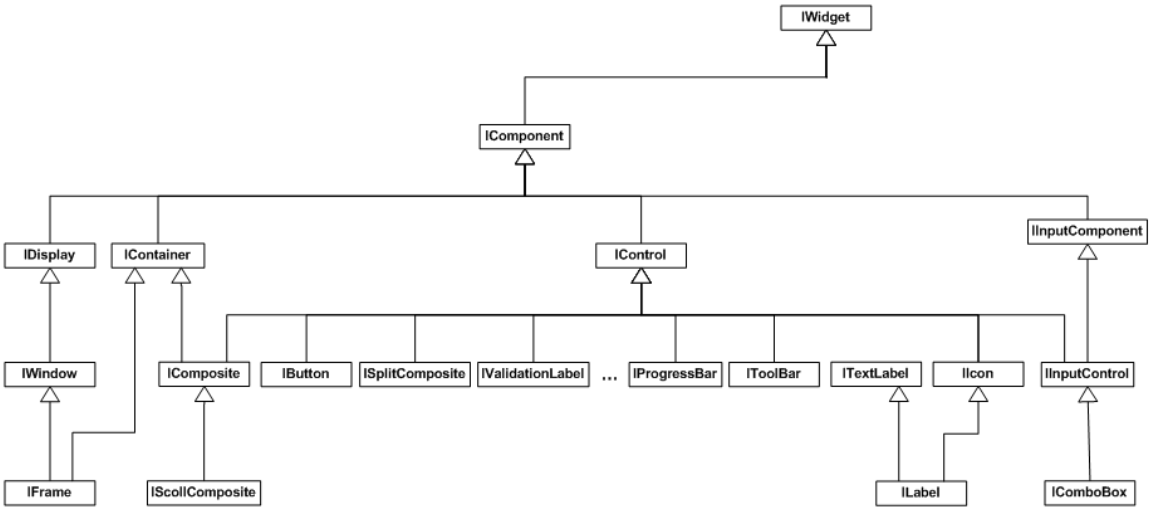


Abbildung 1.2: Widget Hierarchie / Components

Komponenten sind Fenster, Controls, Container oder InputComponents. Container enthalten Controls bzw. Controls sind Elemente von Containern. InputComponents liefern eine Nutzereingabe für einen definierten Datentyp und stellen diesen Wert dar. Dieser kann sowohl einfach sein (z.B. `String`, `Integer`,`Date`, ...) als auch komplex (z.B. `Person`, `Company`, `Rule`, ...).

Items sind zum Einen die Elemente von Menüs oder Toolbars. Weitere Items sind TabItems, also die *Reiter* eines TabFolder sowie die Nodes eines Tree[5]. Eine MenuBar enthält Menüs.

[5] Dieser Aspekt fehlt in der Abbildung

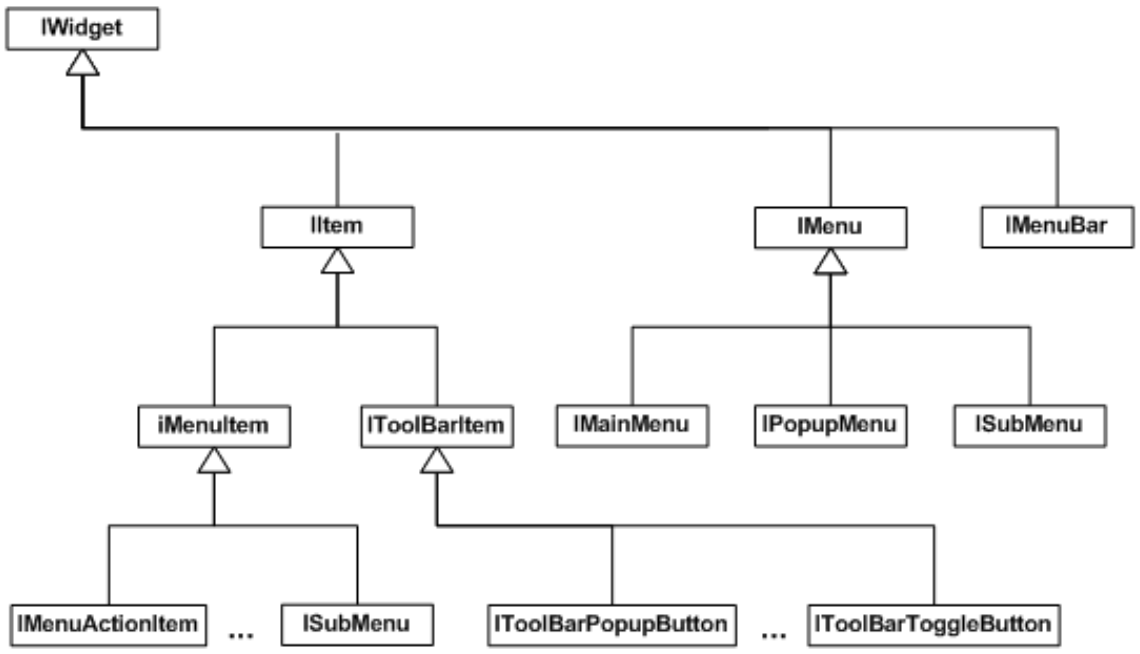


Abbildung 1.3: Widget Hierarchie / Items

Kapitel 2

Getting started

2.1 Maven

Für die Verwendung von jowidgets mit Maven muss folgendes Repository hinzugefügt werden:

```
<repositories>
    <!-- The jowidgets maven repository -->
    <repository>
        <id>jowidgets</id>
        <url>http://jowidgets.org/maven2/</url>
    </repository>
</repositories>
```

2.2 Das Hello World Beispiel

Das Hello World Beispiel soll als Einstieg in jowidgets dienen. Dabei wird bewußt etwas mehr in die Tiefe gegangen, als nötig wäre, um ein Fenster mit einem Button in jowidgets anzuzeigen. Das Beispiel kann gut als Grundlage für die weitere Arbeit dienen. Daher wird empfohlen, es auszuchecken oder nachzuimplementieren.

Um das Hello World Beispiel zu compilieren und zu starten sollten folgende Tools vorhanden sein:

- Java (mindestens 1.6)
- Maven
- GIT
- Eclipse (inklusive M2e Maven Integration)
- Tomcat (für das Starten im Browser)

Jowidgets kann hier https://github.com/jo-source/jo-widgets.git per GIT ausgecheckt werden.

Das Hello World Beispiel befindet sich hier https://github.com/jo-source/jo-widgets/tree/master/modules/helloworld und hat die folgende Verzeichnisstruktur:

Die Module können mit **Import Existing Maven Projects** in eclipse importiert werden.

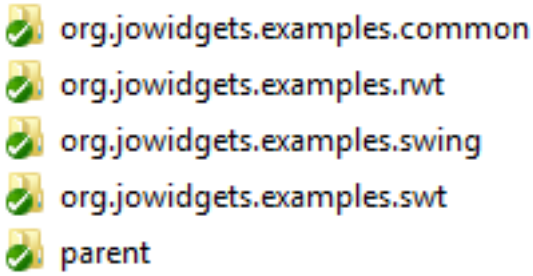


Abbildung 2.1: Hello World Module

2.2.1 Das parent Modul

Im *parent* Verzeichnis findet sich das parent pom.xml.

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.jowidgets.helloworld</groupId>
    <artifactId>org.jowidgets.helloworld.parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <properties>
        <!-- jowidgets needs java 1.6 or higher -->
        <java.version>1.6</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <jowidgets.version>0.40.0</jowidgets.version>
    </properties>

    <repositories>
        <!-- The jowidgets maven repository -->
        <repository>
            <id>jowidgets</id>
            <url>http://jowidgets.org/maven2/</url>
        </repository>
    </repositories>

    <modules>
        <!-- Hold the ui technology independend hello world code -->
        <module>../org.jowidgets.helloworld.common</module>

        <!-- Holds a starter that uses Java Swing -->
        <module>../org.jowidgets.helloworld.starter.swing</module>

        <!-- Holds a starter that uses Eclipse SWT (win32) -->
        <module>../org.jowidgets.helloworld.starter.swt</module>

        <!-- This module creates a war that uses Eclipse RWT -->
        <module>../org.jowidgets.helloworld.starter.rwt</module>
    </modules>

    ...
```

```
44
45 </project>
```

Die Hello World Applikation besteht aus vier Untermodulen.

- org.jowidgets.helloworld.common
- org.jowidgets.helloworld.starter.swing
- org.jowidgets.helloworld.starter.swt
- org.jowidgets.helloworld.starter.rwt

Das Modul `org.jowidgets.helloworld.common` enthält den SPI unabhängigen Code, die drei anderen Module beinhalten die Starter für die jeweilige SPI Implementierung sowie die zugehörigen Maven Abhängigkeiten.

In einem *realen* Projekt hat man normalerweise nicht Starter für alle möglichen SPI Implementierungen. Dennoch ist es eine gute Idee, den SPI unabhängigen Code in ein separates Modul zu packen, um ihn in anderen Projekten, welche eventuell eine andere UI Technologie voraussetzen, besser wiederverwenden zu können.

2.2.2 Das common Modul

Betrachten wir zunächst das *common* pom.xml File im Ordner `org.jowidgets.helloworld.common`. Will man die Kernfunktion von jowidgets nutzen, muss man das Modul `org.jowidgets.tools` hinzufügen.

```
1  <project
2      xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      http://maven.apache.org/xsd/maven-4.0.0.xsd">
6
7      <modelVersion>4.0.0</modelVersion>
8      <artifactId>org.jowidgets.helloworld.common</artifactId>
9
10     <parent>
11         <groupId>org.jowidgets.helloworld</groupId>
12         <artifactId>org.jowidgets.helloworld.parent</artifactId>
13         <version>0.0.1-SNAPSHOT</version>
14         <relativePath>../parent/pom.xml</relativePath>
15     </parent>
16
17     <dependencies>
18         <!-- The jowidgets api and tools -->
19         <dependency>
20             <groupId>org.jowidgets</groupId>
21             <artifactId>org.jowidgets.tools</artifactId>
22             <version>${jowidgets.version}</version>
23         </dependency>
24     </dependencies>
25
26 </project>
```

2.2.3 Transitive Abhängigkeiten des common Moduls

In der Praxis ist immer gut zu wissen, welche transitiven Abhängigkeiten man sich *einhandelt*, wenn man eine neue Technologie einführt. Jowidgets wurde bewusst so entworfen, dass (außer jowidgets

selbst) **möglichst keine weiteren externen Abhängigkeiten** notwendig sind.

Durch das Modul **`org.jowidgets.tools`** hat man die folgenden internen (siehe auch Jowidgets Modulübersicht im Anhang) und **keine externen transitiven Abhängigkeiten**.

org.jowidgets.util UI unabhängige Utilities und Datenstrukturen. Siehe auch Jowidgets Utils.

org.jowidgets.i18n Eine API für (Multi User Locale) Internationalisierung. Siehe auch i18n.

org.jowidgets.classloading.api API für Classloading Aspekte. Dies ist hauptsächlich für die OSGi Kompatibilität notwendig. Siehe auch Jowidgets Classloading.

org.jowidgets.validation Eine (UI unabhängige) API für Validierung. Siehe auch Die Validation API

org.jowidgets.validation.tools Vorgefertigte Validatoren.

org.jowidgets.unit Eine (UI unabhängige) API für den Umgang mit Einheiten (z.B. Hz, Byte, KG, etc.).

org.jowidgets.common Gemeinsame Schnittstellen und Klassen der jowidgets API und der jowidgets SPI.

org.jowidgets.api Die jowidgets API (überwiegend Java Interfaces).

org.jowidgets.tools Während die API überwiegend aus Schnittstellen besteht, finden sich hier nützliche Klassen, welche sich aus den Schnittstellen ergeben, wie zum Beispiel Default Implementierungen, abstrakte Basisklassen, Wrapper, Listener Adapter und weitere.

2.2.4 HelloWorldApplication - Der common Ui Code

Die Klasse `HelloWorldApplication` implementiert die Schnittstelle `IApplication`. Diese hat eine Startmethode, welche den `Lifecycle` als Parameter liefert. Wird auf diesem die Methode `finish()`aufgerufen, wird die Applikation beendet.

Will man jowidgets in nativen Ui Code (z.B. Swing oder SWT) integrieren, hat man in der Regel keine IApplication zum starten. Stattdessen werden das Root Fenster (z.B. Shell oder JFrame) und weitere Widgets (z.B. Composite, JPanel) nativ erzeugt. In Jowidgets Code in native Projekte integrieren wird erläutert, wie man in diesem Fall vorgehen kann.

```
1  package org.jowidgets.helloworld.common;
2
3  import org.jowidgets.api.toolkit.Toolkit;
4  import org.jowidgets.api.widgets.IButton;
5  import org.jowidgets.api.widgets.IFrame;
6  import org.jowidgets.api.widgets.blueprint.IButtonBluePrint;
7  import org.jowidgets.api.widgets.blueprint.IFrameBluePrint;
8  import org.jowidgets.common.application.IApplication;
9  import org.jowidgets.common.application.IApplicationLifecycle;
10 import org.jowidgets.common.types.Dimension;
11 import org.jowidgets.common.widgets.controller.IActionListener;
12 import org.jowidgets.common.widgets.layout.MigLayoutDescriptor;
13 import org.jowidgets.tools.widgets.blueprint.BPF;
14
15 public final class HelloWorldApplication implements IApplication {
16
17     @Override
18     public void start(final IApplicationLifecycle lifecycle) {
```

```
19
20          //Create a frame BluePrint with help of the BluePrintFactory (BPF)
21          final IFrameBluePrint frameBp = BPF.frame();
22          frameBp.setSize(new Dimension(400, 300)).setTitle("Hello World");
23
24          //Create a frame with help of the Toolkit and BluePrint.
25          //This convenience method finishes the ApplicationLifecycle when
26          //the root frame will be closed.
27          final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);
28
29          //Use a simple MigLayout with one column and one row
30          frame.setLayout(new MigLayoutDescriptor("[]", "[]"));
31
32          //Create a button BluePrint with help of the BluePrintFactory (BPF)
33          final IButtonBluePrint buttonBp = BPF.button().setText("Hello World");
34
35          //Add the button defined by the BluePrint to the frame
36          final IButton button = frame.add(buttonBp);
37
38          //Add an ActionListener to the button
39          button.addActionListener(new IActionListener() {
40              @Override
41              public void actionPerformed() {
42                  System.out.println("Hello World");
43              }
44          });
45
46          //set the root frame visible
47          frame.setVisible(true);
48      }
49  }
```

Im Vergleich zu anderern UI Frameworks gibt es in jowidgets einen wesentlichen Unterschied bei der Widget Erzeugung. Widgets werden nicht direkt, also mit Hilfe des `new` Schlüßelwortes instantiiert, sondern von einer Factory erzeugt. Dies hat einige Vorteile, siehe dazu auch Die Generic Widget Factory.

Die Wigdet Factory benötigt für die Erzeugung eines Widgets ein sogenanntes *BluePrint* (Blaupause). Siehe dazu auch BluePrints. BluePrints erhält man von der BluePrint Factory. Die Klasse `BPF` liefert einen Zugriffspunkt auf alle BluePrints der BluePrint Factory. In Zeile 21 wird so das BluePrint für ein Frame und in Zeile 33 das BluePrint für einen Button erzeugt. BluePrints sind Setup und Builder zugleich. Sie werden dazu verwendet, das Widgets zu Konfigurieren und liefern dadurch der Implementierung das initiale Setup des Widgets. In Zeile 22 wird so die Größe und der Titel des Fensters gesetzt. In Zeile 33 wird so das Button Label definiert. BluePrints sind nach dem Prinzip des *BuilderPattern* entworfen, das heißt sie liefern die eigene Instanz als Rückgabewert, um verkettete Aufrufe zu ermöglichen.

In der Regel wird die Generic Widget Factory nicht explizit verwendet. Um das root Fenster zu erhalten, wird in Zeile 27 ein IFrame Widget mit Hilfe des `frameBp` und des Toolkit's erzeugt. Bei der verwendeten Methode handelt es sich um eine *convenience Methode* die beim Schließen des Fensters den `ApplicationLifecycle` beendet. Das Frame ist gleichzeitig auch ein Container. Um ein Widget zu einem Container hinzuzufügen, fügt man das BluePrint hinzu und erhält dafür das Widget. In Zeile 36 wird so der Button erzeugt.

In Zeile 30 wird das Layout mit einer Spalte und einer Zeile für das Frame gesetzt. Zeile 39 fügt dem erzeugten Button einen Listener hinzu, welcher beim *Klicken* eine Konsolenausgabe macht. In Zeile 47 wird das Fenster schlußendlich angezeigt.

Fassen wir noch einmal zusammen: Um Widgets zu erhalten, benötigt man vorab ein BluePrint.

Dieses erhält man von der BluePrint Factory. Auf dem BluePrint kann man das Setup konfigurieren. Für ein BluePrint bekommt man dann ein Widget.

Auch wenn sich das vielleicht erst mal ungewohnt anhört, ist das auch schon alles, was im Vergleich zu anderen UI Frameworks generell anders ist.

Man wird zudem schnell feststellen, dass man dadurch sehr kompakten und zudem gut lesbaren UI Code erzeugen kann. Den im Vergleich zu langen Konstruktoraufrufen ist beim Builder Pattern immer klar, welchen Parameter man konfiguriert. Außerdem kann die Erzeugung von BluePrints meist auch implizit passieren. So könnte man den Code von Zeile 32 bis 36 auch so aufschreiben:

```
    final IButton button = frame.add(BPF.button().setText("Hello World"));
```

Zudem können BluePrints für die Erzeugung mehrerer Instanzen wiederverwendet werden, was sich oft als sehr hilfreich erweißt.

Anbei folgen noch ein paar Anmerkungen zum Code Style:

Unter dem Begriff CleanCode findet sich der Hinweis, immer *sprechende* Variablennamen zu verwenden, und insbesondere kryptische Variablennamen wie: “is, os, hfu, jkl, . . . ” etc. zu vermeiden. Im Allgemeinen ist dem auch voll zuzustimmen.

Da jedoch BluePrints in jowidgets eine so fundamentale Rolle spielen, wird hier dennoch bewusst von dieser Regel abgewichen. Der Name der *Abbreviation* Accessor Klasse `BPF` wurde bewußt kurz gewählt, um eine bessere *inline* Verwendung (siehe oben) zu ermöglichen. (Wer das trotzdem nicht mag, kann anstatt `BPF.button()` auch `Toolkit.getBluePrintFactory().button()` schreiben.)

Um BluePrint Variablen besser von den eigentlichen Widget Variablen unterscheiden zu können, sollten diese per Konvention immer die Endung `bp` oder `bluePrint` haben. Wird das BluePrint nur für die Erzeugung eines einzelnen Widgets verwendet, bietet es sich an, für BluePrint und Widget den gleichen Präfix zu wählen, also zum Beispiel `buttonBp` und `button`.

2.2.5 Der Swing Starter

Um die Applikation mit Swing zu starten, betrachten wir das Modul `org.jowidgets.helloworld.starter.swing` und dort zunächst das folgende pom.xml.

```
1  <project
2      xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7
8      <artifactId>org.jowidgets.helloworld.starter.swing</artifactId>
9
10     <parent>
11         <groupId>org.jowidgets.helloworld</groupId>
12         <artifactId>org.jowidgets.helloworld.parent</artifactId>
13         <version>0.0.1-SNAPSHOT</version>
14         <relativePath>../parent/pom.xml</relativePath>
15     </parent>
16
17     <dependencies>
18
19         <!-- The ui technology independend hello world module -->
20         <dependency>
```

```
21              <groupId>org.jowidgets.helloworld</groupId>
22              <artifactId>org.jowidgets.helloworld.common</artifactId>
23              <version>0.0.1-SNAPSHOT</version>
24          </dependency>
25
26          <!-- The default implementation of the jowidgets api -->
27          <dependency>
28              <groupId>org.jowidgets</groupId>
29              <artifactId>org.jowidgets.impl</artifactId>
30              <version>${jowidgets.version}</version>
31          </dependency>
32
33          <!-- The Swing implementation of the jowidgets spi -->
34          <dependency>
35              <groupId>org.jowidgets</groupId>
36              <artifactId>org.jowidgets.spi.impl.swing</artifactId>
37              <version>${jowidgets.version}</version>
38          </dependency>
39
40      </dependencies>
41
42  </project>
```

Ab Zeile 20 wird das common Modul der HelloWorld Applikation hinzugefügt. Ab Zeile 27 folgt die Defaultimplementierung der jowidgtes API und ab Zeile 34 wird die SPI Implementierung für Swing hinzugefügt.

Dadurch ergeben sich die folgenden (jowidgets internen) transienten Modulabhängigkeiten:

org.jowidgets.spi.impl.common Gemeinsamer Code der von allen SPI Implementierungen genutzt wird.

org.jowidgets.spi.impl.swing.common Die Swing SPI Implementierung.

org.jowidgets.spi.impl.swing Die Swing SPI Implementierung als ServiceLoader Plugin

Die Swing SPI Implementierung hat zudem eine **externe Abhängigkeit auf MigLayout für Swing**.

Die Klasse `HelloWorldStarterSwing` ist für das Starten der Applikation zuständig.

```
1   package org.jowidgets.helloworld.starter.swing;
2
3   import javax.swing.UIManager;
4
5   import org.jowidgets.api.toolkit.Toolkit;
6   import org.jowidgets.helloworld.common.HelloWorldApplication;
7
8   public final class HelloWorldStarterSwing {
9
10      private HelloWorldStarterSwing() {}
11
12      public static void main(final String[] args) throws Exception {
13          UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
14          System.setProperty("apple.laf.useScreenMenuBar", "true");
15          Toolkit.getApplicationRunner().run(new HelloWorldApplication());
16          System.exit(0);
17      }
18  }
```

In Zeile 13 wird das System System Look and Feel für Swing gesetzt, Zeile 14 enthält eine iOS spezifische Property um Menüs *Apple typisch* anzuzeigen. Um die `HelloWorldApplication`, welche `IApplication`

implementiert, zu starten, wird ein IApplicationRunner benötigt. Dieser kann vom Jowidgets Toolkit geholt werden. In Zeile 15 wird so die Applikation gestartet. Der Aufruf blockiert, bis die IApplication Implementierung auf dem übergebenen `IApplicationLifecycle` Object die Methode `finish()` aufruft. Anschließend wird die VM beendet.

Die folgende Abbildung zeigt das Hello World Fenster für die Swing SPI Implementierung:

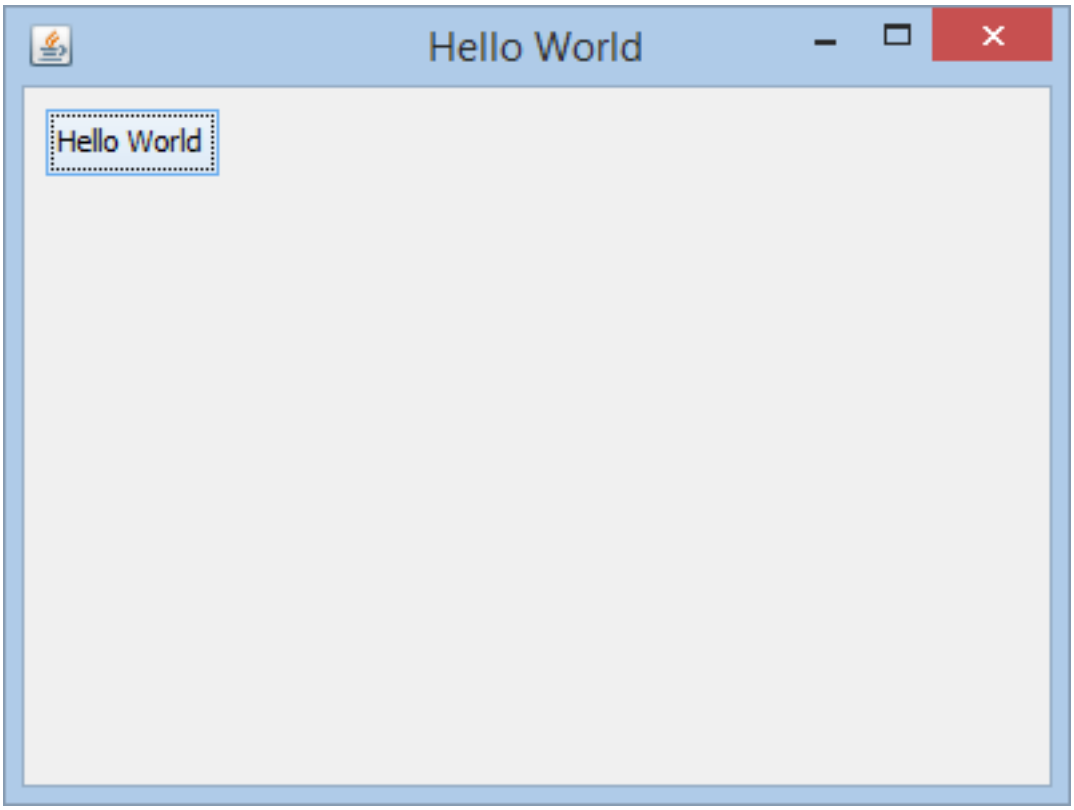


Abbildung 2.2: Hello World Swing

2.2.6 Der Swt Starter

Um die Applikation mit Swt zu starten, betrachten wir das Modul `org.jowidgets.helloworld.starter.swt` und dort zunächst das folgende pom.xml.

```
1  <project
2      xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7
8      <artifactId>org.jowidgets.helloworld.starter.swt</artifactId>
9
10     <parent>
11         <groupId>org.jowidgets.helloworld</groupId>
12         <artifactId>org.jowidgets.helloworld.parent</artifactId>
13         <version>0.0.1-SNAPSHOT</version>
```

```
14          <relativePath>../parent/pom.xml</relativePath>
15      </parent>
16
17      <dependencies>
18
19          <!-- The ui technology independend hello world module -->
20          <dependency>
21              <groupId>org.jowidgets.helloworld</groupId>
22              <artifactId>org.jowidgets.helloworld.common</artifactId>
23              <version>0.0.1-SNAPSHOT</version>
24          </dependency>
25
26          <!-- The default implementation of the jowidgets api -->
27          <dependency>
28              <groupId>org.jowidgets</groupId>
29              <artifactId>org.jowidgets.impl</artifactId>
30              <version>${jowidgets.version}</version>
31          </dependency>
32
33          <!-- The SWT implementation of the jowidgets spi -->
34          <dependency>
35              <groupId>org.jowidgets</groupId>
36              <artifactId>org.jowidgets.spi.impl.swt</artifactId>
37              <version>${jowidgets.version}</version>
38          </dependency>
39
40          <!-- The SWT implementation for win32 -->
41          <dependency>
42              <groupId>org.eclipse</groupId>
43              <artifactId>swt-win32-win32-x86</artifactId>
44              <version>4.3</version>
45          </dependency>
46
47      </dependencies>
48
49  </project>
```

Ab Zeile 20 wird das common Modul der HelloWorld Applikation hinzugefügt. Ab Zeile 27 folgt die Defaultimplementierung der jowidgtes API und ab Zeile 34 wird die SPI Implementierung für Swt hinzugefügt.

Dadurch ergeben sich die folgenden (jowidgets internen) transienten Modulabhängigkeiten:

org.jowidgets.spi.impl.common Gemeinsamer Code der von allen SPI Implementierungen genutzt wird.

org.jowidgets.spi.impl.swt.common Die Swt SPI Implementierung.

org.jowidgets.spi.impl.swt Die Swt SPI Implementierung als ServiceLoader Plugin

Selbstverständlich wird auch eine **externe Abhängigkeit auf Swt** benötigt (ab Zeile 41).

Die Swt SPI Implementierung hat zudem eine **externe Abhängigkeit auf MigLayout für Swt**.

Hinweis: Bei der Verwendung von 64Bit Java unter Windows ist die SWT Abhängigkeit wie folgt auszutauschen

```
1       <!-- The SWT implementation for win64 -->
2       <dependency>
3           <groupId>org.eclipse</groupId>
```

```
        <artifactId>swt-win32-win32-x86_64</artifactId>
        <version>4.3</version>
    </dependency>
```

Weitere SWT Maven Artefakte für andere Betriebsysteme finden sich unter anderem hier: http://www.jowidgets.org/maven2/org/eclipse/

Die Klasse `HelloWorldStarterSwt` ist für das Starten der Applikation zuständig.

```
package org.jowidgets.helloworld.starter.swt;

import org.jowidgets.api.toolkit.Toolkit;
import org.jowidgets.helloworld.common.HelloWorldApplication;

public final class HelloWorldStarterSwt {

    private HelloWorldStarterSwt() {}

    public static void main(final String[] args) throws Exception {
        Toolkit.getApplicationRunner().run(new HelloWorldApplication());
        System.exit(0);
    }
}
```

Das Starten verhält sich analog zum Hello World Swing Starter

Die folgende Abbildung zeigt das Hello World Fenster für die Swt SPI Implementierung:

2.2.7 Der Rwt Starter

Um die Applikation mit Rwt als Webapplikation zu starten, betrachten wir das Modul `org.jowidgets.helloworld.starter.rwt` und dort zunächst das folgende pom.xml.

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>org.jowidgets.helloworld.starter.rwt</artifactId>
    <packaging>war</packaging>

    <parent>
        <groupId>org.jowidgets.helloworld</groupId>
        <artifactId>org.jowidgets.helloworld.parent</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../parent/pom.xml</relativePath>
    </parent>

    <dependencies>

        <!-- The ui technology independend hello world module -->
        <dependency>
            <groupId>org.jowidgets.helloworld</groupId>
            <artifactId>org.jowidgets.helloworld.common</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>

```

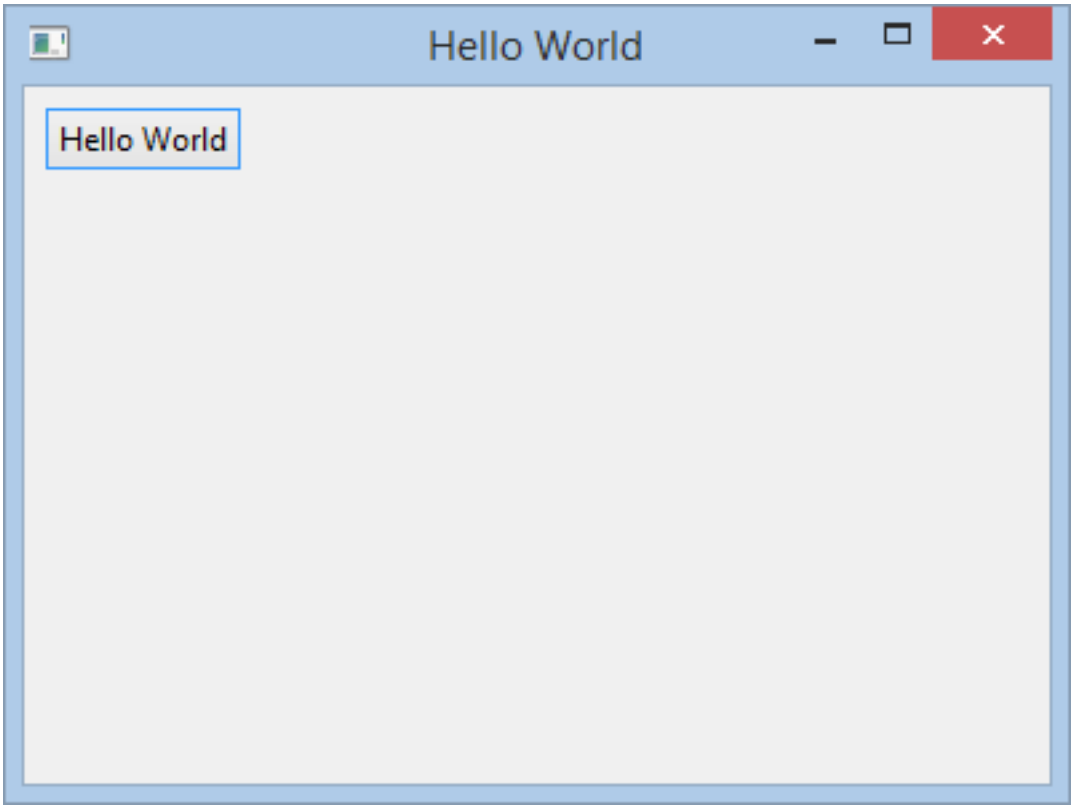


Abbildung 2.3: Hello World Swt

```
27          <!-- The default implementation of the jowidgets api -->
28          <dependency>
29              <groupId>org.jowidgets</groupId>
30              <artifactId>org.jowidgets.impl</artifactId>
31              <version>${jowidgets.version}</version>
32          </dependency>
33
34          <!-- The RWT implementation of the jowidgets spi -->
35          <dependency>
36              <groupId>org.jowidgets</groupId>
37              <artifactId>org.jowidgets.spi.impl.rwt</artifactId>
38              <version>${jowidgets.version}</version>
39          </dependency>
40
41          <!-- The RWT dependency -->
42          <dependency>
43              <groupId>org.eclipse</groupId>
44              <artifactId>org.eclipse.rap.rwt</artifactId>
45              <version>2.0.0.20130205-1612</version>
46          </dependency>
47
48      </dependencies>
49
50      <build>
51          <finalName>helloWorldRwt</finalName>
52      </build>
53
54  </project>
```

Ab Zeile 20 wird das common Modul der HelloWorld Applikation hinzugefügt. Ab Zeile 27 folgt die Defaultimplementierung der jowidgtes API und ab Zeile 34 wird die SPI Implementierung für Rwt hinzugefügt.

Dadurch ergeben sich die folgenden (jowidgets internen) transienten Modulabhängigkeiten:

org.jowidgets.spi.impl.common Gemeinsamer Code der von allen SPI Implementierungen genutzt wird.

org.jowidgets.spi.impl.swt.common Die Swt SPI Implementierung.

org.jowidgets.spi.impl.rwt Die Rwt SPI Implementierung

Selbstverständlich wird auch eine **externe Abhängigkeit auf Rwt** benötigt (ab Zeile 41).

Die Swt SPI Implementierung hat zudem eine **externe Abhängigkeit auf MigLayout für Swt**.

Betrachten wir als nächstes die Klassen `HelloWorldConfiguration` und `HelloWorldStarterRwt`

```
1   package org.jowidgets.helloworld.starter.rwt;
2
3   import java.util.HashMap;
4   import java.util.Map;
5
6   import org.eclipse.rap.rwt.application.Application;
7   import org.eclipse.rap.rwt.application.Application.OperationMode;
8   import org.eclipse.rap.rwt.application.ApplicationConfiguration;
9   import org.eclipse.rap.rwt.client.WebClient;
10
11  public final class HelloWorldConfiguration implements ApplicationConfiguration {
12
13      @Override
```

```
14      public void configure(final Application application) {
15          application.setOperationMode(OperationMode.SWT_COMPATIBILITY);
16          final Map<String, String> properties = new HashMap<String, String>();
17          properties.put(WebClient.PAGE_TITLE, "Hello World");
18          application.addEntryPoint("/HelloWorld", HelloWorldStarterRwt.class, properties);
19      }
20  }
```

```
1   package org.jowidgets.helloworld.starter.rwt;
2
3   import org.jowidgets.api.toolkit.Toolkit;
4   import org.jowidgets.helloworld.common.HelloWorldApplication;
5   import org.jowidgets.spi.impl.rwt.RwtEntryPoint;
6
7   public final class HelloWorldStarterRwt extends RwtEntryPoint {
8
9       //URL: http://127.0.0.1:8080/helloWorldRwt/HelloWorld
10      public HelloWorldStarterRwt() {
11          super(new Runnable() {
12              @Override
13              public void run() {
14                  Toolkit.getApplicationRunner().run(new HelloWorldApplication());
15              }
16          });
17      }
18
19  }
```

In Zeile 14 wird analog zu Swing und Swt die Applikation mit Hilfe des ApplicationRunners gestartet.

Für die Webapplikation ist zudem noch die folgende web.xml Datei erforderlich.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <web-app
3       xmlns="http://java.sun.com/xml/ns/j2ee"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
6       http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
7       version="2.4">
8
9       <display-name>Hello World Rwt</display-name>
10
11      <context-param>
12          <param-name>org.eclipse.rap.applicationConfiguration</param-name>
13          <param-value>org.jowidgets.helloworld.starter.rwt.HelloWorldConfiguration</param-value>
14      </context-param>
15
16      <listener>
17          <listener-class>org.eclipse.rap.rwt.engine.RWTServletContextListener</listener-class>
18      </listener>
19
20      <servlet>
21          <servlet-name>HelloWorld</servlet-name>
22          <servlet-class>org.eclipse.rap.rwt.engine.RWTServlet</servlet-class>
23      </servlet>
24
25      <servlet-mapping>
26          <servlet-name>HelloWorld</servlet-name>
27          <url-pattern>/HelloWorld</url-pattern>
28      </servlet-mapping>
29  </web-app>
```

Um die Applikation zu starten, muss zunächst das `helloWorldRwt.war` gebaut werden. Dazu kann man in der Konsole im Verzeichnis `helloworld/parent` den Befehl

```
mvn clean install
```

eingeben. Im Ordner `helloworld/org.jowidgets.helloworld.starter.rwt/target` findet sich dann die Datei `helloWorldRwt.war`. Diese kann zum Beispiel in einem Tomcat deployed werden. Macht man das lokal, kann man die Applikation dann mit der Url: `http://127.0.0.1:8080/helloWorldRwt/HelloWorld` im Browser starten.

Die folgende Abbildung zeigt das Hello World Fenster im Firefox Browser.

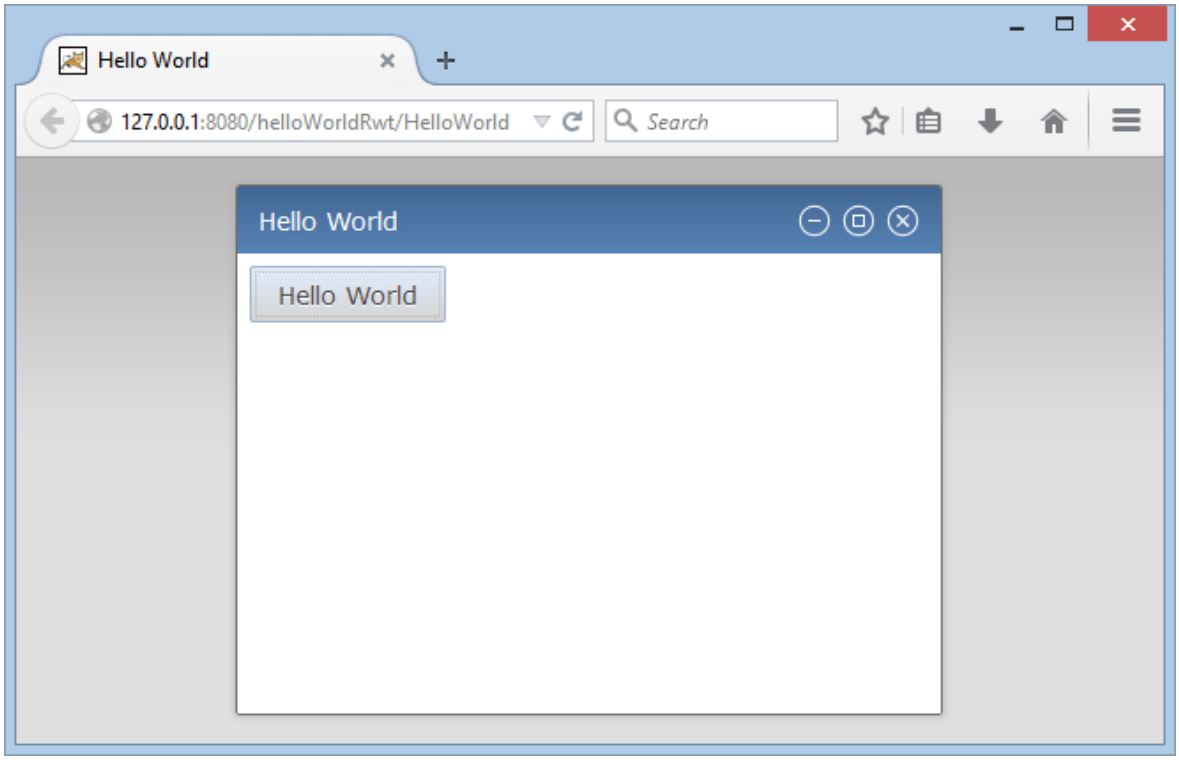


Abbildung 2.4: Hello World Rwt

Für ein tieferes Verständnis von Rwt sei auf die RAP / RWT Dokumentation verwiesen.

Weitere Information zum Thema jowidgets und RAP / RWT finden sich im Abschnitt Jowidgets und RAP

Kapitel 3

Jowidgets Basisfunktionen

3.1 Das Jowidgets Toolkit

Die Klasse `org.jowidgets.api.toolkit.Toolkit` liefert eine Instanz der Schnittstelle `org.jowidgets.api.toolkit.IToolkit`.

```
IToolkit toolkit = Toolkit.getInstance();
```

Sie stellt den **Zugriffspunkt auf die Jowidgets API** dar. Die Klasse Toolkit bietet zusätzlich zu der Methode `getInstance()` statische *Abreviation Accessor Methoden* für alle Methoden der Schnittstelle `org.jowidgets.api.toolkit.IToolkit`. Dadurch kann man zum Beispiel anstatt:

```
IUiThreadAccess uiThreadAccess = Toolkit.getInstance().getUiThreadAccess();
```

einfach

```
IUiThreadAccess uiThreadAccess = Toolkit.getUiThreadAccess();
```

schreiben.

Es sei an dieser Stelle darauf hingewiesen, das das Toolkit den Kapselungsmechanismus auf die API darstellt. Eine Aufruf einer Funktion über das Toolkit ist u.U. nicht immer intuitiv. Daher existieren zum Teil auch weitere *Abreviation Accessor Klassen*, um Zugriffe auf das Toolkit abzukürzen. So kann man zum Beispiel anstatt:

```
ITreeExpansionAction action = Toolkit.getDefaultActionFactory().expandTreeAction(tree);
```

auch

```
ITreeExpansionAction action = ExpandTreeAction.create(tree);
```

schreiben.

Per Konvention hat die *Abreviation Accessor Klasse* dann meist den Namen der Schnittstelle ohne vorgestelltes `I`. Handelt es sich um eine Factory Methode heißt diese `create()`. Existiert ein Builder, erhält man diesen auf der selben Accessor Klasse mit Hilfe der Methode `builder()`. Bezogen auf die `TreeExpandionAction` würde dass dann so aussehen:

```
ITreeExpansionActionBuilder builder = ExpandTreeAction.builder(tree);
```

Hinweise auf weitere *Abreviation Accesor Klassen* werden jeweils in den entsprechenden Abschnitten gegeben.

Im folgenden Text ist, soweit nicht gesondert anders vermerkt, mit **Toolkit** immer das jowidgets Toolkit **`org.jowidgets.api.toolkit.Toolkit`** gemeint.

3.1.1 Die Toolkit Initialisierung

Das Toolkit wird beim ersten Zugriff über den ServiceLoader Mechanismus vom Modul `org.jowidgets.impl` erzeugt (falls nicht vorab manuell ein anderer `IToolkitProvider` initialisiert wurde).

Für die SWT, Swing und JavaFx Implementierung existiert zur Laufzeit genau eine Toolkit Instanz, für die RWT Implementierung existiert genau eine Toolkit Instanz pro User Session. Im Fat / Rich Client Kontext kann das Toolkit also als Singleton betrachtet werden, im Web Kontext als Session Singleton.

Mit Hilfe des Toolkit Interceptors kann man sich benachrichtigen lassen, wenn das / ein Toolkit erzeugt wird, zum Beispiel um Widget Defaults zu überschreiben, eigene Widgets (Factories) zu registrieren, Icons zu überschreiben, Converter anzupassen oder zu registrieren usw..

3.1.2 Übersicht der Toolkit Methoden

Es folgt eine kurze Übersicht über alle Methoden der Schnittstelle `org.jowidgets.api.toolkit.IToolkit` mit kurzen Erläuterungen oder Verweisen auf die zugehörigen Abschnitte.

3.1.2.1 Application Runner

```
IApplicationRunner getApplicationRunner();
```

Der `IApplicationRunner` wird zum Starten einer jowidgets standalone Applikation benötigt. Sie dazu auch Der Application Runner.

3.1.2.2 UI Thread Access

```
IUiThreadAccess getUiThreadAccess();
```

Liefert den Zugriff auf den UI Thread. Der Zugriff muss im UI Thread erfolgen. Sie auch Der Ui Thread Access

3.1.2.3 BluePrintFactory

```
IBluePrintFactory getBluePrintFactory();
```

Liefert den Zugriff auf die BluePrintFactory

3.1.2.4 BluePrintProxyFactory

```
IBluePrintProxyFactory getBluePrintProxyFactory();
```

Liefert den Zugriff auf die BluePrintProxyFactory

3.1.2.5 ConverterProvider

```
IConverterProvider getConverterProvider();
```

Liefert den jowidgets Converter Provider. Siehe auch Jowidgets Converter

3.1.2.6 SliderConverterFactory

```
ISliderConverterFactory getSliderConverterFactory();
```

Liefert eine Factory für Slider Converter. Sie auch Das Slider Viewer Widget

3.1.2.7 RootFrame Erzeugung

```
IFrame createRootFrame(IFrameDescriptor descriptor);
```

Erzeugt ein root Frame für ein Frame Descriptor (BluePrint).

```
IFrame createRootFrame(IFrameDescriptor descriptor, IApplicationLifecycle lifecycle);
```

Erzeugt ein root Frame für ein Frame Descriptor (BluePrint). Zusätzlich wird ein Window Listener auf dem erzeugten Frame hinzugefügt, welches auf dem lifecycle die Methode `finish()` aufruft, sobald das Frame geschlossen wird.

3.1.2.8 Generic Widget Factory

```
IGenericWidgetFactory getWidgetFactory();
```

Liefert die Widget Factory von jowidgets. Siehe auch Die Generic Widget Factory

3.1.2.9 WidgetWrapperFactory

```
IWidgetWrapperFactory getWidgetWrapperFactory();
```

Die WidgetWrapperFactory kann verwendet werden, um aus nativen Widgets (z.B. JFrame, Shell, JPanel, Composite) Wrapper zu erstellen, welche die zugehörigen jowidgets Schnittstellen implementieren (z.B. IFrame und IComposite). Siehe auch Jowidgets Code in native Projekte integrieren.

3.1.2.10 Images

```
IImageFactory getImageFactory();

IImageRegistry getImageRegistry();
```

Liefert die Image Factory und Image Registry. Siehe auch Icons und Images.

3.1.2.11 Dialog Panes

```
IMessagePane getMessagePane();
```

Das Message Pane liefert einen vereinfachten Zugriff auf den MessageDialog.

```
IQuestionPane getQuestionPane();
```

Das Question Pane liefert einen vereinfachten Zugriff auf den QuestionDialog.

```
ILoginPane getLoginPane();
```

Das Login Pane liefert einen vereinfachten Zugriff auf den LoginDialog.

3.1.2.12 Layouting

```
ILayoutFactoryProvider getLayoutFactoryProvider();
```

Der Layout Factory Provider bietet einen Zugriff auf verschiedene Layouts. Siehe auch Layouting

3.1.2.13 ActionBuilderFactory

```
IActionBuilderFactory getActionBuilderFactory();
```

Stellt die Action Builder Factory zur Verfügung. Siehe auch Actions und Commands

3.1.2.14 Default Actions

```
IDefaultActionFactory getDefaultActionFactory();
```

Liefert diverse default Actions. Derzeit existieren ausschließlich default Actions für Bäume. Siehe daher auch Das Tree Widget

3.1.2.15 ModelFactory

```
IModelFactoryProvider getModelFactoryProvider();
```

Liefert diverse Model Factories für Menu Items, die Tabelle und das Levelmeter Widget

3.1.2.16 TextMaskBuilder

```
ITextMaskBuilder createTextMaskBuilder();
```

Liefert einen Builder für Textmasken. Siehe auch Maskierte Texteingaben

3.1.2.17 InputContentCreator

```
IInputContentCreatorFactory getInputContentCreatorFactory();
```

Liefert spezielle Content Creator für ein InputComposite oder ein InputDialog

3.1.2.18 WaitAnimation

```
IWaitAnimationProcessor getWaitAnimationProcessor();
```

Liefert Zugriff auf die Wait Animation

3.1.2.19 AnimationRunnerBuilder

```
IAnimationRunnerBuilder getAnimationRunnerBuilder();
```

Erzeugt einen Builder für einen `AnimationRunner`. Siehe auch Animationen

3.1.2.20 Delayed Events

```
IDelayedEventRunnerBuilder getDelayedEventRunnerBuilder();
```

Erzeugt einen Builder für einen `DelayedEventRunner`. Siehe auch Verzögerte Events

3.1.2.21 Drag and Drop / Copy and Paste

```
IClipboard getClipboard();
```

Liefert Zugriff auf das System Clipboard. Siehe auch Copy and Paste und Drag and Drop.

```
ITransferableBuilder createTransferableBuilder();
```

Erzeugt einen `TransferableBuilder`. Siehe auch Copy and Paste und Drag and Drop.

3.1.2.22 Widget Utils

```
IWidgetUtils getWidgetUtils();
```

Liefert Zugriff auf die Widget Utils

```
IWindow getActiveWindow();
```

Gibt das aktive Fenster zurück. U.A. hilfreich bei der Erzeugung von Dialogen, welche über dem aktiven Fenster geöffnet werden sollen.

```
List<IWindow> getAllWindows();
```

Gibt eine Liste aller Fenster zurück.

3.1.2.23 Toolkit Properties

```
<VALUE_TYPE> void setValue(ITypedKey<VALUE_TYPE> key, VALUE_TYPE value);

<VALUE_TYPE> VALUE_TYPE getValue(ITypedKey<VALUE_TYPE> key);
```

Mit Hilfe dieser Methoden kann man Properties auf dem Toolkit setzen und auslesen. Wenn man jowidgets Applikation schreibt, die auch Web kompatibel sein sollen, muss man sehr vorsichtig mit Singletons sein, da sich die JVM mehrere Nutzer teilen. Ansonsten könnte sich das Ändern eines globalen Zustands eines Nutzers auch auf einen anderen Nutzer auswirken.

Falls man dennoch *globale Variablen* benötigt, zum Beispiel zum Speichern des Security Context oder ähnlichem, kann man mit Hilfe dieser Methoden ein *Session Singleton* realisieren, da pro User Session genau ein Toolkit existiert. Siehe zudem auch Typed Properties.

3.1.2.24 Umrechnungsmethoden

```
    Position toScreen(final Position localPosition, final IComponent component);
```

Transformiert eine lokale Position einer Komponente in eine absolute Bildschirmposition.

```
    Position toLocal(final Position screenPosition, final IComponent component);
```

Transformiert eine Bildschirmposition in eine lokale Position einer Komponente.

3.1.2.25 SupportedWidgets

```
    ISupportedWidgets getSupportedWidgets();
```

Liefert die Info, ob bestimmte Widgets unterstützt werden. Derzeit betrift dies ausschließlich den FileChooser und DirectoryChooser. Diese werden in Webapplikationen nicht unterstützt.

3.1.2.26 SpiMigLayoutSupport

```
    boolean hasSpiMigLayoutSupport();
```

Liefert die Information, ob die verwendete SPI Implementierung eine native Mig Layout Implementierung bietet. Nur für interne Zwecke relevant. Siehe dazu auch Mig Layout

3.2 Der Application Runner

Der Application Runner dienst zum Starten einer jowidgets standalone Applikation. Standalone bedeutet, dass die initialen Widgets sowie der Event Dispatcher Thread über die Jowidget API erzeugt werden.

Die Schnittstelle `IApplicationRunner` sieht wie folgt aus:

```
1  package org.jowidgets.common.application;
2
3  public interface IApplicationRunner {
4
5      void run(IApplication application);
6
7  }
```

Eine Implementierung erhält man vom Toolkit. Die Methode `run()` blockiert, bis die Applikation beendet wurde.

Die Schnittstelle `IApplication` wird *selbst* implementiert. Siehe dazu auch HelloWorldApplication - Der common Ui Code

```java
package org.jowidgets.common.application;

public interface IApplication {

    void start(final IApplicationLifecycle lifecycle);

}
```

In der `start()` Methode wird ein `IApplicationLifecycle` übergeben.

```java
package org.jowidgets.common.application;

public interface IApplicationLifecycle {

    void finish();

}
```

Wird auf dem `IApplicationLifecycle` die Methode `finish()` aufgerufen, wird die Applikation beendet. Dabei werden alle Widgets des zugehörigen Toolkit disposed.

Wie man jowidgets (ohne ApplicationRunner) in nativen (Swing, Swt, Rwt, ...) Code integriert findet sich im Abschnitt Jowidgets Code in native Projekte integrieren.

3.3 Der Ui Thread Access

Wie in anderen UI Frameworks (wie Swing oder Swt) gibt es auch in jowidgets einen *UI Thread*, in welchem alle UI Events *dispatched* werden. Alle Modifikationen an Widgets oder deren *Models* müssen in diesem Thread erfolgen. Reagiert man auf Events, welche durch Nutzereingaben ausgelöst wurden, befindet man sich bereits automatisch im UI Thread. Dabei ist, wie auch bei anderen UI Frameworks, darauf zu achten, den UI Thread nicht für lange Berechnungen zu blockieren, weil ansonsten keine Events mehr verarbeitet werden können und die Oberfläche für diese Zeit *einfriert*.

Operationen die potentiell lange dauern können, wie Beispielsweise das Aufrufen eines Service, sollten daher immer in einem eigenen Thread stattfinden.

Will man aus einem anderen Thread heraus wieder Methoden auf UI Elementen aufrufen, bietet die Schnittstelle `IUiThreadAccess` dafür die folgenden Methoden:

```
void invokeLater(Runnable runnable);

void invokeAndWait(Runnable runnable) throws InterruptedException;
```

Die Methode `invokeLater()` führt das übergebene `Runnable` zu einem späteren Zeitpunkt im Ui Thread aus. Nachdem die Methode zurückkehrt, kann man jedoch nicht davon ausgehen, dass das Runnable bereits ausgeführt wurde.

Die Methode `invokeAndWait()` führt auch das Runnable im Ui Thread aus, blockiert aber so lange, bis das `Runnable` ausgeführt wurde. Diese Methode ist mit höchster Vorsicht zu verwenden. Bei falscher Verwendung erzeugt man dadurch sehr schnell einen *Deadlock*. Statt blockierenden Methodenaufrufen sollte man besser mit Callback's arbeiten.

Beide Methoden können in jedem beliebigen Thread aufgerufen werden.

Um zu prüfen, ob der aktuelle Thread der UI Thread ist, kann man die folgende Methode verwenden:

```
boolean isUiThread();
```

Der Aufruf:

```
IUiThreadAccess uiThreadAccess = Toolkit.getUiThreadAccess();
```

muss immer im UI Thread erfolgen. Dies liegt unter anderem daran, dass es im Web Kontext mehrere UI Threads (einer pro User Session) existieren, und ein beliebiger Thread nicht wissen kann, welcher UI Thread verwendet werden soll.

In der Praxis übergibt man die `IUiThreadAccess` Instanz einfach an den anderen Thread. Folgendes Beispiel zeigt, wie man das einfach mit Hilfe des Ausführungsstacks machen kann:

```
1  //Add an listener to the button that does a long lasting calculation
2  button.addActionListener(new IActionListener() {
3      @Override
4      public void actionPerformed() {
5
6          //this will be invoked in the ui thread
7          final IUiThreadAccess uiThreadAccess = Toolkit.getUiThreadAccess();
8
9          //execute the service calculation in an service thread
10         executorService.execute(new Runnable() {
11
12             @Override
13             public void run() {
14                 //this will be invoked in the service thread
15                 final ServiceResult result = longLastingService.doLongLastingCalculation();
16
17                 uiThreadAccess.invokeLater(new Runnable() {
18                     @Override
19                     public void run() {
20                         //this will be executed in the ui thread
21                         serviceResultWidget.showResult(result);
22                     }
23                 });
24             }
25
26         });
27     }
28 });
```

Hinweis: Das obige Beispiel wurde bewusst vereinfacht. Es fehlen u.A. wichtige Aspekte wie Fehlerbehandlung oder das Abbrechen der Berechnung.

3.4 BluePrints - Übersicht

BluePrints werden benötigt, um Widgets zu erzeugen.

Mit Hilfe eines BluePrint wird das (Default) Setup eines Widgets festgelegt. Wenn ein Widget erzeugt wird, werden alle Eigenschaften, welche auf dem Setup definiert sind, für das Widget übernommen. Einige Parameter eines Setups sind Pflichtparameter. Diese sind sind mit der `@Mandatory` Annotation gekennzeichnet. Für Pflichtparameter existieren, wenn dies *sinnvoll* möglich ist, Defaultwerte, welche beliebig überschrieben werden können. Siehe dazu auch Widget Defaults. Die Setter Methoden eines BluePrint haben, wie beim Builder Pattern üblich, immer die Instanz als Rückgabewert. Dadurch lassen sich die Methodenaufrufe einfach verketten.

Ein BluePrint kann Eigenschaften enthalten, welche für das Widget nicht mehr veränderbar sind. Zum Beispiel muss die `Orientation` eines `SplitComposite` initial auf horizontal oder vertikal festgelegt werden, und ist nachträglich nicht mehr änderbar. Durch die klare Trennung der Setup und Widget Schnittstellen lassen sich (auch für eigene Widgets) sogenannte imutable Member einfach umsetzen, ohne dabei lange Parameterlisten in Konstruktoren zu benötigen. Dies kann die Implementierung eines Widgets erleichtern, weil nicht alle Eigenschaften modifizierbar implementiert werden müssen.

Die Frage, ob eine Eigenschaft imutable ist oder nicht ist eine Designfrage. Man könnte sich zum Beispiel auf den Standpunkt stellen, dass ein Fenster entweder *closeable* ist oder nicht. Diesen Zustand während der Anzeige des Fensters zu ändern wäre ungewöhnlich, daher kann man sich den Implementierungsaufwand dafür auch einsparen. Das man den Titel eines Fensters zur Laufzeit ändert, ist jedoch üblich. Beispielsweise zeigt das Explorer Fenster von Windows immer den ausgewählten Ordner im Titel an. Diese Eigenschaft sollte also eher veränderbar entworfen werden. Das Prinzip, dass Widgets unveränderbare Eigenschaften haben, findet sich auch bei SWT wieder. Dort werden diese über den Style, welcher eine Bitmaske darstellt, gesetzt.

Für die Erstellung eigener Widget Bibliotheken kann der BluePrint Mechanismus verwendet werden. Dabei müssen die BluePrint Schnittstellen nicht selbst implementiert werden, da die Implementierung von der Blue Print Proxy Factory mit Hilfe von Java Proxies umgesetzt wird. Das Definieren der BluePrint Schnittstelle reicht also aus.

3.4.1 Die BluePrint Factory

Für die Core Widgets kann man sich die Instanzen der BluePrints von der BluePrintFactory erzeugen lassen. Diese erhält man vom Toolkit wie folgt:

```
IBluePrintFactory bluePrintFactory = Toolkit.getBluePrintFactory();
```

Andere Widget Bibliotheken oder auch die Addon Widgets stellen jeweils ihre eigene BluePrint Factory zur Verfügung. Die Klasse BrowserBPF liefert zum Beispiel die BluePrints für die Browser Widgets.

3.4.1.1 Die Abbreviation Accessor Klasse BPF

Alle Methoden der Schnittstelle IBluePrintFactory sind auch über die statische Abbreviation Accessor Klasse org.jowidgets.tools.widgets.blueprint.BPF verfügbar. Dadurch kann man zum Beispiel anstatt:

```
final IFrameBluePrint frameBp = Toolkit.getBluePrintFactory().frame();
```

auch das Folgende schreiben:

```
final IFrameBluePrint frameBp = BPF.frame();
```

3.4.1.2 Beispiel für die Verwendung des IFrameBluePrint

Im folgenden Beispiel wird ein BluePrint für ein Frame mit dem Titel “My Frame” erzeugt, welches immer beim Anzeigen zentriert werden soll (bezüglich des Vaterfensters, und falls es ein Root Fenster ist, bezüglich des Bildschirms), welches beim Schließen automatisch `disposed` wird und dessen Größe nicht durch den Nutzer geändert werden kann:

```
final IFrameBluePrint frameBp = BPF.frame();
frameBp
    .setTitle("My Frame")
    .setAutoCenterPolicy(AutoCenterPolicy.ALWAYS)
    .setAutoDispose(true)
    .setResizable(false);
```

Mit diesem Frame BluePrint kann man zum Beispiel mit Hilfe des Toolkit ein Root Frame

```
IFrame frame = Toolkit.createRootFrame(frameBp);
```

oder auch wie folgt ein Kind Fenster erzeugen

```
IFrame childFrame = frame.createChildWindow(frameBp);
```

3.4.1.3 Beispiel für die Verwendung des ISliderViewerBluePrint

Im folgenden Beispiel wird ein BluePrint für ein `SliderViewer` erzeugt, welcher `Double` Werte von 0.0 bis 1.0 annimmt, vertikal ausgerichtet ist, keine Scala anzeigt, das Tooltip "Brigthness" hat und an den `ObservabelValue` `brightness` gebunden ist.

```
final ISliderViewerBluePrint<Double> sliderViewerBp = BPF.sliderViewer();
sliderViewerBp
    .setConverter(SliderConverterFactory.linearConverter(1.0))
    .setOrientation(Orientation.VERTICAL)
    .setRenderTicks(false)
    .setToolTipText("Brigthness")
    .setObservableValue(brightness);
```

Mit Hilfe dieses BluePrint könnte zum Beispiel wie folgt ein `SliderViewer` Widget erzeugt werden:

```
final ISliderViewer<Double> sliderViewer = frame.add(sliderViewerBp);
```

3.4.1.4 Beispiele für die Verwendung des ILabelBluePrint

Das folgende Beispiel zeigt die Wiederverwendung eines Label BluePrint:

```
final ILabelBluePrint labelBp   = BPF.label();
labelBp
    .setAlignment(AlignmentHorizontal.RIGHT)
    .setForegroundColor(Colors.ERROR)
    .setIcon(IconsSmall.ERROR);

container.add(labelBp.setText("Error 1"));
container.add(labelBp.setText("Error 2"));
container.add(labelBp.setText("Error 3"));
```

Das Alignment, die Farbe sowie das Icon wird nur ein Mal definiert und daraus anschließend drei Label Widgets mit unterschiedlichem Text erzeugt.

3.4.2 Setup, Setup Builder, Descriptor und Widget Schnittstellen

Ein Widget (der Typ, nicht die Instanz) ist durch seine BluePrint Schnittstelle und seine Widget Schnittstelle eindeutig festgelegt.

Die Widget Schnittstelle definiert die Schnittstelle des Widgets, die nach seiner Erzeugung (zum Beispiel für Modifikationen, Abfragen und Funktionen) zur Verfügung steht.

Die BluePrint Schnittstelle vereint die folgenden Aspekte:

- Widget Setup
- Widget Descriptor
- Widget Setup Builder

Für die Core Widgets sind diese Aspekte in unterschiedliche Schnittstellen aufgeteilt.[1]

3.4.2.1 Label Widget Beispiel

Die Aufteilung der Schnittstellen soll anhand des Label Widgets verdeutlicht werden. Ein Label Widget setzt sich aus einem Icon und einem Text Label zusammen. Ein Label Widget hat die BluePrint Schnittstelle `ILabelBluePrint` und die Widget Schnittstelle `ILabel`.

3.4.2.2 Widget Schnittstelle

Die Widget Schnittstelle legt den Vertrag des Widgets (nach seiner Erzeugung) fest. Ein Widget Interface muss mindestens von IWidget abgeleitet sein. Widgets welche zu einem Container hinzugefügt werden sollen, sind mindestens von IControl abgeleitet usw..

Die Schnittstelle `ILabel` ist von `ITextLabel` und `IIcon` abgeleitet, welche wiederum von IControl, IComponent und IWidget abgeleitet sind, und hat damit die folgenden Methoden:

```
1    //inherited from IIcon
2
3    void setIcon(IImageConstant icon);
4
5    IImageConstant getIcon();
6
7    //inherited from ITextLabel
8
9    void setFontSize(int size);
10
11   void setFontName(String fontName);
12
13   void setMarkup(Markup markup);
14
15   void setText(String text);
16
17   String getText();
18
19   //inherited from IWidget
20
```

[1] Dies muss bei der Erstellung eigener Widget Bibliotheken nicht zwingend so gemacht werden. Das Zusammenfassung von Setup und Setup Builder in eine Schnittstelle hat sich durchaus als praktikabel erwiesen und wird zum Beispiel auch bei den Widgets der jo-client-platform so umgesetzt.

```
21      Object getUiReference()
22
23      //...and a lot more
24
25      //inherited from IComponent
26
27      void setForegroundColor(final IColorConstant colorValue);
28
29      //...and a lot more
30
31      //inherited from IControl
32
33      void setToolTipText(String toolTip);
34
35      //...and a lot more
```

Anmerkung: Die von IControl, IComponent und IWidget geerbten Methoden wurden dabei nicht vollständig aufgezählt.

3.4.2.3 Widget Setup

Das Widget Setup liefert der Widget Implementierung die Konfiguration eines Widgets. Ein Setup besteht immer aus Properties, welche mit `get()`, `is()` oder `has()` abgefragt werden können. Setup Schnittstellen können von anderen abgeleitet sein. Zum Beispiel leitet `ILabelSetup` von `IIconSetup` und von `ITextLabelSetup` ab, und hat keine zusätzlichen Methoden.

Ein `ILabelSetup` hat dadurch die folgenden Methoden:

```
IImageConstant getIcon();

String getText();

String getToolTipText();

@Mandatory
Markup getMarkup();

AlignmentHorizontal getAlignment();

Integer getFontSize();

String getFontName();

Boolean isVisible();

IColorConstant getForegroundColor();

IColorConstant getBackgroundColor();
```

Die `@Mandatory` gibt an, dass diese Properties nicht `null` sein dürfen. Wenn es (sinnvoll) möglich ist, haben solche Properties einen Defaultwert. Für das Markup und das Alignment ist ein sinvoller Default möglich (`Markup.DEFAULT` und `AlignmentHorizontal.LEFT`). Stellt eine Property zum Beispiel ein Interface dar, welches der Nutzer des Widgets implementiert (sozusagen als SPI), ist es unter umständen nicht möglich, dafür einen sinnvollen Default anzubieten. Dann führt das Weglassen dieser Property zu einem Fehler, wenn das Widget erzeugt wird.

3.4.2.4 Widget Setup Builder

Ein Widget Setup Builder liefert die Setter Methoden zu einen Widget Setup. Diese sind alle nach dem folgenden Muster aufgebaut:

```
BLUE_PRINT_TYPE set(PROPERTY_TYPE property);
```

Dabei ist `PROPERTY_TYPE` der Java Typ der Property, die gesetzt werden soll. Der BLUE_PRINT_TYPE ist der Typ des BluePrint, um verkettete Aufrufe zu ermöglichen.

Die Schnittstelle `ILabelSetupBuilder` ist von `IIconSetupBuilder` und `ITextLabelSetupBuilder` abgeleitet und hat die folgenden einfachen Setter Methoden:

```
BLUE_PRINT_TYPE setIcon(IImageConstant icon);

BLUE_PRINT_TYPE setText(String text);

BLUE_PRINT_TYPE setToolTipText(String text);

BLUE_PRINT_TYPE setMarkup(Markup markup);

BLUE_PRINT_TYPE setAlignment(AlignmentHorizontal alignmentHorizontal);

BLUE_PRINT_TYPE setFontSize(Integer size);

BLUE_PRINT_TYPE setFontName(String fontName);

BLUE_PRINT_TYPE setVisible(Boolean visible);

BLUE_PRINT_TYPE setForegroundColor(final IColorConstant foregroundColor);

BLUE_PRINT_TYPE setBackgroundColor(final IColorConstant backgroundColor);
```

Zusätzlich zu den Bean Property Setter Methoden kann ein Setup Builder weitere Convenience Methoden haben. Die Schnittstelle `ILabelSetupBuilder` hat folgende Convenience Methoden:

```
INSTANCE_TYPE alignLeft();

INSTANCE_TYPE alignCenter();

INSTANCE_TYPE alignRight();

INSTANCE_TYPE setStrong();

INSTANCE_TYPE setEmphasized();
```

Diese Methoden ermöglichen eine verkürzte Schreibweise. So kann man zum Beispiel anstatt:

```
final ILabelBluePrint labelBp =
    BPF.label().setAlignment(AlignmentHorizontal.RIGHT).setMarkup(Markup.STRONG);
```

auch

```
 final ILabelBluePrint labelBp = BPF.label().alignRight().setStrong();
```

Sowohl die Getter Methoden eines Widget Setup als auch die einfachen Setter Methoden eines Widget Setup Builders werden mit Hilfe eines Java Proxy implementiert. Einen solchen BluePrintProxy erhält

man von der BluePrint Proxy Factory. Insbesondere bei der Erstellung eigener Widget Bibliotheken reicht also die Definition der Setup und Setup Builder Schnittstelle aus. Die Convenience Methoden sind durch (eine oder mehrere) eigene Schnittstellen definiert. Eine Implementierung der Convenience Schnittstelle muss bei der BluePrint Proxy Factory registriert werden.

Die Widget Setup Builder von jowidgets sind von der Schnittstelle `ISetupBuilder` abgeleitet. Diese hat die folgende Methode:

```
    INSTANCE_TYPE setSetup(IComponentSetupCommon descriptor);
```

Dadurch kann ein anderes Setup, welches auch einen anderen Typ haben kann, auf dem Builder gesetzt werden. Dabei werden alle Properties mit gleichen Namen und gleichem Typ auf dem Builder gesetzt. Auf diese Weise lassen sich leicht Kopien von Setups erzeugen.

3.4.2.5 Widget Descriptor

Jeder Widget Typ benötigt seinen eigenen Widget Descriptor Typ, welcher mit Hilfe einer Schnittstelle definiert wird, die von `IWidgetDescriptor` abgeleitet ist.

Die Schnittstelle `ILabelDescriptor` ist zum Beispiel wie folgt definiert:

```
    public interface ILabelDescriptor extends ILabelSetup, IWidgetDescriptor<ILabel> {}
```

Für die Schnittstelle ILabelDescriptor kann genau eine Implementierung (zur selben Zeit) in der Generic Widget Factory registriert sein. Der Widget Descriptor ist gleichzeitig auch ein Widget Setup (im konkreten Fall ein `ILabelSetup`). Ein Widget Descriptor liefert somit alles, was von der Generic Widget Factory für die Erzeugung eines Widget benötigt wird.

3.4.2.6 BluePrint Schnittstelle

Die BluePrint Schnittstelle fügt dem Widget Descriptor noch den Widget Setup Builder Aspekt hinzu.

Die Schnittstelle `ILabelBluePrint` ist zum Beispiel wie folgt definiert:

```
    public interface ILabelBluePrint extends ILabelSetupBuilder<ILabelBluePrint>, ILabelDescriptor {}
```

Hinweis: Für jedes Widget existiert genau eine eigene BluePrint Schnittstelle, während unterschiedliche Widgets sich die gleiche Widget Schnittstelle Teilen können (z.B. `IFrame` für das Frame Widget (`IFrameBluePrint`) und das Dialog Widget (`IDialogBluePrint`). Die Hierarchien der Setup und Widget Schnittstellen müssen nicht zwingend gleich sein. So ist ein `LoginDialogSetup` zum Beispiel vom `ITitledWindowSetup` abgeleitet, obwohl ein `ILoginDialog` nicht von `IWindow` abgeleitet ist.

Hinweis: Der Begriff BluePrint wird oft stellvertretend für den Begriff WidgetDescriptor verwendet. Dies liegt unter anderem daran, dass ein BluePrint einen Widget Typ genauso eindeutig spezifiziert wie ein WidgetDescriptor und in der Praxis beide Aspekte oft in einer Schnittstelle vereint sind. Ein BluePrint fügt zu einem separierten (in einer eigenen Schnittstelle befindlichen) WidgetDescriptor noch den Aspekt des Setup Builder hinzu, weshalb der Begriff WidgetDescriptor nicht synonym für BluePrint verwendet werden sollte, wenn der Builder Aspekt in diesem Kontext auch relevant ist.

3.4.2.7 Zusammenfassen des Builder und Setup Aspekts

Die ursprüngliche Idee, den Builder und Setup Aspekt zu trennen, hat sich in der Praxis nicht bewährt. Es wird daher empfohlen, bei der Erstellung eigener Widget Bibliotheken die Builder und Setup Methoden in einer Schnittstelle unterzubringen. Für das Label würde man dann zum Beispiel die `ILabelSetupBuilder` Schnittstelle wie folgt definieren:

```
    IImageConstant getIcon();

    BLUE_PRINT_TYPE setIcon(IImageConstant icon);

    String getText();

    BLUE_PRINT_TYPE setText(String text);

    String getToolTipText();

    BLUE_PRINT_TYPE setToolTipText(String text);

    @Mandatory
    Markup getMarkup();

    BLUE_PRINT_TYPE setMarkup(Markup markup);

    AlignmentHorizontal getAlignment();

    BLUE_PRINT_TYPE setAlignment(AlignmentHorizontal alignmentHorizontal);

    Integer getFontSize();

    BLUE_PRINT_TYPE setFontSize(Integer size);

    String getFontName();

    BLUE_PRINT_TYPE setFontName(String fontName);

    Boolean isVisible();

    BLUE_PRINT_TYPE setVisible(Boolean visible);

    IColorConstant getForegroundColor();

    BLUE_PRINT_TYPE setForegroundColor(final IColorConstant foregroundColor);

    IColorConstant getBackgroundColor();
```

Dann wäre auch die Trennung von Descriptor und BluePrint überflüssig und die Schnittstelle `ILabelBluePrint` würde also wie folgt aussehen:

```
    public interface ILabelBluePrint extends
        ILabelSetupBuilder<ILabelBluePrint>,
        IWidgetDescriptor<ILabel> {}
```

Die Trennung der BluePrint Schnittstelle und er SetupBuilder Schnittstelle wird jedoch nach wie vor empfohlen, um die Möglichkeit zu haben, von einem Setup Builder ableiten zu können. Von einem BluePrint sollte nicht abgeleitet werden, unter Anderem weil für das abgeleitete BluePrint der Descriptor Typ nicht mehr eindeutig sein könnte.

3.4.3 Die BluePrint Proxy Factory

Die Blue Print Proxy Factory implementiert eine BluePrint Schnittstelle mit Hilfe von Java Proxies. Wenn man bei der Erstellung eigener Widget Bibliotheken die BluePrints auch auf diese Weise erzeugt, lassen sich dadurch die Default Setups einfach überschreiben.

Man erhält die Instanz der `IBluePrintProxyFactory` wie folgt von Toolkit:

```
IBluePrintProxyFactory bluePrintProxyFactory = Toolkit.getBluePrintProxyFactory();
```

Es folgt eine kurze Beschreibung der Methoden:

3.4.3.1 Erzeugen eines BluePrint Proxy

TODO

3.4.3.2 Hinzufügen oder Setzen eines Default Initializer

TODO

3.4.3.3 Setzen einer Convenience Methoden Implementierung

TODO

3.5 Die Validation API

Die Validation API bietet Schnittstellen und Funktionen für die Validierung von Objekten und findet zum Beispiel Verwendung beim InputField, dem InputComposite, dem InputDialog oder dem ValidationLabel. Darüber hinaus kann die Validation API auch für die Verwendung eigener Komponenten herangezogen werden.

Die Validation API befindet sich im Modul `org.jowidgets.validation`. Dieses hat weder jowidgets interne noch externe transitive Abhängigkeiten. Insbesondere hat die API dadurch auch keine Abhängigkeiten auf UI Aspekte und kann somit auch für die serverseitige Validierung herangezogen werden.

Sie bildet die Basis für die jo-client-platform Bean Validation, welche Adapter für die Javax Bean Validation (JSR 303) bereitstellt.

Im Vergleich zu einer Bean Validation API liefert die Validation API keine besonderen Aspekte bezüglich der Validierung von Properties eines Beans, sondern ausschließlich Aspekte für die Validierung von Objekten. Diese können natürlich sowohl Beans als Bean Properties sein. Die Validation API ist damit allgemeiner als eine Bean Validation API.

Die jowidgets Validation API unterstützt im Vergleich zur Javax Bean Validation API eine differenzierte Unterscheidung von Fehlertypen. Neben `ok` und `error` gibt es weitere Typen wie `warning`, `info`, etc. (siehe auch Message Types).

Die Validation API wurde nicht entworfen um Javax Bean Validation zu ersetzen, sondern um damit zu *koexistieren*. Javax Bean Validatoren lassen sich zum Beispiel einfach auf jo-client-platform Bean Validatoren adaptieren. Siehe zum Beispiel BeanPropertyValidatorAdapter

3.5.1 Die Schnittstelle IValidator

Ein Validator validiert eine oder mehrere Bedingungen für einen gegebenen Wert. Die Schnittstelle sieht wie folgt aus:

```
1  public interface IValidator<VALUE_TYPE> {
2
3      IValidationResult validate(VALUE_TYPE value);
4
5  }
```

Der zu validierende Wert kann `null` sein. Das Ergebnis einer Validierung muss **ungleich null** sein. Ein Validation Result besteht aus einer Liste von Validation Messages welche einen Message Type haben.

Das folgende Beispiel implementiert einen `NotNullValidator`:

```
1  public final class NotNullValidator<VALUE_TYPE> implements IValidator<VALUE_TYPE> {
2
3      @Override
4      public IValidationResult validate(final VALUE_TYPE value) {
5          if (value == null){
6              return ValidationResult.error("Must not be null");
7          }
8          else{
9              return ValidationResult.ok();
10         }
11     }
12
13 }
```

3.5.1.1 OkValidator

Die statische Accessor Klasse `Validator` liefert einen OkValidator mit Hilfe der folgenden statische Methode:

```
    public static <VALUE_TYPE> IValidator<VALUE_TYPE> okValidator() {...}
```

Dieser Validator liefert für alle Werte `ValidationResult.ok()` zurück.

3.5.2 Message Type

Die Enum `org.jowidgets.validation.MessageType` liefert die möglichen Message Typen. Diese werden in der folgenden Tabelle dargestellt:

Type	Valid
OK	yes
INFO	yes
WARNING	yes
INFO_ERROR	no
ERROR	no

Die Message Typen sind nach ihrem Schweregrad (Severity) sortiert, wobei `Error` der *schwerste* Fehler ist. In der Spalte *Valid* wird angegeben, ob der validierte Wert als *valide* einstuft wird. Valide Werte können weiter verarbeitet werden, zum Beispiel indem sie in die Datenbank geschrieben, oder für eine andersartige Transaktion verwendet werden. Nicht valide Werte sind abzulehnen.

Es folgt eine kurze Beschreibung der Semantik der einzelnen Message Typen:

- Ein Validation Result hat gar keine oder genau eine **OK** Message ohne Text, wenn die Validierung erfolgreich war und es keine weiteren Informationen oder Warnungen zur Validierung gibt. Per Konvention hat eine OK Message keinen Message Text und Context.
- Der Typ **INFO** kann verwendet werden, um Informationen anzuzeigen, wie zum Beispiel: “Die Angabe ist freiwillig”, “Eingabe korrekt” oder “Gut gemacht!”.
- Messages vom Typ **WARNING** können verwendet werden, wenn der Wert zwar (technisch) valide ist, es sich aber eventuell um einen Irrtum handelt, zum Beispiel weil der Wert ungewöhnlich erscheint. Auch kann der Nutzer über zusätzliche Folgen gewarnt werden, falls die Eingabe so übernommen wird. Beispiele für Message Texte sind: “Möchten die wirklich 100 Fernseher bestellen?”, “Die Maße liegen außerhalb des Normbereichs”, “Bei dieser Bestellmenge fallen zusätzliche Gebühren an!”, “Für diesen Artikel kann keine UsSt ausgewiesen werden!”.
- Der Typ **INFO_ERROR** kann für Fehler verwendet werden, bei denen der Wert zwar technisch nicht valide ist (und somit auch nicht akzeptiert wird), der Nutzer aber bisher nichts falsch gemacht hat. Dies gilt zum Beispiel für Pflichtfelder, deren Editierung noch nicht begonnen wurde oder für Eingaben, die zum Beispiel durch Ergänzung noch richtig werden könnten. Beispiele wären: “Pflichtangabe”, “Bitte vervollständigen Sie die Eingabe”.
- Der Typ **ERROR** gilt für alle anderen Fehler.

Die Typen `INFO` und `INFO_ERROR` wurden Aufgrund von Kundenfeedback eingeführt. So wollten Kunden neben negativen auch positives Feedback beim Ausfüllen einer Maske haben, um sich bestätigt zu fühlen, alles richtig gemacht zu haben. Des Weiteren wurde bemängelt, dass eine Maske zum Erzeugen eines Datensatzes mit Pflichtfeldern bereits beim Öffnen mit etlichen Fehlern angezeigt wurde, obwohl man offensichtlich doch noch gar nichts falsch gemacht habe. Meldungen vom Typ `INFO_ERROR` enthalten daher eher Information, was zu tun ist, damit es richtig wird und werden in der Regel (konfigurierbar) nicht mit roter Farbe angezeigt. Meldungen von Typ `ERROR` geben eher an, was falsch ist. Ein gutes Beispiel für einen **schlechten** `ERROR` Text wäre: “Geben sie einen gültigen Wert ein”, besser wäre “Es sind nicht mehr als 20 Zeichen erlaubt”.

Die Existenz der Fehlertypen bedeutet nicht, dass auch allen Typen Verwendung finden müssen. Für einige Anwendungsfälle reicht eventuell `OK==richtig` und `ERROR==falsch` aus.

3.5.2.1 Der Valid Status

Die Enum `MessageType` bietet folgende Methode, um zu prüfen, ob der Typ zur Klasse der validen Messages gehört oder nicht.

```
public boolean isValid() {...}
```

Die Werte werden anhand der oben angezeigten Tabelle zurückgegeben.

3.5.2.2 Vergleichsmethoden der Enum MessageType

Die Enum `MessageType` bietet folgende Methoden, um den Schweregrad (Severity) zweier Message Types zu vergleichen.

```
    public boolean equalOrWorse(final MessageType messageType) {...}

    public boolean worse(final MessageType messageType) {...}
```

Die Methode `equalOrWorse` liefert `true` zurück, falls der MessageType die gleiche oder eine höhere Severity hat, als der übergebene. Die Methode `worse()` liefert `true` zurück, falls der MessageType eine höhere Severity hat, als der übergebene.

Beispiel:

```
    System.out.println(MessageType.ERROR.equalOrWorse(MessageType.WARNING));
    System.out.println(MessageType.WARNING.equalOrWorse(MessageType.ERROR));
```

Ergebnis:

```
    true
    false
```

3.5.3 Validation Message

Eine `IValidationMessage` stellt eine einzelne Nachricht eines `IValidationResult` bereit. Eine Validation Message ist imutable, das heißt die Properties können nachträglich nicht mehr geändert werden. Die Schnittstelle sieht wie folgt aus:

```
1  public interface IValidationMessage {
2
3      MessageType getType();
4
5      String getText();
6
7      String getContext();
8
9      IValidationMessage withContext(String context);
10
11     boolean equalOrWorse(final IValidationMessage message);
12
13     boolean worse(final IValidationMessage message);
14 }
```

Die Methode `getType()` liefert den `MessageType` zurück. Dieser ist nie `null`. Der Message Text (`getText()`) liefert den eigentliche Fehler oder Info Text. Dieser kann `null` sein, was jedoch nur für Fehler vom Typ `OK` angeraten wird. Der `context` gibt an, wo der Fehler aufgetreten ist. Das kann zum Beispiel der Name des Attributes einer Eingabemaske sein. Auch der Context kann `null` sein.

Mit Hilfe der Methode withContext() kann eine Kopie der Message erstellt werden, welche den übergeben `context` als neuen `context` bekommt.

Die Methode `equalOrWorse()` und `worse()` vergleichen den Schweregrad (Severity) zweier Messages, analog zu den Methoden gleichen Namens auf der Enum MessageType.

3.5.4 Validation Result

Ein Validation Result bündelt das Ergebnis einer Validierung und besteht aus 0 bis n Validation Messages, welche alle einen unterschiedlichen Message Type haben können. Ein ValidationResult ist imutable, wodurch sichergestellt ist, dass sich ein einmal ausgewertetes Ergebnis nicht mehr ändern kann.

3.5.4.1 Zugriff auf die Validation Messages

Für den Zugriff auf die einzelnen Messages bietet die Schnittstelle `IValidationResult` die folgenden Methoden:

```
List<IValidationMessage> getAll();

List<IValidationMessage> getErrors();

List<IValidationMessage> getInfoErrors();

List<IValidationMessage> getWarnings();

List<IValidationMessage> getInfos();
```

Die Methode getAll() liefert alle Messages, die anderen Methoden liefern die Messages des entsprechenden Typs. Die Messages sind in der Reihenfolge angeordnet, wie sie hinzugefügt wurden. Die Ergebnisliste ist nicht modifizierbar (*unmodifieable*). Die Default Implementierung erzeugt die Listen *lazy* bei der ersten Anfrage. (Das gilt auch für die `All` Liste.) Falls man mit der First Worst Message auskommt, kann das Rechenleistung und Speicher sparen.

3.5.4.2 First Worst Message

In einigen Anwendungsfällen kann es ausreichen, nur die erste aufgetretenen Message mit höchstem Schweregrad zu kennen. Dazu kann die folgende Methode auf einem `IValidationResult` verwendet werden:

```
IValidationMessage getWorstFirst();
```

Diese Methode liefert **immer** eine Message zurück. Gibt es keine **echten** Messages, wird eine Message vom Typ `OK` zurückgegeben. Diese hat per Konvention keinen Message Text und Message Context. Ansonsten wird die Message zurückgegeben, welche den höchsten Schweregrad hat. Existieren mehrere Messages mit diesem Schweregrad, wird die herangezogen, welche als erstes aufgetreten ist, bzw. dem Ergebnis hinzugefügt wurde.

Die Default Implementierung aktualisiert den Wert immer direkt beim Hinzufügen neuer Messages (Bei der Message Verkettung oder mit Hilfe des Validation Result Builder), so dass dieser nicht explizit berechnet werden muss.

Die Abfrage der First Worst Message ist also effizienter als die Verwendung der Message Listen

3.5.4.3 Gesamtergebnis

In bestimmten Fällen sind die eigentlichen Messages gar nicht relevant, sondern nur, ob das Ergebnis valide ist oder nicht. Dafür kann die folgende Methode verwendet werden:

```
boolean isValid();
```

Diese gibt `false` zurück, falls es mindestens eine Messages gibt, welche nicht valid ist und ansonsten `true`.

Die Methode:

```
boolean isOk();
```

liefert `false` zurück, falls es mindestens eine Message mit Schweregrad `INFO` oder höher gibt, und sonst `true`

3.5.4.4 Die Validation Result Accessor Klasse

Die Accessor Klasse `ValidationResult` liefert folgende statische Methoden zur Erzeugung eines `IValidationResult` mit genau einer Validation Message:

```
public static IValidationResult create() {...}

public static IValidationResult ok() {...}

public static IValidationResult create(final IValidationMessage message) {...}

public static IValidationResult warning(final String text) {...}

public static IValidationResult infoError(final String text) {...}

public static IValidationResult error(final String text){...}

public static IValidationResult warning(final String context, final String text) {...}

public static IValidationResult infoError(final String context, final String text) {...}

public static IValidationResult error(final String context, final String text){...}
```

Mit Hilfe der folgenden Methode kann ein `IValidationResultBuilder` erzeugt werden:

```
public static IValidationResultBuilder builder() {...}
```

3.5.4.5 Message Verkettung

Ein Validation Result ist immutable. Daher können zu einem Ergebnis nachträglich auch keine Messages hinzugefügt werden. Es ist jedoch möglich, zu einem Validation Result eine Message hinzuzufügen, indem man das Ergebnis kopiert und dabei die neue Nachricht hinzufügt. Dazu können die folgenden Methoden verwendet werden:

```
IValidationResult withMessage(final IValidationMessage message);

IValidationResult withError(final String text);

IValidationResult withInfoError(final String text);

IValidationResult withWarning(final String text);
```

```
IValidationResult withInfo(final String text);

IValidationResult withError(final String context, final String text);

IValidationResult withInfoError(final String context, final String text);

IValidationResult withWarning(final String context, final String text);

IValidationResult withInfo(final String context, final String text);
```

Das resultierende Validation Result ist eine Kopie des aktuellen Validation Results, welchem die übergebene Message hinzugefügt wurde. Die Methode `withMessage()` verlangt eine `IValidationMessage`, die anderen Methoden sind Convenience Methoden, welche die Validation Message mit Hilfe des Parameter `text` (und optional `context`) erzeugen.

Das folgende Beispiel soll das verdeutlichen:

```
1 IValidationResult result = ValidationResult.create();
2 result = result.withInfo("Info message");
3 result = result.withError("Error message");
4 result = result.withWarning("Warn message");
```

Es werden dem initialen Validation Result (Zeile 1) drei weitere Messages hinzugefügt. Bei dieser Methode ist darauf zu achten, dass das `result` immer wieder neue zugewiesen werden muss. Um diese potentielle Fehlerquelle zu vermeiden, kann auch ein `IValidationResultBuilder` verwendet werden.

3.5.4.6 Ändern des Context

Mit Hilfe der folgenden Methode kann der Context für alle Validation Messages eines `IValidationResult` geändert werden, indem eine Kopie erzeugt wird und auf dieser der Context geändert wird:

```
IValidationResult withContext(final String context);
```

3.5.4.7 Validation Result Builder

Die Schnittstelle `IValidationResultBuilder` hat die folgenden Methoden:

```
IValidationResultBuilder addMessage(final IValidationMessage message);

IValidationResultBuilder addInfo(final String text);

IValidationResultBuilder addWarning(final String text);

IValidationResultBuilder addInfoError(final String text);

IValidationResultBuilder addError(final String text);

IValidationResultBuilder addInfo(final String context, final String text);

IValidationResultBuilder addWarning(final String context, final String text);

IValidationResultBuilder addInfoError(final String context, final String text);

IValidationResultBuilder addError(final String context, final String text);
```

```
IValidationResultBuilder addResult(final IValidationResult result);

IValidationResult build();
```

Der Builder liefert Methoden zum Hinzufügen von Validation Messages. Mit Hilfe der Methode `addResult()` kann man alle Messages eines anderen `IValidationResult` hinzufügen. Mit Hilfe der Methode `build()` wird das erzeugte `IValidationResult` zurückgegeben.

Das folgende Beispiel verwendet den `IValidationResultBuilder` für die Erzeugung eines Validation Result:

```
1  final IValidationResultBuilder builder = ValidationResult.builder();
2  builder
3      .addInfo("Info message")
4      .addError("Error message")
5      .addWarning("Warn message");
6
7  final IValidationResult result = builder.build();
```

Das resultierende Validation Result ist identisch mit dem obigen Beispiel bei der Message Verkettung. Der Vorteil beim Builder Ansatz ist, dass die Neuzuweisung entfällt (welche bei der Message Verkettung versehentlich vergessen werden könnte).

3.5.5 Validator Composite

Ein Validator Composite ist ein `IValidator`, welche mehrere `IValidator` verwendet, um seine `validate()` Methode zu implementieren. Die statische Accessor Klasss `ValidatorComposite` liefert folgende Methoden zur Erzeugung eines Validator Composite:

```
public static <VALUE_TYPE> IValidator<VALUE_TYPE> create(
    final IValidator<VALUE_TYPE> validator1,
    final IValidator<VALUE_TYPE> validator2) {...}

public static <VALUE_TYPE> IValidatorCompositeBuilder<VALUE_TYPE> builder() {...}
```

Die erste Methode erzeugt aus zwei Validatoren einen *neuen* `IValidator`. Dabei können sowohl `validator1` als auch `validator2` als auch beide `null` sein. Der resultierende Validator fügt die Validation Results beider Validatoren zu einem neuen Validation Result zusammen. Ist ein Parameter (`validator1`, `validator2`) `null` wird stellvertretend ein OkValidator verwendet.

Die Methode `builder()` liefert ein `IValidatorCompositeBuilder`. Dieser hat die folgenden Methoden:

```
IValidatorCompositeBuilder<VALUE_TYPE> add(IValidator<VALUE_TYPE> validator);

IValidatorCompositeBuilder<VALUE_TYPE> addAll(Iterable<? extends IValidator<VALUE_TYPE>> validators);

IValidator<VALUE_TYPE> build();
```

Die Methode `add()` fügt einen einzelen Validator hinzu, die Methode `addAll()` eine Liste von Validatoren. Die Methode build() erzeugt einen neuen Composite Validator. Das folgende Beispiel demonstriert die Verwendung:

```
    final IValidatorCompositeBuilder<Person> builder = ValidatorComposite.builder();
    builder
        .add(notNullValidator)
        .add(adultValidator)
        .add(maleValidator);

    IValidator<Person> maleAdultPersonValidator = builder.build();
```

3.5.6 IValidatable

Ein `IValidatable` ist ein Objekt, was in der Lage ist, seinen eigenen Zustand zu validieren. Die Schnittstelle sieht wie folgt aus:

```
public interface IValidateable {

    IValidationResult validate();

    void addValidationConditionListener(IValidationConditionListener listener);

    void removeValidationConditionListener(IValidationConditionListener listener);

}
```

Ein `IValidationConditionListener` wird aufgerufen, wenn sich die Bedingungen für die Validierung geändert haben, zum Beispiel weil der zu validierende Wert sich geändert hat, oder weil sich die Validierungsregeln geändert haben. Der Listener sieht wie folgt aus:

```
public interface IValidationConditionListener {

    void validationConditionsChanged();

}
```

3.6 Allgemeine Widget Schnittstellen

Der folgende Abschnitt enthält eine Übersicht über die allgemeinen Widget Schnittstellen und deren Methoden innerhalb der Widget Hierarchie. Für weitere Informationen sei auf die Jowidgets API Spezifikation verwiesen.

3.6.1 Die Schnittstelle IWidget

Alle Widgets implementieren die Schnittstelle `org.jowidgets.api.widgets.IWidget`. Es folgt eine kurze Übersicht über die wichtigsten Methoden:

3.6.1.1 Ui Referenz

```
    Object getUiReference();
```

Liefert die UI Referenz des Widgets. Der Typ hängt von der verwendeten SPI Implementierung ab. Beispielsweise wird für ein `org.jowidgets.api.widgets.IButton` bei Verwendung der Swing Spi Implementierung ein `javax.swing.JButton` und bei der Verwendung der SWT SPI Implementierung ein `org.eclipse.swt.widgets.Button` zurückgegeben.

Die Ui Referenz kann zum Beispiel verwendet werden, um *native* Widgets oder Funktionen *nativer* Widgets zu verwenden, welche in jowidgets nicht vorhanden sind. Allerdings hat man dann nicht mehr die Möglichkeit, den Code mit anderen nativen Ui Technologien zu verwenden.

Tipp: Wird solch eine natives Widget oder eine Funktion mehrfach verwendet, empfiehlt es sich, eine eigene Widget Schnittstelle dafür zu definieren. (Siehe dazu Erstellung eigener Widget Bibliotheken). Die Widget Implementierung kann dann vorerst nur für die benötigte *native* Technologie implementiert werden. Soll ein Modul, welches dieses Widget verwendet, später auch mit einer anderen Technologie verwendet werden, muss nur die Implementierung des Widgets angepasst werden, und nicht das Modul.

3.6.1.2 Enablement

```
void setEnabled(boolean enabled);

boolean isEnabled();
```

Ein Widget das *disabled* ist nimmt keine Nutzereingaben mehr an. Per default sind alle Widgets initial *enabled*.

3.6.1.3 Widget Parent

```
IWidget getParent();
```

Liefert das übergeordnete Widget oder null, falls das Widget ein *Root Element* ist.

3.6.1.4 Widget Root

```
IWidget getRoot();
```

Für zusammengesetzte (Composite) Widgets wird das *Root Widget* zurückgegeben, also zum Beispiel ein org.jowidgets.api.widgets.IComposite. Ansonsten wird das Widget selbst zurückgegeben.

3.6.1.5 Dispose Management

```
void dispose();

boolean isDisposed();

void addDisposeListener(IDisposeListener listener);

void removeDisposeListener(IDisposeListener listener);
```

Mit Hilfe der Methode `dispose()` kann ein Widgets *disposed* werden, wenn man es nicht mehr benötigt. Ein Widget, das *disposed* wurde, kann nicht mehr verwendet werden. Wird zum Beispiel ein Fenster *disposed* wird es auch geschlossen. Wird ein Control disposed, wird es auch aus seinem Container entfernt. Mit Hilfe eines `IDisposeListener` kann man sich als Observer registrieren, um über das *Dispose* eines Widgets informiert zu werden.

3.6.2 Die Schnittstelle IComponent

Die Schnittstelle `IComponent` stellt die gemeinsamen Funktionen für Fenster und Controls bereit. Die Schnittstelle `IComponent` erweitert `IWidget`.

Es folgt eine Übersicht der wichtigsten Methoden:

3.6.2.1 Neuzeichnen

```
void redraw();
```

Markiert die Komponente, dass sie zu nächstmöglichen Zeitpunkt neu gezeichnet werden muss.

```
void setRedrawEnabled(boolean enabled);
```

Deaktiviert, bzw. aktiviert das Neuzeichnen einer Komponente (und deren Kinder, falls vorhanden). Nachdem das Neuzeichnen deaktiviert wurde, werden alle anstehenden Paint Events verworfen, bis wieder `setRedrawEnabled(true)` aufgerufen wird.

Diese Funktion kann zum Beispiel verwendet werden, um *Flackereffekte* zu vermeiden, während ein `Container` umstrukturiert wird, also Kinder hinzukommen oder entfernt werden.

Das Deaktivieren mittels `setRedrawEnabled(false)` ist ein *Hinweis* der nicht für alle Platformen sowie SPI Implementierungen verfügbar ist. Beispielsweise hat diese Funktion unter Swing keinen Effekt. Glücklicherweise gibt es unter Swing aber auch keine Probleme mit *Flackern*, wenn Container umorganisiert werden.

Ein Aufruf von `setRedrawEnabled(true)` markiert zudem die Komponente, dass sie neu gezeichnet werden muss (siehe Methode redraw()).

3.6.2.2 Eingabefokus

In einer Fenster basierten Anwendung kann zur selben Zeit genau ein Control den Eingabefokus haben. Dieses Control empfängt dann die Tastatureingaben des Benutzers. Um den Eingabefokus zu erhalten, kann die folgenden Methode verwendet werden:

```
boolean requestFocus();
```

Der Aufruf der Methode garantiert nicht, dass der Eingabefokus tatsächlich zugewiesen wurde.

Mittels folgender Methode kann überprüft werden, ob eine Komponente den Eingabefokus besitzt.

```
boolean hasFocus();
```

Um sich benachrichtigen zu lassen, dass ein Fokuswechsel stattgefunden hat, kann ein `IFocusListener` verwendet werden:

```
void addFocusListener(IFocusListener listener);

void removeFocusListener(IFocusListener listener);
```

Dieser hat die Methoden:

```
void focusGained();

void focusLost();
```

3.6.2.3 Tastatureingaben

Um Tastatureingaben zur erhalten kann ein `IKeyListener` verwendet werden:

```
void addKeyListener(IKeyListener listener);

void removeKeyListener(IKeyListener listener);
```

Dieser hat die folgenden Methoden:

```
void keyPressed(final IKeyEvent event);

void keyReleased(final IKeyEvent event);
```

Die Klasse `org.jowidgets.tools.controller.KeyAdapter` implementiert die `IKeyListener` Schnittstelle mit leeren Methodenrümpfen und kann verwendet werden, falls nicht alle Methoden implementiert werden sollen.

Die Schnittstelle `IKeyEvent` hat die folgenden Methoden:

```
VirtualKey getVirtualKey();

Character getCharacter();

Character getResultingCharacter();

Set<Modifier> getModifier();
```

Die `enum VirtualKey` enthält die F-Tasten, Steuertasten wie ENTER, BACKSPACE, TAB, PFEILTASTEN, etc, sowie Zahlen und Buchstaben. Eine vollständige Übersicht über alle VirtualKeys findet sich in der API Spezifikation: http://www.jowidgets.org/api_doc/org/jowidgets/common/types/VirtualKey.html

Die Methode `getCharacter()` liefert das zugehörige Zeichen der Taste **ohne** Anwendung der Modifier. Auf einer deutschen Tastatur liefert somit `SHIFT + 1` das Zeichen `1` und **nicht** das Zeichen `!`.

Dagegen liefert die Methode `getResultingCharacter()` das zugehörige Zeichen **mit** Anwendung der Modifier. Im vorigen Beispiel also anstatt `1` das Zeichen `!`.

Die Methode `getModifier()` liefert ein `Set` der verwendeten Modifier Tasten (CTRL, SHIFT, ALT).

3.6.2.4 Mauseingaben

Um einfache Mauseingaben zur erhalten, kann ein IMouseListener verwendet werden:

```
void addMouseListener(IMouseListener listener);

void removeMouseListener(IMouseListener listener);
```

Ein `IMouseListener` hat die folgenden Methoden:

```
void mousePressed(IMouseButtonEvent event);

void mouseReleased(IMouseButtonEvent event);

void mouseDoubleClicked(IMouseButtonEvent event);

void mouseEnter(IMouseEvent event);

void mouseExit(IMouseEvent event);
```

Ein `IMouseEvent` liefert die Position des Ereignisses im Koordinatensystem der zugehörigen Komponente:

```
Position getPosition();
```

Ein `IMouseButtonEvent` liefert zusätzlich den Button sowie die Tastatur Modifier.

```
MouseButton getMouseButton();

Set<Modifier> getModifiers();
```

Die Klasse `org.jowidgets.tools.controller.MouseAdapter` implementiert die `IMouseListener` Schnittstelle mit leeren Methodenrümpfen und kann verwendet werden, falls nicht alle Methoden implementiert werden sollen.

Um Ereignisse zu Mausbewegungen zur erhalten, kann ein `IMouseMotionListener` verwendet werden:

```
void addMouseMotionListener(IMouseMotionListener listener);

void removeMouseMotionListener(IMouseMotionListener listener);
```

Dieser hat die folgenden Methoden:

```
void mouseMoved(IMouseEvent event);

void mouseDragged(IMouseButtonEvent event);
```

Die Methode `mouseMoved()` wird bei jeder Änderung der Mausposition aufgerufen, die Methode `mouseDragged()`, wenn dabei eine Maustaste *gedrückt* wird.

Die Klasse `org.jowidgets.tools.controller.MouseMotionAdapter` implementiert die `IMouseMotionListener` Schnittstelle mit leeren Methodenrümpfen und kann verwendet werden, falls nicht alle Methoden implementiert werden sollen.

3.6.2.5 Popup Events

Popup Events werden ausgelöst, wenn der Benutzer ein Kontextmenü (Popup Menü) öffnen möchte. Die notwendigen *Popup Trigger* können auf unterschiedlichen Betriebssystemen variieren. Unter Windows geschieht dies zum Beispiel durch einen `Rechtsklick` mit der Maus. Um Popup Events zu erhalten, kann ein `IPopupDetectionListener` verwendet werden:

```
void addPopupDetectionListener(IPopupDetectionListener listener);

void removePopupDetectionListener(IPopupDetectionListener listener);
```

Dieser hat die folgende Methode:

```
void popupDetected(Position position);
```

Die Methode übergibt die Position im Koordinatensystem der Komponente.

3.6.2.6 Popup Menus

Mit Hilfe der folgenden Methode kann ein Popup Menü erzeugt werden:

```
IPopupMenu createPopupMenu();
```

Ein Popup Menu implementiert die Schnittstelle `IMenu`, somit kann man zum Beispiel mit Hilfe der Methode `addItem()` Menüeinträge hinzufügen. Das Menü kann mit Hilfe der Methode `show()` angezeigt werden:

```
void show(Position position);
```

Dabei ist die Position im Koordinatensystem der Komponente anzugeben, für die das Popup Menü erzeugt wurde.

Folgendes Beispiel zeigt die Verwendung eines PopMenüs:

```
1  final IPopupMenu menu = frame.createPopupMenu();
2
3  final ISelectableMenuItem option = menu.addItem(BPF.checkedMenuItem("My option"));
4  option.addItemListener(new IItemStateListener() {
5      @Override
6      public void itemStateChanged() {
7          System.out.println("My option selected: " + option.isSelected());
8      }
9  });
10
11 frame.addPopupDetectionListener(new IPopupDetectionListener() {
12     @Override
13     public void popupDetected(final Position position) {
14         menu.show(position);
15     }
16 });
```

Popup Menüs können auch mit Hilfe von Menu Models erzeugt werden. Die Methode:

```
    void setPopupMenu(IMenuModel model);
```

erzeugt ein Popup Menü und bindet es an das übergebene Menu Model. Sobald ein Popup Event ausgelöst wurde, wird das zugehörige Popup Menü angezeigt. Das obige Beispiel sieht dann wie folgt aus:

```
1   final MenuModel menu = new MenuModel();
2
3   final ICheckedItemModel option = menu.addCheckedItem("My option");
4   option.addItemListener(new IItemStateListener() {
5       @Override
6       public void itemStateChanged() {
7           System.out.println("My option selected: " + option.isSelected());
8       }
9   });
10
11  frame.setPopupMenu(menu);
```

Die Verwendung von MenuModels hat mehrere Vorteile. Im obigen Beispiel spart man sich zum einen den folgenden *Boilerplate Code*:

```
1   frame.addPopupDetectionListener(new IPopupDetectionListener() {
2       @Override
3       public void popupDetected(final Position position) {
4           menu.show(position);
5       }
6   });
```

Zum Anderen kann ein MenuModel wiederverwendet werden. Im folgenden wird das selbe PopMenu auch im Hauptmenü einer Menübar hinzugefügt:

```
1   menu.setText("Main Menu");
2
3   final MenuBarModel menuBar = new MenuBarModel();
4   menuBar.addMenu(menu);
5
6   frame.setMenuBar(menuBar);
```

Weitere Information finden sich auch in den Abschnitten Menüs und Items sowie Menü und Item Models.

3.6.2.7 Farben

Mit Hilfe der folgenden Methoden kann die Vorder- und Hintergrundfarbe umgesetzt werden:

```
    void setForegroundColor(final IColorConstant colorValue);

    void setBackgroundColor(final IColorConstant colorValue);
```

Nicht alle Komponenten unterstützen diese Funktion für alle SPI Implementierungen und auf allen Betriebssystemen. So kann zum Beispiel unter Windows und SWT nicht die Hintergrundfarbe eines Buttons geändert werden. Durch das Setzen von `null` wird die Default Farbe verwendet.

Um die aktuell gesetzte Farbe zu erhalten, können folgende Methoden verwendet werden:

```
IColorConstant getForegroundColor();

IColorConstant getBackgroundColor();
```

Wurde zuvor keine spezielle Farbe gesetzt und ist somit die Default Farbe aktiv, wird 'null' zurückgegeben.

Zu allgemeinen Verwendung von Farben siehe auch den Abschnitt über Farben.

3.6.2.8 Mauszeiger

Mit Hilfe der folgenden Methode kann der Mauszeiger (Cursor) geändert werden:

```
void setCursor(final Cursor cursor);
```

3.6.2.9 Der `visible` Status

Der `visible` Status einer Komponente legt fest, ob eine Komponente *potentiell* sichtbar ist. Der Status kann mit den folgenden Methoden gesetzt und abgefragt werden:

```
void setVisible(final boolean visible);

boolean isVisible();
```

Die Verwendung kann für Fenster und Controls unterschiedlich sein. Ein Fenster wird durch `setVisible(true)` auf dem Bildschirm angezeigt und durch `setVisible(false)` versteckt.

Bei Controls hängt es vom verwendeten LayoutManager ab, wie damit verfahren wird. Mig Layout bietet zum Beispiel verschiedene *hidemodes* für den Umgang mit nicht sichtbaren Controls an.

Der Status `visible==true` sagt nicht aus, dass die Komponente tatsächlich auf dem Bildschirm *sichtbar* ist. So werden zum Beispiel Controls in nicht sichtbaren `TabItems` (siehe auch TabFolder) mit Status `visible==true` nicht angezeigt. Minimierte Fenster haben, auch wenn sie nicht auf dem Bildschirm sichtbar sind, den Status `visible==true`.

3.6.2.10 Der `showing` Status

Um Informationen über die Sichtbarkeit auf dem Bildschirm zu erlangen, kann der `showing` Status verwendet werden. Dieser wird mittels der folgenden Methode abgefragt:

```
boolean isShowing();
```

Der Status ist wie folgt definiert:

Eine *Root* Komponente hat den Status `showing==true` wenn `visible==true` ist. Eine Kind Komponente hat den Status `showing==true` wenn sie den Status `visible==true` **und** wenn ihr Vater den Status `showing==true` hat.

Unter der Annahme, das der verwendetet LayoutManager Controls mit dem Status `visible==false` nicht anzeigt, gilt dann folgendes:

- `showing==false` -> Die Komponente ist definitiv nicht am Bildschirm sichtbar

- `showing==true` -> Die Komponente ist eventuell am Bildschirm sichtbar. (Sie könnte jedoch durch andere Fenster verdeckt sein oder sich im nicht sichtbaren Bereich eines ScrollComposite befinden und somit dennoch nicht angezeigt werden.)

Der Status `showing==false` kann etwa ausgenutzt werden, um die Performance zu optimieren, indem für nicht dargestellte Controls *Berechnungen* ausgelassen werden.

Um sich über Änderungen des `showing` Status informieren zu lassen, kann ein `IShowingStateListener` verwendet werden:

```
void addShowingStateListener(IShowingStateListener listener);

void removeShowingStateListener(IShowingStateListener listener);
```

Dieser hat die folgende Methode:

```
void showingStateChanged(boolean isShowing);
```

3.6.2.11 Größe und Position

Mit folgenden Methoden kann die Größe eine Komponente gesetzt und ausgelesen werden:

```
void setSize(final Dimension size);

void setSize(int width, int height);

Dimension getSize();
```

Folgende Methoden dienen zum Setzen und Auslesen der Position:

```
void setPosition(final Position position);

void setPosition(int x, int y);

Position getPosition();
```

Bei Controls ist die Position relativ zum Usprung ihrer Vaterkomponente definiert.

Für Fenster handelt es sich bei der Position um Bildschirmkoordinaten. Dies gilt auch für Kindfenster wie zum Beispiel einen Dialog. Dessen Koordinaten sind **nicht** relativ zum Vaterfenster zu verstehen.

Der Ursprung eines Koordinatensystems zur Positionsangabe ist immer oben links. Y - Werte verlaufen also von *oben* nach *unten*.

Um die Position und Größe einer Komponente *in Einem* zu setzen, können die folgenden Methoden verwendet werden:

```
void setBounds(Rectangle bounds);

Rectangle getBounds();
```

Die Klassen `Dimension`, `Position` und `Rectangle` wurden *immutable* entworfen. Das bedeutet, dass ihre Koordinaten im Nachhinein nicht modifiziert werden können.

3.6.2.12 Koordinatenumrechung

Zur Umrechnung von Bildschirmkoordinaten in lokale Koordinaten (bezüglich der Vaterkomponente) oder zurück können die folgenden Methoden verwendet werden:

```
Position toLocal(final Position screenPosition);

Position toScreen(final Position localPosition);
```

Zur Umrechnung von Koordinaten in ein Koordinatensystem einer anderen Komponente oder zurück können die folgenden Methoden verwendet werden:

```
Position fromComponent(final IComponentCommon component, final Position componentPosition);

Position toComponent(final Position componentPosition, final IComponentCommon component);
```

3.6.3 Die Schnittstelle IContainer

Ein Container ist eine Komponente, welche eine Liste von Controls enthält. Die Schnittstelle `IContainer` erweitert `IComponent` und somit auch `IWidget`. Die Controls eines Containers werden mit Hilfe eines Layouters angeordnet. Es folgt eine Beschreibung der wichtigsten Methoden.

3.6.3.1 Hinzufügen von Controls mit Hilfe von BluePrints

Folgende Methoden können verwendet werden, um Controls mit Hilfe eines BluePrints zu einem Container hinzuzufügen:

```
<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    int index,
    IWidgetDescriptor<? extends WIDGET_TYPE> descriptor,
    Object layoutConstraints);

<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    int index,
    IWidgetDescriptor<? extends WIDGET_TYPE> descriptor);

<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    IWidgetDescriptor<? extends WIDGET_TYPE> descriptor,
    Object layoutConstraints);

<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    IWidgetDescriptor<? extends WIDGET_TYPE> descriptor);
```

Ein Control wird hinzugefügt, indem man ein BluePrint hinzufügt. Jedes BluePrint implementiert die Schnittstelle `IWidgetDescriptor` für einen konkreten `WIDGET_TYPE`. Die `add()` Methode liefert das erzeugte Control als Rückgabewert.

Der Index bestimmt, an welcher Stelle das Control in die interne Liste eingefügt werden soll. Wird kein Index angegeben, wird das Control am Ende hinzugefügt.

Die `layoutConstraints` werden vom verwendeten Layouter ausgewertet und sind somit *layout spezifisch*. Da ein Layouter nicht immer Constraints benötigt, können diese auch weggelassen werden. Zudem ist es möglich, die Constraints nachträglich auf dem Control zu setzen.

Es folgt ein Beispiel für die Verwendung der `add()` Methode:

```
1   final String growCC = "growx, w 0::";
2
3   final IInputField<Date> dateField = container.add(BPF.inputFieldDate(), growCC);
4
5   final IComboBoxSelectionBluePrint<Gender> genderCmbBp = BPF.comboBoxSelection(Gender.values());
6   final IComboBox<Gender> genderCmb = container.add(genderCmbBp, growCC);
7
8   final IButton okButton = container.add(BPF.buttonOk());
9
10  final ICheckBoxBluePrint licenceCbBp = BPF.checkBox().setText("Accept");
11  final ICheckBox licenceCb = container.add(2, licenceCbBp);
```

In Zeile 3-8 wird ein Datumsfeld, eine `ComboBox` für die Enum `Gender` sowie ein `OkButton` hinzugefügt. Das Datumsfeld sowie die Combobox haben ein MigLayout Constraint, dass sie den ganzen horizontalen Platz ausfüllen sollen (`growx`).

In Zeile 10-11 wird dann eine Checkbox an Position zwei (zwischen Combobox und Botton) eingefügt.

3.6.3.2 Hinzufügen von eigenen (Custom) Controls

Will man selbst Controls erstellen, hat man folgende Möglichkeiten:

- Man erstellt eine eigene Widget Bibliothek, siehe Erstellung eigener Widget Bibliotheken. Dabei wird auch ein BluePrint für das Control definiert welches man wie weiter oben beschrieben verwenden kann. Dies ist das empfohlene Vorgehen, wenn das Control wiederverwendet werden soll, da ein solch erstelltes Widget alle Vorzüge von jowidgets bietet.
- Man leitet von einem Basis Widget ab, um ein Widget zu kapseln.
- Man kapselt ein Widgets mit Hilfe eines `ICustomWidgetCreator`.

Folgende Methoden können verwendet werden, um Controls mit Hilfe eines `ICustomWidgetCreator` zu einem Container hinzuzufügen:

```
<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    int index,
    ICustomWidgetCreator<WIDGET_TYPE> creator,
    Object layoutConstraints);

<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    int index,
    ICustomWidgetCreator<WIDGET_TYPE> creator);

<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    ICustomWidgetCreator<WIDGET_TYPE> creator,
    Object layoutConstraints);

<WIDGET_TYPE extends IControl> WIDGET_TYPE add(
    ICustomWidgetCreator<WIDGET_TYPE> creator);
```

Die Verwendung des `index` und der `layoutConstraints` ist analog wie beider Verwendung von BluePrints.

Es folgt ein Beispiel für eine Implementierung der `ICustomWidgetCreator` Schnittstelle:

```
1  public final class ErrorLabel implements ICustomWidgetCreator<ILabel> {
2
3      private final String errorText;
4
5      public ErrorLabel(final String errorText) {
6          this.errorText = errorText;
7      }
8
9      @Override
10     public ILabel create(final ICustomWidgetFactory widgetFactory) {
11         final ILabelBluePrint labelBp = BPF.label();
12         labelBp.setText(errorText).setIcon(IconsSmall.ERROR);
13         final ILabel label = widgetFactory.create(labelBp);
14         return label;
15     }
16 }
```

In Zeile 13 wird die `ICustomWidgetFactory` verwendet, um ein `ILabel` Widget zu erzeugen.

Das `ErrorLabel` Control kann nun wie folgt einem Container hinzugefügt werden:

```
1      final ILabel label = container.add(new ErrorLabel("Ups!!!"));
```

Das folgende Beispiel Implementiert ein Control, welches den gesamten Platz mit schwarzer Farbe ausfüllt:

```
1  public final class BlackBox implements ICustomWidgetCreator<IControl> {
2
3      @Override
4      public IControl create(final ICustomWidgetFactory widgetFactory) {
5          final ICanvas canvas = widgetFactory.create(BPF.canvas());
6
7          canvas.addPaintListener(new IPaintListener() {
8              @Override
9              public void paint(final IPaintEvent paintEvent) {
10                 final IGraphicContext gc = paintEvent.getGraphicContext();
11                 final Rectangle bounds = gc.getBounds();
12                 gc.setBackgroundColor(Colors.BLACK);
13                 gc.clearRectangle(
14                     bounds.getX(),
15                     bounds.getY(),
16                     bounds.getWidth(),
17                     bounds.getHeight());
18             }
19         });
20
21         return canvas;
22     }
23 }
```

Hier wird in Zeile 5 die `ICustomWidgetFactory` verwendet, um ein Canvas zu erzeugen welches im `IPaintListener` ab Zeile 7 den gesammten Bereich mit schwarzer Farbe ausfüllt.

Das `BlackBox` Control kann dann die folgt einem Container hinzugefügt werden:

```
1      final IControl = frame.add(new BlackBox(), "growx, w 0::");
```

3.6.3.3 Entfernen von Controls

Controls können entweder auf dem Container mittels der folgenden Methoden entfernt werden:

```
boolean remove(IControl control);

void removeAll();
```

Oder man ruft auf dem Control selbst die `dispose()` Methode auf:

```
1 final IInputField<Date> dateField = container.add(BPF.inputFieldDate(), growCC);
2 dateField.dispose();
```

In Zeile 1 wird ein `DateField` zum Container hinzugefügt, in Zeile 2 wird es wieder entfernt.

Im folgenden Beispiel wird der gleiche Effekt erzielt, nur dass das Entfernen über den Container stattfindet.

```
1 final IInputField<Date> dateField = container.add(BPF.inputFieldDate(), growCC);
2 container.remove(dateField);
```

In beiden Fällen wird das `DateField` disposed und kann anschließend nicht mehr verwendet werden.

3.6.3.4 Verwendung von Layouts

Folgende Methoden können verwendet werden, um das Layout für den Container festzulegen.

```
void setLayout(ILayoutDescriptor layoutDescriptor);

<LAYOUT_TYPE extends ILayouter> LAYOUT_TYPE setLayout(ILayoutFactory<LAYOUT_TYPE> layoutFactory);
```

Das Layout kann entweder über ein `ILayoutDescriptor` oder über eine `ILayoutFactory` festgelegt werden. Der resultierende `ILayouter` ist dafür verantwortlich, die Controls eines Containers in der *richtigen* Größe an die *richtige* Position zu Zeichnen.

Mittels der folgenden Methoden kann der Layoutprozess *manuell* angestoßen werden:

```
void layoutBegin();

void layoutEnd();

void layout();

void layoutLater();
```

Die Methode `layout()` berechnet das Layout für den Container neu.

Die Methode `layoutBegin()` kann verwendet werden, um anzuzeigen, dass mehrere Änderungen des Containers folgen, welche das Layout beeinflussen. Dadurch wird verhindert, dass der Container neu gezeichnet wird, bis die Methode `layoutEnd()` aufgerufen wird. Diese kann verwendet werden, um *Flackereffekte* zu vermeiden.

Die Methode `layoutLater()` führt ein Layout in einem späteren UI Event durch. **Mehrere** Aufrufe der Methode `layoutLater()` im aktuellen Event bewirken dabei nur **einen** späteren Aufruf der Methode `layout()`.

Damit lässt sich ein in der Praxis gelegentlich auftretendes Problem lösen, welches bewirken kann, dass viele *unabhängige* Änderungen eines Containers in einem Ui Event ein sofortiges `layout()` nach

sich ziehen. Dann wird in einem Event mehrfach ein layout() durchgeführt, obwohl das Ergebnis nicht mehr als ein Mal pro UI Event sichtbar werden kann. De facto wird also nur der letzte `layout()` Aufruf für den Nutzer sichtbar, die vielen anderen `layout()` Aufrufe haben dabei unnötiger Weise den UI Thread blockiert. Durch den mehrfachen Aufruf der Methode `layoutLater()` anstatt `layout()` tritt das Problem dann nicht mehr auf, weil nur ein mal zu einem späteren Zeitpunkt ein `layout()` durchgeführt wird.

Um einen eigenen Layouter zu implementieren, sind die folgenden Methoden relevant:

```
Rectangle getClientArea();

Dimension computeDecoratedSize(Dimension clientAreaSize);
```

Die Methode `getClientArea()` gibt genau den Bereich zurück, welcher für das Zeichnen der Controls zur Verfügung steht.

Die Methode `computeDecoratedSize()` berechnet für eine gewünschte `clientAreaSize` die notwendige Größe des gesamten Containers.

Für weitere Details zu diesem Thema sei auf den Abschnitt Layouting verwiesen.

3.6.3.5 Tab Order

Um die Reihenfolge festzulegen, in welcher durch die Controls durch Verwendung der Tabulator Taste navigiert wird, können die folgenden Methoden verwendet werden:

```
void setTabOrder(Collection<? extends IControl> tabOrder);

void setTabOrder(IControl... controls);
```

Wird null übergeben, wird die *default* Reihenfolge verwendet

3.6.3.6 Zugriff auf die Controls eines Containers

Mit Hilfe der folgenden Methode erhält man alle derzeit vorhanden Kind Controls eines Containers:

```
List<IControl> getChildren();
```

Dabei handelt es sich um eine nicht modifizierbare (unmodifieable) Kopie der aktuellen Liste der Kind Controls. Dadurch ist zum Beispiel das folgende möglich, ohne eine `ConcurrentModificationException` zu bekommen:

```
1  container.layoutBegin();
2  for (final IControl childControl : container.getChildren()) {
3      if (!filter.accept(childControl)) {
4          container.remove(childControl);
5      }
6  }
7  container.layoutEnd();
```

Um sich darüber informieren zu lassen, wenn Kinder hinzugefügt oder entfernt werden, kann ein `IContainerListener` verwendet werden:

```
void addContainerListener(IContainerListener listener);

void removeContainerListener(IContainerListener listener);
```

Dieser hat die folgenden Methoden:

```
void afterAdded(IControl control);

void beforeRemove(IControl control);
```

Die Methode `afterAdded()` wird aufgerufen, nachdem ein Control zum Container hinzugefügt wurde. Die Methode `beforeRemove()` wird aufgerufen, bevor ein Control vom Container entfernt wird. Dabei spielt es keine Rolle, ob das Control mittels `dispose()` oder mittels `remove()` entfernt wird. Das Control ist zum Zeitpunkt des Aufrufs der Methode `beforeRemove()` **noch nicht** disposed.

3.6.3.7 Rekursiver Zugriff auf die Controls eines Containers

Ein Container kann Controls enthalten, welche selbst wiederum Container sein können. Dies ist zum Beispiel mit Hilfe eines Composite möglich, welches ein `IContainer` und ein `IControl` zugleich ist.

In der Praxis kann es vorkommen, dass zu einem eigenen Control Controls hinzugefügt werden, deren interner Aufbau einem über die Zeit (Controls können kommen und gehen) verborgen ist. Dennoch hätte man eventuell gerne Kenntnis über alle Kinder eines Containers (auch rekursiv), zum Beispiel um einen Listener hinzuzufügen, welcher ein PopupMenu öffnet.

Zu diesem Zweck kann eine `IContainerRegistry` verwendet werden, welche dem Container hinzugefügt wird:

```
void addContainerRegistry(IContainerRegistry registry);

void removeContainerRegistry(IContainerRegistry registry);
```

Die Schnittstelle `IContainerRegistry` hat die folgenden Methoden:

```
void register(IControl control);

void unregister(IControl control);
```

Die Methode `register()` wird für alle bereits vorhandenen (zum Zeitpunkt des Aufrufes von `addContainerRegistry()`) Controls sowie für alle in der Zukunft hinzugefügten Controls aufgerufen. Das ganze wird *rekursiv* durchgeführt, wodurch auch alle Kind Controls sowie deren Kinder usw. erfasst werden.

Werden Controls (oder Kind Controls, usw.) entfernt, wird die Methode `unregister()` aufgerufen. Kind Controls welche bereits vor dem Aufruf der Methode `addContainerRegistry()` entfernt wurden, werden dabei nicht berücksichtigt.

Zum rekursiven hinzufügen von Listenern zu einem Container und dessen Kindern stehen folgende Methoden zur Verfügung:

```
void addComponentListenerRecursive(IListenerFactory<IComponentListener> listenerFactory);

void removeComponentListenerRecursive(IListenerFactory<IComponentListener> listenerFactory);
```

```
    void addFocusListenerRecursive(IListenerFactory<IFocusListener> listenerFactory);

    void removeFocusListenerRecursive(IListenerFactory<IFocusListener> listenerFactory);

    void addKeyListenerRecursive(IListenerFactory<IKeyListener> listenerFactory);

    void removeKeyListenerRecursive(IListenerFactory<IKeyListener> listenerFactory);

    void addMouseListenerRecursive(IListenerFactory<IMouseListener> listenerFactory);

    void removeMouseListenerRecursive(IListenerFactory<IMouseListener> listenerFactory);

    void addPopupDetectionListenerRecursive(IListenerFactory<IPopupDetectionListener> listenerFactory);

    void removePopupDetectionListenerRecursive(IListenerFactory<IPopupDetectionListener> listenerFactory);
```

Die Schnittstelle `IListenerFactory` sieht wie folgt aus:

```
public interface IListenerFactory<LISTENER_TYPE> {

    LISTENER_TYPE create(IComponent component);
}
```

Durch das Hinzufügen eines rekursiven Listeners wird für den Container selbst und für jedes aktuelle und in der Zukunft hinzugefügte Control die `create()` Methode aufgerufen. Wird durch den Implementierer der `IListenerFactory` Schnittstelle ein Listener erzeugt, wird dieser der Component hinzugefügt, und vor dem dispose der Component wieder entfernt. Die `create()` Methode kann auch `null` zurückgeben, wodurch für diese Component kein Listener hinzugefügt wird.

Das folgende Beispiel soll dies verdeutlichen:

```
1   final IPopupMenu labelMenu = container.createPopupMenu();
2   labelMenu.setModel(labelMenuModel);
3
4   IListenerFactory<IPopupDetectionListener> listenerFactory;
5   listenerFactory = new IListenerFactory<IPopupDetectionListener>() {
6       @Override
7       public IPopupDetectionListener create(final IComponent component) {
8           if (component instanceof ILabel) {
9               final ILabel label = (ILabel) component;
10              return new IPopupDetectionListener() {
11                  @Override
12                  public void popupDetected(final Position position) {
13                      labelMenuModel.setLabel(label);
14                      labelMenu.show(label.toComponent(position, container));
15                  }
16              };
17          }
18          return null;
19      }
20  };
21
22  container.addPopupDetectionListenerRecursive(listenerFactory);
```

Für alle Labels eines Containers wird ein Kontextmenü (`labelMenu`) hinzugefügt. Bevor das Menü angezeigt wird, wird das Label auf dem zugehörigen Model (Zeile 13) gesetzt, um etwaige Aktionen bezüglich des ausgewählten Labels durchführen zu können.

Das Kontextmenü könnte zum Beispiel Aktionen enthalten, welche den Inhalt des Labels in die Zwischenablage kopieren oder dessen Farbe ändern, etc.

3.6.4 Die Schnittstelle IControl

Die Schnittstelle `IControl` stellt die gemeinsamen Funktionen für alle Controls bereits. `IControl` erweitert `IComponent` und somit auch `IWidget`.

Controls stellen Schnittstellen zur Nutzerinteraktion dar. Beispiele für elementare Controls sind Buttons, Checkboxen, Comboboxen, Eingabefelder, Slider etc. Controls können aber auch komplex sein, wie zum Beispiel ein InputComposite.

Controls werden immer zu Containern hinzugefügt. Es folgt eine kurze Übersicht der wichtigsten Methoden:

3.6.4.1 Tooltip

Mit der folgenden Methode kann der Tooltip Text gesetzt werden:

```
    void setToolTipText(String toolTip);
```

Wird `null` gesetzt, dann wird für das Control kein Tooltip angezeigt.

3.6.4.2 Layout Constraints

Mit den folgenden Methoden können die Layout Constraints für ein Control gesetzt und ausgelesen werden.

```
    void setLayoutConstraints(Object layoutConstraints);

    Object getLayoutConstraints();
```

Die Layout Constraints können `null` sein. In der Regel werden die Layout Constraints bereits beim Hinzufügen zum Container gesetzt:

```
1   final IInputField<Date> dateField = container.add(BPF.inputFieldDate(), "growx, w 0::");
```

In diesem Fall würde die Methode `dateField.getLayoutConstraints()` den String `"growx, w 0::"` zurückgeben.

Die Layout Constraints sind *Layouter spezifisch*, das bedeutet, ein Layouter definiert, welche Constraints gültig sind.

Werden die Layout Constraints nach dem *layouten* des zugehörigen Containers geändert, so muss dieser neu *gelayoutet* werden.

Weitere Informationen finden sich im Abschnitt Layouting

3.6.4.3 Default Größen

Mittels der folgenden Methoden können die Default Größen für ein Control ermittelt werden.

```
    Dimension getMinSize();

    Dimension getPreferredSize();

    Dimension getMaxSize();
```

Die Methoden dürfen nicht `null` zurückgeben. Werden eigene Controls entworfen, sollte darauf geachtet werden, *sinnvolle* Default Größen zu liefern.

Die `MinSize` definiert die minimale Größe, die ein Control haben soll, damit es sinnvoll angezeigt werden kann. Die `PreferredSize` gibt an, wie groß ein Control sein soll, um es optimal anzuzeigen. Die `MaxSize` definiert, wie groß ein Control maximal angezeigt werden soll. Die Default Größen können sich zur Laufzeit ändern. So ändert sich Beispielsweise die `PreferredSize` eines Textfeldes abhängig vom gesetzten Text.

Einige Layouter wie zum Beispiel Mib Layout bieten die Möglichkeit, die Default Größen mit Hilfe von Constraints zu überschreiben. Die MigLayout Constraints `"w 0:100:200"` setzen zum Beispiel die `MinWidth` auf `0`, die `PreferredWidth` auf `100` und die `MaxWidth` auf `200`, unabhängig davon, was die Methoden `getMinSize()`, `getPreferredSize()` und `getMaxSize()` für Werte zurückliefern. Andere Layouter wie zum Beispiel das FillLayout ignorieren die Default Größen vollständig.

Mit Hilfe der folgenden Methoden können die Default Größen eines Controls geändert werden:

```
void setMinSize(final Dimension minSize);

void setPreferredSize(Dimension preferredSize);

void setMaxSize(Dimension maxSize);
```

Nach dem Umsetzen einer Default Größe geben die zugehörigen Getter Methoden den neuen Wert zurück. Wird eine Default Größe auf `null` gesetzt, wird vom zugehörigen Getter wieder der **Default Wert** und nicht `null` zurückgegeben.

3.6.4.4 Drag and Drop

Mit den folgenden Methoden kann einem Control *Drag and Drop* Funktionalität hinzugefügt werden:

```
IDragSource getDragSource();

IDropTarget getDropTarget();
```

Für weitere Informationen sei auf den Abschnitt Drag and Drop verwiesen.

3.6.5 Die Schnittstelle IDisplay

Die Schnittstelle `IDisplay` leitet von `IWidget` ab und ist die Basisschnittstelle für alle eigenständig anzeigbaren Widgets. Dazu zählen alle Arten von Fenstern und Dialogen. Die Schnittstelle `IDisplay` hat selbst keine zusätzlichen Methoden, um beim Entwurf von Dialog oder Fensterschnittstellen eine größtmögliche Freiheit zu haben, und nicht Methoden zu erben, deren Implementierung keinen Sinn macht und dadurch zu einer `UnsupportedOpperationException` führen würde.

Einige Dialoge leiten daher bewusst nicht von IWindow ab, weil zum Beispiel das Erzeugen von Kind Fenstern nicht möglich sein soll, oder der Aufruf der Methode `pack()` keinen Effekt hätte. Dazu zählen zum Beispiel der File Chooser, der Directory Chooser, der Message Dialog, der Question Dialog und weitere.

3.6.6 Die Schnittstelle IWindow

Die Schnittstelle `IWindow` stellt gemeinsamen Funktionen für Fenster bereits. `IWindow` erweitert `IDisplay` und somit auch `IWidget`. Es folgt eine kurze Übersicht der wichtigsten Methoden:

3.6.6.1 Kind Fenster

Ein Kind Fenster lässt sich mit Hilfe der folgenden Methode erzeugen:

```
<WIDGET_TYPE extends IDisplay, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
    WIDGET_TYPE createChildWindow(final DESCRIPTOR_TYPE descriptor);
```

Ein Kind Fenster wird mit Hilfe eines BluePrint (IWidgetDescriptor) erzeugt. Dieses wird der Methode `createChildWindow()` übergeben, welche das erzeugte Fenster zurück gibt.

Im folgenden Beispiel wird so ein FileChooser zum Öffnen einer Datei erstellt:

```
1  IFileChooserBluePrint fileChooserBp = BPF.fileChooser(FileChooserType.OPEN_FILE);
2  final IFileChooser fileChooser = frame.createChildWindow(fileChooserBp);
3  final DialogResult result = fileChooser.open();
4  if (DialogResult.OK.equals(result)) {
5      System.out.println(fileChooser.getSelectedFiles());
6  }
```

Die folgende Methode liefert die Kind Fenster eines Fensters.

```
List<IDisplay> getChildWindows();
```

Dabei handelt es sich um eine nicht modifizierbare Kopie aller aktuell vorhandenen Kind Fenster. Fenster die bereits disposed wurden, sind nicht enthalten.

Im obigen Beispiel würde ein Aufruf von `getChildWindows()` zwischen Zeile 2 und Zeile 3 eine Referenz auf den erzeugten File Chooser enthalten, und bei einem Aufruf nach Zeile 3 nicht mehr, da der Aufruf `open()` blockiert, bis der File Chooser wieder geschlossen wird, und anschließend das FileChooser Fenster automatisch disposed wurde.

3.6.6.2 Window Listener

Die folgenden Methoden können verwendet werden um `IWindowListener` zu registrieren oder zu deregistrieren:

```
void addWindowListener(IWindowListener listener);

void removeWindowListener(IWindowListener listener);
```

Ein `IWindowListener` hat die folgenden Methoden:

```
void windowActivated();

void windowDeactivated();

void windowIconified();

void windowDeiconified();

void windowClosing(IVetoable vetoable);

void windowClosed();
```

In einer Fenster basierten Anwendung kann zur selben Zeit genau ein Fenster aktiv sein. Die Methoden `windowActivated()` und `windowDeactivated()` informieren über eine Änderung des `active` Status. Die Methode Toolkit.getActiveWindow() liefert das gerade aktive Fenster.

Die Methode `windowClosing(IVetoable vetoable)` wird aufgerufen, bevor ein Fenster geschlossen werden soll. Mit Hilfe des `vetoable` hat man die Möglichkeit, das Schließen zu verhindern. Das folgende Beispiel soll dies verdeutlichen:

```
1   frame.addWindowListener(new WindowAdapter() {
2       @Override
3       public void windowClosing(final IVetoable vetoable) {
4           final String msg = "Close Window?";
5           final QuestionResult questionResult = QuestionPane.askYesNoQuestion(msg, msg);
6           if (!QuestionResult.YES.equals(questionResult)) {
7               vetoable.veto();
8           }
9       }
10  });
```

Beim Schließen des Fensters wird ein QuestionDialog angezeigt, um den Nutzer zu fragen, ob das Fenster wirklich geschlossen werden soll. Falls nicht wird in Zeile 7 ein *Veto eingelegt*, wodurch das Fenster nicht geschlossen wird. Die folgende Abbildung zeigt das Ergebnis nachdem das `X` zum Schließen gedrückt wurde:

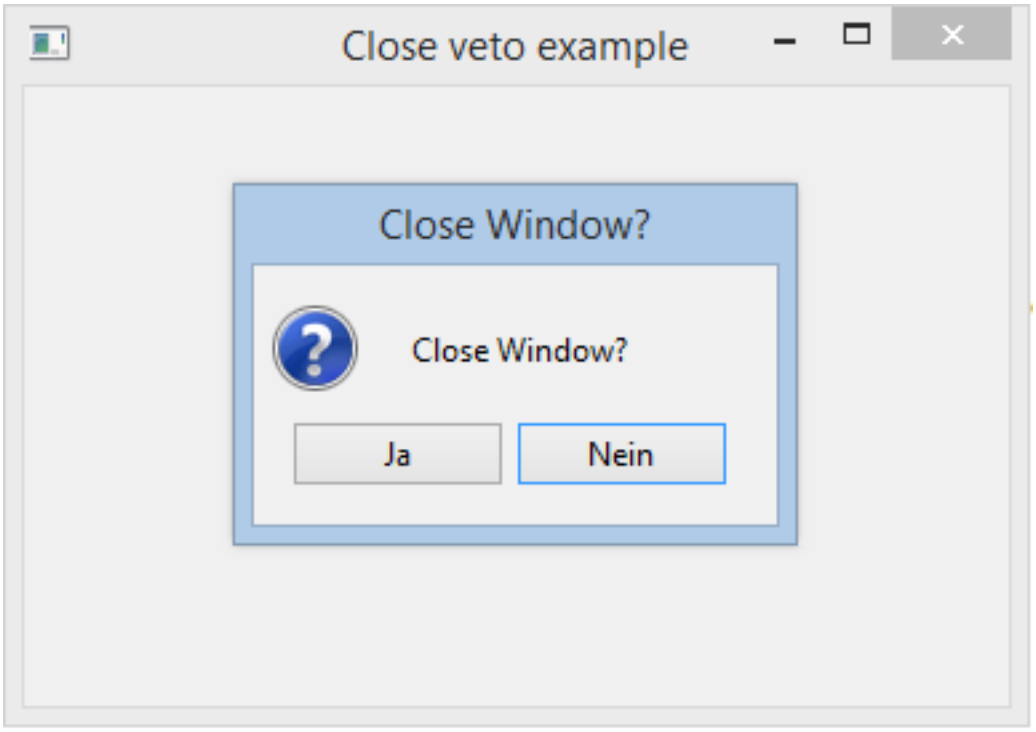


Abbildung 3.1: Close Window Veto Beispiel

Die Methode `windowClosed()` wird aufgerufen, nachdem das Fenster geschlossen wurden. **Achtung:** Wenn ein Fenster geschlossen, bedeutet dies nicht automatisch, das das Fenster dann auch disposed ist.

Die Klasse `org.jowidgets.tools.controller.WindowAdapter` implementiert die `IWindowListener` Schnittstelle mit leeren Methodenrümpfen und kann verwendet werden, falls nicht alle Methoden

implementiert werden sollen.

3.6.6.3 Packen von Fenstern

Die Methode pack

```
void pack();
```

sorgt dafür, das für das Fenster die `PreferredSize` berechnet und gesetzt wird. Anschließend wird auf dem zugehörigen Container ein `layout()` durchgeführt.

Mit Hilfe der folgenden Methoden kann das `pack()` beeinflußt werden:

```
void setMinPackSize(Dimension size);

void setMaxPackSize(Dimension size);
```

Die `MinPackSize` legt eine minimale Größe fest, die das Fenster mindestens (trotz pack) haben soll. Dadurch kann zum Beispiel verhindert werden, dass ein Fenster mit *dynamisch* erzeugten Inhalt zu klein angezeigt wird.

Die `MaxPackSize` legt eine maximale Größe fest, die das Fenster höchsten (trotz pack) haben soll. Dadurch kann zum Beispiel verhindert werden, dass ein Fenster mit *dynamisch* erzeugten Inhalt zu groß angezeigt wird.

3.6.6.4 Weitere Utilities

Die folgende Methode:

```
Rectangle getParentBounds();
```

liefert die `ParentBounds`. Ist das Fenster ein Root Fenster (`getParent() == null`) werden die Bounds des Bildschirm zurückgegeben. Ist es ein Kind Fenster, werden die Bounds des Vaters zurückgegeben. Die Methode liefert somit nie `null` zurück. Dies kann zum Beispiel hilfreich sein, um ein Fenster relativ zum Vater anzuzeigen, ohne vorab überprüfen zum müssen, ob es sich um ein Root Fenster oder ein Kind Fenster handelt.

Die folgende Methode:

```
void centerLocation();
```

setzt die Position des Fensters so, das es zentriert zu den `ParentBounds` angezeigt wird. Root Fenster werden somit zentriert auf dem Bildschirm angezeigt, Kind Fenster zentriert bezüglich des Vater Fensters. Dabei haben beide Fenster dann den gleichen Mittelpunkt.

3.6.7 Die Schnittstelle IInputComponent

InputComponents liefern eine Nutzereingabe für einen definierten Datentyp und stellen diesen Wert dar. Dieser kann sowohl einfach sein (z.B. String, Integer, Date, ...) als auch komplex (z.B. Person, Company, Rule, ...). Die Schnittstelle `IInputComponent<VALUE_TYPE>` erweitert `IComponent` und somit auch

IWidget. Zudem ist ein InputComponent von IValidatable abgeleitet und kann somit seinen aktuellen Zustand selbst validieren. Konkrete Implementierungen sind zum Beispiel InputField, InputComposite, InputDialog, ComboBox, CheckBox, Slider, Slider Viewer etc..

Es folgt eine kurze Beschreibung der wichtigsten Methoden:

3.6.7.1 Value Access

Mittels der folgenden Methoden kann der Wert gesetzt und ausgelesen werden:

```
void setValue(VALUE_TYPE value);

VALUE_TYPE getValue();
```

Der Wert `null` ist beim Setzen erlaubt und daher auch beim `getValue()` möglich.

3.6.7.2 InputListener

Um sich über Änderungen des Wertes informieren zu lassen, kann ein `IInputListener` verwendet werden:

```
void addInputListener(IInputListener listener);

void removeInputListener(IInputListener listener);
```

Dieser hat die folgende Methode:

```
void inputChanged();
```

3.6.7.3 Validierung

Ein InputComponent ist von IValidatable abgeleitet. Es kann also ein `IValidationConditionListener` registriert werden, der aufgerufen wird, wenn sich die Validierungsbedingungen geändert haben.

Mit Hilfe der folgenden Methode kann ein externer Validierer hinzugefügt werden.

```
void addValidator(IValidator<VALUE_TYPE> validator);
```

Er wird immer ausgewertet, wenn sich die Validierungsbedingungen geändert haben. Sein Ergebnis fließt in die Implementierung der Methode `validate()` von IValidatable ein. Ein `IInputComponent` kann zusätzliche interne Validierer besitzen.

3.6.7.4 InputChanged Events vs. ValidationConditionChanged Events

Wenn der Nutzer die Eingabe eines `IInputComponent` ändert, kann sowohl ein `inputChanged()` auf den registrierten `IInputListener` Objekten als auch ein `validationConditionChanged()` auf den registrierten `IValidationConditionListener` Objekten aufgerufen werden.

Dabei wird **immer** zuerst das `inputChanged()` aufgerufen, und anschließend das `validationConditionChanged()`. Da InputComponents aus anderen InputComponents bestehen

können, kann man dadurch sicherstellen, das erst das Binding mit Hilfe von `inputChanged()` durchgeführt wird, und anschließend die Validierung. **Implementierer der Schnittstelle `IInputComponent` müssen darauf achten, dies einzuhalten.**

Es kann vorkommen, dass `validationConditionChanged()` aufgerufen wird, ohne das ein Aufruf von `inputChanged()` vorausgeht. Dies soll an einem Beispiel verdeutlicht werden. Ein Eingabefeld zur Eingabe eines Datums verwendet ein Textfeld zur Eingabe von Zeichenketten (String) für die Implementierung, welche intern in ein Date Objekt konvertiert werden. Ein interner Validierer verwendet den Eingabetext für Validierungen. Obwohl sich der Text geändert hat, und dadurch ein `validationConditionChanged()` ausgelöst wird, kann es sein, das `getValue()` vorher und nachher `null` ist, weil die Zeichenkette sich weder vor noch nach der Änderung in ein Date Objekt konvertieren lässt. Es wäre aber möglich, dass der Validierer für die Zeichenkette unterschiedliche Ergebnisse hat, da es vor der Änderung durch das Hinzufügen von Zeichen noch richtig werden könnte (INFO_ERROR), danach aber nicht mehr (ERROR).

3.6.7.5 Modfication State

Zum Auslesen und zurücksetzten des `modifcationState` können die folgenden Methoden verwendet werden:

```
boolean hasModifications();

void resetModificationState();
```

Der ModificationState gibt an, ob es seit der Erzeugung oder seit dem letzten Aufruf von `resetModificationState()` Änderungen an der Komponente gab. Die Methode `resetModifciationState()` macht **nicht** die eigentliche Modifikation rückgängig, sondern bewirkt, dass der `modificationState` auf `false` zurückgesetzt wird.

Der `modificationState` bezieht sich im Allgemeinen **nicht** auf den per `getValue()` verfügbaren Wert, sondern darauf, ob der Nutzer irgend eine Änderung vorgenommen hat. Siehe dazu auch InputChanged Events vs. ValidationConditionChanged Events. Der `modificationState` ist zum Beispiel relevant, wenn der Nutzer unterschiedliche Validation Messages angezeigt bekommen soll, abhängig ob etwas durch Ihn geändert wurde oder nicht.

3.6.7.6 Editierbarkeit

Mit Hilfe der folgenden Methoden kann die Editierbarkeit gesetzt und abgefragt werden:

```
void setEditable(boolean editable);

boolean isEditable();
```

3.6.8 Die Schnittstelle IItem

Items sind zum Einen die Elemente von Menüs oder Toolbars. Weitere Items sind TabItems, also die *Reiter* eines TabFolder sowie die Nodes eines Tree. Items sind somit keine eigenständigen Komponenten sondern Teil anderer Komponenten oder Menüs. Eine TreeNode kann nicht ohne zugehörigen Tree existieren, ein TabItem nicht ohne zugehörigen TabFolder und ein MenuItem nicht ohne zugehöriges Menü.

Items können einen Text (Label[2]), ein Tooltip und ein Icon haben. Zum Setzen und Auslesen dieser Eigenschaften existieren die folgenden Methoden:

```
    void setText(String text);

    String getText();

    void setToolTipText(String text);

    String getToolTipText();

    IImageConstant getIcon();

    void setIcon(IImageConstant icon);
```

Die Verwendung von Icons wird im Abschnitt Icons und Images beschrieben.

3.7 Widget Wrapper

Für die meisten Widget Schnittstellen existieren Wrapper Klassen, welche es ermöglichen, von einem Widget abzuleiten und eigene Funktionen hinzuzufügen. Die Wrapper befinden sich im Packet `org.jowidgets.tools.widgets.wrapper`. Die folgenden Wrapper sind derzeit vorhanden:

- ButtonWrapper
- CheckBoxWrapper
- ComboBoxWrapper
- ComponentWrapper
- CompositeWrapper
- ContainerWrapper
- ControlWrapper
- DisplayWrapper
- FrameWrapper
- InputControlWrapper
- InputDialogWrapper
- TabFolderWrapper
- TableWrapper
- TreeWrapper
- WidgetWrapper
- WindowWrapper

Die Implementierung eines Wrappers ist immer nach dem gleichen Schema aufgebaut. Das folgende Beispiel zeigt den `ComboBoxWrapper`:

```
1 package org.jowidgets.tools.widgets.wrapper;
2
3 import java.util.Collection;
4 import java.util.List;
5
```

[2] Die Eigenschaft `text` sollte passender `label` heißen. Angelehnt an das SWT *wording* wurde jedoch anfangs `text` gewählt. Dies nachträglich zu ändern würde einen nicht unerheblichen Aufwand mit sich bringen und müsste dann konsequenterweise auch an anderen Stellen (z.B. beim Button) umgesetzt werden.

```
import org.jowidgets.api.widgets.IComboBox;
import org.jowidgets.util.IObservableValue;

public class ComboBoxWrapper<VALUE_TYPE>
    extends InputControlWrapper<VALUE_TYPE>
    implements IComboBox<VALUE_TYPE> {

    public ComboBoxWrapper(final IComboBox<VALUE_TYPE> widget) {
        super(widget);
    }

    @Override
    protected IComboBox<VALUE_TYPE> getWidget() {
        return (IComboBox<VALUE_TYPE>) super.getWidget();
    }

    @Override
    public IObservableValue<VALUE_TYPE> getObservableValue() {
        return getWidget().getObservableValue();
    }

    @Override
    public List<VALUE_TYPE> getElements() {
        return getWidget().getElements();
    }

    @Override
    public void setElements(final Collection<? extends VALUE_TYPE> elements) {
        getWidget().setElements(elements);
    }

    // ... removed some methods in example

    @Override
    public boolean isPopupVisible() {
        return getWidget().isPopupVisible();
    }

}
```

Falls für eine Widget Schnittstelle noch kein Wrapper existiert, kann ein solcher also einfach selbst erstellt werden[3].

3.7.1 Widget Wrapper in Kombination mit der IWidgetFactory

Widget Wrapper werden unter anderem bei der Erstellung eigener Widget Bibliotheken benötigt.

Das folgende Beispiel zeigt die Widget Factory und Implementierung des Label Widget. Ein Label Widget besteht aus einen Icon und einem Text Label.

```
public final class LabelFactory implements IWidgetFactory<ILabel, ILabelDescriptor> {

    @Override
    public ILabel create(final Object parentUiReference, final ILabelDescriptor descriptor) {
        final ICompositeBluePrint compositeBp = BPF.composite();
        compositeBp.setSetup(descriptor);

        final IComposite composite = Toolkit.getWidgetFactory().create(
            parentUiReference,
```

[3] Patches mit solchen Erweiterungen werden dankbar angenommen und integriert.

```
10              compositeBp);
11
12          return new LabelImpl(composite, descriptor);
13      }
14
15  }
```

Für die Implementierung soll ein Composite verwendet werden, welches das Icon und das Text Label enthält. In Zeile 5 wird ein Composite BluePrint erstellt. In Zeile 6 werden alle passenden Parameter des Label Descriptor auch auf dem Composite gesetzt. In Zeile 8 wird das Composite erzeugt und in Zeile 12 wir dieses an die Klasse `LabelImpl` übergeben.

Die Klasse `LabelImpl` sieht wie folgt aus:

```
1   public final class LabelImpl extends ControlWrapper implements ILabel {
2
3       private final IIcon iconWidget;
4       private final ITextLabel textLabelWidget;
5       private final IComposite composite;
6
7       private String text;
8       private IImageConstant icon;
9
10      public LabelImpl(final IComposite composite, final ILabelSetup setup) {
11          super(composite);
12
13          this.composite = composite;
14
15          final IIconDescriptor iconDescriptor = BPF.icon(setup.getIcon()).setSetup(setup);
16          this.iconWidget = composite.add(iconDescriptor, "w 0::");
17          this.icon = setup.getIcon();
18
19          final ITextLabelDescriptor textLabelDescriptor = BPF.textLabel().setSetup(setup);
20          this.textLabelWidget = composite.add(textLabelDescriptor, "w 0::");
21
22          setLayout();
23
24          VisibiliySettingsInvoker.setVisibility(setup, this);
25          ColorSettingsInvoker.setColors(setup, this);
26      }
27
28      @Override
29      public void setToolTipText(final String text) {
30          textLabelWidget.setToolTipText(text);
31          iconWidget.setToolTipText(text);
32      }
33
34      @Override
35      public void setForegroundColor(final IColorConstant colorValue) {
36          textLabelWidget.setForegroundColor(colorValue);
37      }
38
39      @Override
40      public void setBackgroundColor(final IColorConstant colorValue) {
41          textLabelWidget.setBackgroundColor(colorValue);
42          iconWidget.setBackgroundColor(colorValue);
43      }
44
45      //... removed some methods in example
46
47  }
```

Es wird von der Klasse `ControlWrapper` abgeleitet. Dadurch werden alle Methoden von `IControl`, die

nicht von `LabelImpl` überschrieben werden, an das Composite delegiert.

Hinweis: Es wurde bewusst nicht von `CompositeWrapper` abgeleitet, da die Schnittstelle `ILabel` auch nicht von `IComposite` abgeleitet ist. Bei der Verwendung von Swing oder Swt findet man häufig eigene, firmen- oder projektinterne Widgets, die zum Beispiel von JPanel oder Composite angeleitet sind. Diese haben dadurch in Ihrer öffentlichen Schnittstelle Methoden wie `remove()`, `add()` oder `setLayout()` welche vom Benutzer des Widgets nicht verwendet werden sollten (zum Beispiel hat ein Label Widget keine `setLayout()` oder `remove()` Methode). Bei der Erstellung eigener Widget Bibliotheken kann man explizit die Schnittstelle für das Widget (z.B. ILabel) festlegen. Dadurch wäre auch das Ableiten von CompositeWrapper nicht tragisch, da die Widget Factory nur ein ILabel zurückgibt. Wird keine eigene Widget Schnittstelle festgelegt ist es jedoch umso wichtiger, keinen Wrapper zu verwenden, durch den man Methoden erbt, die für das Widget nicht sinvoll sind.

3.8 Base Widgets

Die Base Widgets können zur Kapselung eigener Widgets verwendet werden, ohne dass man dazu eine eigene Widget Bibliotheken erstellen muss. Sie stellen somit die *klassische* Variante zur Erstellung eigener Widgets dar. Um alle Vorteile von jowidgets zu nutzen wird jedoch empfohlen, Widget in einer eigenen Widget Bibliotheken zu kapseln.

Die Base Widgets sind von den Widget Wrappern abgeleitet. Es ist auch einfach möglich, eigene Base Widgets zu erstellen. Die Base Widgets befinden sich im Packet `org.jowidgets.tools.widgets.base`.

Derzeit existieren die folgenden Base Widgets:

- Frame
- Dialog
- Composite Control

3.8.1 Frame

Die Klasse `Frame` ist wie folgt implementiert:

```
1  public class Frame extends FrameWrapper implements IFrame {
2
3      public Frame(final String title, final IImageConstant icon) {
4          this(BPF.frame(title).setIcon(icon));
5      }
6
7      public Frame(final String title) {
8          this(BPF.frame(title));
9      }
10
11     public Frame(final IFrameDescriptor descriptor) {
12         super(Toolkit.createRootFrame(descriptor));
13     }
14
15     protected IFrame getFrame() {
16         return (IFrame) super.getWidget();
17     }
18
19 }
```

Das BaseFrameSnipped demonstriert die Verwendung der Klasse `Frame`:

```
public final class BaseFrameSnipped implements IApplication {

    @Override
    public void start(final IApplicationLifecycle lifecycle) {

        final IFrame frame = new MyFrame();

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosed() {
                lifecycle.finish();
            }
        });

        frame.setVisible(true);
    }

    private final class MyFrame extends Frame {

        public MyFrame() {
            super("Base frame Snipped");

            setMinPackSize(new Dimension(300, 0));

            setLayout(new MigLayoutDescriptor("wrap", "[][grow]", "[][]"));

            add(BPF.textLabel("Label 1"));
            add(BPF.inputFieldString(), "growx");

            add(BPF.textLabel("Label 2"));
            add(BPF.inputFieldString(), "growx");
        }

    }
}
```

3.8.2 Dialog

Die Klasse `Dialog` ist wie folgt implementiert:

```
public class Dialog extends FrameWrapper implements IFrame {

    public Dialog(final IWindow parent, final String title, final IImageConstant icon) {
        this(parent, BPF.dialog(title).setIcon(icon));
    }

    public Dialog(final IWindow parent, final String title) {
        this(parent, BPF.dialog(title));
    }

    public Dialog(final IWindow parent, final IWidgetDescriptor<? extends IFrame> descriptor) {
        super(Toolkit.getWidgetFactory().create(
            Assert.getParamNotNull(parent, "parent").getUiReference(),
            descriptor));
    }

    protected IFrame getFrame() {
        return (IFrame) super.getWidget();
    }
}
```

Das BaseDialogSnipped demonstriert die Verwendung der Klasse `Dialog`:

```java
public final class BaseDialogSnipped implements IApplication {

    @Override
    public void start(final IApplicationLifecycle lifecycle) {

        //create the root frame
        final IFrameBluePrint frameBp = BPF.frame().setTitle("Base dialog Snipped");
        frameBp.setSize(new Dimension(300, 200));
        final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);
        frame.setLayout(new MigLayoutDescriptor("[]", "[]"));

        //create a dialog
        final IFrame dialog = new MyDialog(frame);

        //create a button that opens the dialog
        final IButton button = frame.add(BPF.button("Open dialog"));
        button.addActionListener(new IActionListener() {
            @Override
            public void actionPerformed() {
                dialog.setVisible(true);
            }
        });

        //set the frame visible
        frame.setVisible(true);
    }

    private final class MyDialog extends Dialog {

        public MyDialog(final IWindow parent) {
            super(parent, "My Dialog");

            setMinPackSize(new Dimension(300, 0));

            setLayout(new MigLayoutDescriptor("wrap", "[][grow]", "[][]"));

            add(BPF.textLabel("Label 1"));
            add(BPF.inputFieldString(), "growx");

            add(BPF.textLabel("Label 2"));
            add(BPF.inputFieldString(), "growx");
        }

    }
}
```

3.8.3 Composite Control

Das Composite Control implementiert die Schnittstelle `IControl` und bietet ein `IComposite` als Content Pane zum Hinzufügen von Controls.

Die Klasse `CompositeControl` ist wie folgt implementiert:

```java
public class CompositeControl extends ControlWrapper implements IControl {

    public CompositeControl(final IContainer parent) {
        this(parent, BPF.composite(), null);
    }

    public CompositeControl(
        final IContainer parent,
```

```
9           final int index) {
10
11          this(parent, index, BPF.composite(), null);
12      }
13
14      public CompositeControl(
15          final IContainer parent,
16          final ICompositeDescriptor descriptor) {
17
18          this(parent, descriptor, null);
19      }
20
21      public CompositeControl(
22          final IContainer parent,
23          final int index,
24          final ICompositeDescriptor descriptor) {
25
26          this(parent, index, descriptor, null);
27      }
28
29      public CompositeControl(
30          final IContainer parent,
31          final Object layoutConstraints) {
32
33          this(parent, BPF.composite(), layoutConstraints);
34      }
35
36      public CompositeControl(
37          final IContainer parent,
38          final int index,
39          final Object layoutConstraints) {
40
41          this(parent, index, BPF.composite(), layoutConstraints);
42      }
43
44      public CompositeControl(
45          final IContainer parent,
46          final ICompositeDescriptor descriptor,
47          final Object layoutConstraints) {
48
49          super(Assert.getParamNotNull(parent, "parent").add(
50              descriptor,
51              layoutConstraints));
52      }
53
54      public CompositeControl(
55          final IContainer parent,
56          final int index,
57          final ICompositeDescriptor descriptor,
58          final Object layoutConstraints) {
59
60          super(Assert.getParamNotNull(parent, "parent").add(
61                  index,
62                  descriptor,
63                  layoutConstraints));
64      }
65
66      protected IComposite getComposite() {
67          return (IComposite) super.getWidget();
68      }
69
70  }
```

Das BaseCompositeControlSnipped demonstriert die Verwendung der Klasse `Dialog`:

```java
public final class BaseCompositeControlSnipped implements IApplication {

    @Override
    public void start(final IApplicationLifecycle lifecycle) {

        final IFrame frame = new MyFrame();

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosed() {
                lifecycle.finish();
            }
        });

        frame.setVisible(true);
    }

    private final class MyFrame extends Frame {

        public MyFrame() {
            super("Base composite control snipped");

            setMinPackSize(new Dimension(300, 0));

            setLayout(FillLayout.get());

            final MyControl control = new MyControl(this, "growx, growy");
            control.setText1("Hello");
            control.setText2("World");
        }

    }

    private final class MyControl extends CompositeControl {

        private final IInputField<String> field1;
        private final IInputField<String> field2;

        public MyControl(final IContainer parent, final Object layoutConstraints) {
            super(parent, layoutConstraints);

            final IComposite composite = getComposite();

            composite.setLayout(new MigLayoutDescriptor("wrap", "[][grow]", "[][]"));

            composite.add(BPF.textLabel("Label 1"));
            field1 = composite.add(BPF.inputFieldString(), "growx");

            composite.add(BPF.textLabel("Label 2"));
            field2 = composite.add(BPF.inputFieldString(), "growx");
        }

        public void setText1(final String text) {
            field1.setValue(text);
        }

        public void setText2(final String text) {
            field2.setValue(text);
        }

    }
}
```

3.9 Layouting

Um die Controls eines Containers anzuordnen benötigt man einen Layouter. Man kann entweder vorgefertigte Layouter verwenden, oder selbst einen Custom Layouter implementieren.

Anfangs unterstütze jowidgets ausschließlich Mig Layout als Layout Mechanismus. Dieses musste von einer SPI Implementierung unterstützt werden [4]. Für die Definition des Layouts wird dabei die Klasse `MigLayoutDescriptor` verwendet, welche das Tagging Interface `ILayoutDescriptor` implementiert.

Später wurde im Rahmen einer Bacheloarbeit jowidgets um die Möglichkeit erweitert, eigene Layouter zu erstellen. Dabei wurde die Schnittstelle `ILayouter` eingeführt, welche ebenfalls von `ILayoutDescriptor` abgeleitet ist. Zudem wurde Mig Layout für diese Schnittstelle portiert, so dass eine SPI Implementierung nicht mehr zwingend eine Mig Layout Implementierung anbieten muss. Außerdem wurden weitere vorgefertigte Layouter hinzugefügt.

Zur besseren Unterscheidung wird das portierte Mig Layout im folgenden als **Mib Layout**[5] bezeichnet. Im Gegensatz dazu wird Mig Layout als **Natives Mig Layout** bezeichnet, wenn betont werden soll, dass es sich **nicht** um Mib Layout handelt.

Um ein Layout auf einem Container zu setzen, bietet dieser die folgenden Methoden an:

```
    void setLayout(ILayoutDescriptor layoutDescriptor);

    <LAYOUT_TYPE extends ILayouter> LAYOUT_TYPE setLayout(
        ILayoutFactory<LAYOUT_TYPE> layoutFactory);
```

Mit Hilfe der ersten Methode wird entweder ein `ILayouter` oder der native `MigLayoutDescriptor` gesetzt.

Die zweite Methode bietet die Möglichkeit, einen Layouter mit Hilfe einer `ILayoutFactory` zu setzen. Der Layouter wird dabei erzeugt und zurückgegeben. Die Layout Factory sieht wie folgt aus:

```
1 public interface ILayoutFactory<LAYOUTER_TYPE extends ILayouter> {
2
3     LAYOUTER_TYPE create(IContainer container);
4
5 }
```

Layout Factories lassen sich für verschiedene Container wiederverwenden. Die vordefinierten Layouts von Jowidgets (mit Ausnahme des nativen Mig Layout) bieten eine Implementierung der `ILayoutFactory` Schnittstelle.

3.9.1 Mig Layout (nativ)

Mig Layout ist ein freier, von Mikael Grev, Inhaber der Firma MiG InfoCom AB, entwickelter Layout Manager (http://www.miglayout.com/), der unter BSD Lizenz steht. Es existieren Implementierungen für Swing, Swt und Java FX 2. Der Layout Manager hat einen sehr flexiblen Grid basierten Layout Ansatz und eignet sich für sehr viele Anwendungsfälle. Insbesondere ist damit die Erstellung von formularbasierten Masken sehr intuitiv und einfach umzusetzen.

Um Mig Layout mit jowidgets zu verwenden, wird empfohlen, vorab den Mig Layout Quick Start Guide zu studieren.

[4] Was nicht problematisch war, da MigLayout Implementierungen bereits für Swing und Swt existierten.
[5] In Anlehnung an den Entwickler, der die Portierung durchgeführt hat.

Die Klasse `org.jowidgets.common.widgets.layout.MigLayoutDescriptor` bietet die Möglichkeit, die `LayoutConstraints`, `RowConstraints` und `ColumnConstraints` für das Layout festzulegen. Die Constraints haben die gleiche Bedeutung wie in den Klassen `net.miginfocom.swt.MigLayout` oder `net.miginfocom.swing.MigLayout`.

Im folgenden Beispiel wird ein MigLayout definiert.

```
1     container.setLayout(new MigLayoutDescriptor("wrap", "[][grow, 0::]", "[][]"));
```

Das Layout hat zwei Spalten und zwei Zeilen, wobei die zweite Spalte wächst und eine `MinSize` von `0` hat. Die `LayoutConstraints="wrap"` bedeuten, dass automatisch eine neue Zeile begonnen wird, wenn die aktuelle Zeile voll ist (in diesem Fall also nach zwei Controls).

Die Cell Constraints werden im folgenden Beispiel beim Hinzufügen der Controls zum Container gesetzt:

```
1     final String textFieldCC = "growx, w 0::";
2
3     //row 0, column 0
4     container.add(BPF.textLabel("Field 1"));
5
6     //row 0, column 1
7     container.add(BPF.textField(), textFieldCC);
8
9     //row 1 (autowrap has wraped to row 1), column 0
10    container.add(BPF.textLabel("Field 2"));
11
12    //row 1, column 1
13    container.add(BPF.textField(), textFieldCC);
```

Die Textfelder wachsen horizontal und haben eine `MinSize` von `0`.

Die folgende Abbildung zeigt das Ergebnis:

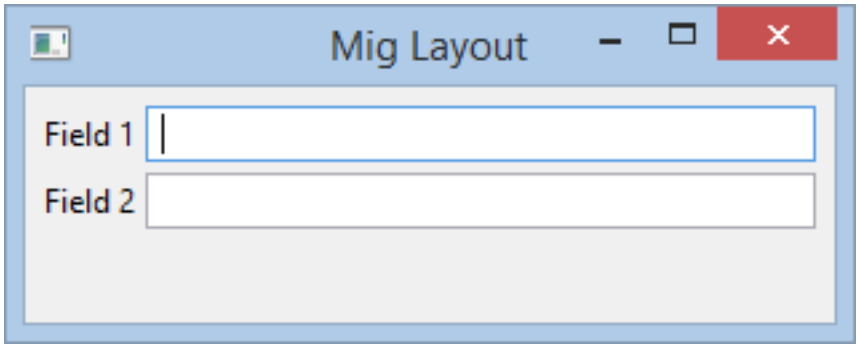


Abbildung 3.2: Mig Layout Beispiel

Hinweise:

- Da native Swt Controls keine `MinSize` haben, verwendet die Swt Mig Layout Implementierung als `MinSize` die `PreferredSize`. Das kann unter Umständen zu Problemen führen. Es wird daher empfohlen, die `MinSize` immer explizit anzugeben, wenn diese von der `PreferredSize` abweicht.
- Wenn eine SPI Implementierung kein natives Mig Layout unterstützt, wird bei der Verwendung des nativen `MigLayoutDescriptor` automatisch Mib Layout verwendet.

- Wenn Mib Layout verwendet wird, wird **kein** natives Mig Layout verwendet, auch wenn die verwendete SPI Implementierung ein natives Mig Layout bereitstellt.
- Die auf einem Control gesetzten Default Größen `MinSize`, `PreferredSize` und `MaxSize` werden vom nativen MigLayout **nicht** ausgewertet. Es wird daher empfohlen, die Methoden zum Ändern der Default Größen auf Controls nicht in Kombination mit MigLayout zu verwenden, um eine größt mögliche Kompatibilität zu gewährleisten!

3.9.2 Custom Layouts

Bevor die vordefinierten Layouts von jowidgets vorgestellt werden, soll vorab die `ILayouter` Schnittstelle besprochen werden, um ein besseres Verständnis zu schaffen, was beim Layouten eines Containers passiert.

Um einen eigenes (custom) Layout zu verwenden, muss diese Schnittstelle implementiert werden:

```
1  public interface ILayouter extends ILayoutDescriptor {
2
3      void layout();
4
5      Dimension getMinSize();
6
7      Dimension getPreferredSize();
8
9      Dimension getMaxSize();
10
11     void invalidate();
12
13 }
```

Die Methode `layout()` ist dafür zuständig, auf den Controls eines Containers die Größe und die Position zu setzen.

Die `ClientArea` ist der Bereich des Containers, welcher für das Zeichnen der Controls zur Verfügung steht. Die `DecoratedSize` ist die gesamte Größe des Containers bei einer gegebenen `ClientArea` Größe. Bei einem Fenster kommen zum Beispiel bei der `DecoratedSize` noch der Rahmen oder ein eventuelles Menü hinzu.

Die Methoden `getMinSize()`, `getPreferredSize()` und `getMaxSize()` geben die minimale, bevorzugte und maximale Größe des Containers zum aktuellen Zeitpunkt zurück, die der Layouter benötigt, um seine Controls zu layouten. Die zurückgegebenen Größen beziehen sich dabei auf die `DecoratedSize` und **nicht** auf die Größe der `ClientArea`.

Die Methode `invalidate()` wird aufgerufen, wenn sich die Struktur des Layouts geändert haben *könnte*. Dies kann ein guter Zeitpunkt sein, um *gecachte* Werte zu löschen.

Die Verwendung der Schnittstelle soll Anhand eines Beispiels verdeutlicht werden. Es wird ein vereinfachtes Fill Layout implementiert. Ein Fill Layout zeichnet das erste sichtbare Control eines Containers so, dass es die `ClientArea` voll ausfüllt. Eine Implementierung könnte wie folgt aussehen:

```
1  final class FillLayout implements ILayouter {
2
3      private final IContainer container;
4
5      private Dimension minSize;
6      private Dimension preferredSize;
7
```

```
8       FillLayout(final IContainer container) {
9           this.container = container;
10      }
11
12      @Override
13      public void layout() {
14          final IControl control = getFirstVisibleControl();
15          if (control != null) {
16              final Rectangle clientArea = container.getClientArea();
17              control.setPosition(clientArea.getPosition());
18              control.setSize(clientArea.getSize());
19          }
20      }
21
22      @Override
23      public Dimension getMinSize() {
24          if (minSize == null) {
25              this.minSize = calcMinSize();
26          }
27          return minSize;
28      }
29
30      @Override
31      public Dimension getPreferredSize() {
32          if (preferredSize == null) {
33              this.preferredSize = calcPreferredSize();
34          }
35          return preferredSize;
36      }
37
38      @Override
39      public Dimension getMaxSize() {
40          return Dimension.MAX;
41      }
42
43      @Override
44      public void invalidate() {
45          minSize = null;
46          preferredSize = null;
47      }
48
49      private Dimension calcMinSize() {
50          final IControl control = getFirstVisibleControl();
51          if (control != null) {
52              return container.computeDecoratedSize(control.getMinSize());
53          }
54          else {
55              return container.computeDecoratedSize(new Dimension(0, 0));
56          }
57      }
58
59      private Dimension calcPreferredSize() {
60          final IControl control = getFirstVisibleControl();
61          if (control != null) {
62              return container.computeDecoratedSize(control.getPreferredSize());
63          }
64          else {
65              return container.computeDecoratedSize(new Dimension(0, 0));
66          }
67      }
68
69      private IControl getFirstVisibleControl() {
70          for (final IControl control : container.getChildren()) {
71              if (control.isVisible()) {
```

```
72                  return control;
73              }
74          }
75          return null;
76      }
77  }
```

Folgendes ist dabei zu beachten:

- Die Werte für die `MinSize` und `PreferredSize` werden, solange `invalidate()` nicht aufgerufen wird, nur ein Mal berechnet. Dadurch kann die Performance gesteigert werden. Gerade bei verschachtelten Containern kann das Berechnen der `PreferredSize` unter Umständen *teuer* sein.
- Die Methoden `calcMinSize()` und `calcPreferredSize()` verwenden die Methode `computeDecoratedSize()` des Containers um aus der für die `ClientArea` gültigen Große die `DecoratedSize` zu berechnen (Zeile 52, 55, 62, 65). Wird dies nicht berücksichtigt, funktioniert das Layout nur für die Container korrekt, wo die `ClientArea` Größe mit der `DecoratedSize` übereinstimmt.

Um einen eigenen (custom) Layouter zu implementieren, ist es eventuell hilfreich, den Source Code der vordefinierten Layout Manager zu studieren. Dieser findet sich unter anderem hier.

3.9.3 Flow Layout

Bei einem Flow Layout werden alle Controls nebeneinander oder untereinander gezeichnet. Die Größe der Controls wird auf deren `PreferredSize` gesetzt, wenn genügend Platz vorhanden ist, ansonsten werden die Controls gleichmäßig verkleinert, jedoch nicht kleiner als ihre `MinSize`.

Die Accessor Klasse `org.jowidgets.api.layout.FlowLayout` liefert einen Zugriff auf ein Flow Layout. Sie hat folgende Methoden:

```
public static ILayoutFactory<ILayouter> get(){...}

public static IFlowLayoutFactoryBuilder builder(){...}
```

Ein `IFlowLayoutFactoryBuilder` hat die folgenden Methoden:

```
IFlowLayoutFactoryBuilder gap(int gap);

IFlowLayoutFactoryBuilder orientation(Orientation orientation);

IFlowLayoutFactoryBuilder vertical();

IFlowLayoutFactoryBuilder horizontal();

ILayoutFactory<ILayouter> build();
```

Die Orientation gibt an, ob die Elemente nebeneinander oder untereinander angeordnet werden. Die default `Orientation` ist `HORIZONTAL`. Die Methoden `vertical()` und `horizontal()` setzen ebenfalls die `Orientation`, jedoch mit verkürzter Schreibweise. Der `gap` definiert den *Freiraum* zwischen den Controls. Der default `gap` beträgt 4 Pixel. Die Methode `build()` liefert eine neue `ILayoutFactory` zurück.

Folgendes Beispiel demonstriert Verwendung:

```
container.setLayout(FlowLayout.get());
container.add(BPF.textLabel().setText("Attribute1"));
final ITextControl textField = container.add(BPF.textField());
textField.setText("This is the most common attribute");
```

Die folgende Abbildung zeigt das Ergebnis:

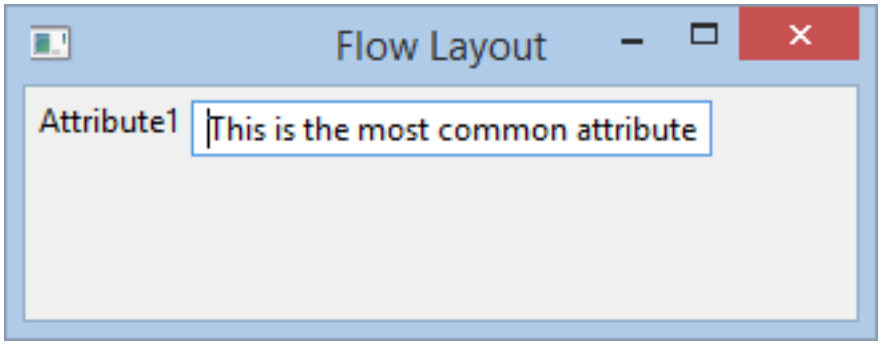


Abbildung 3.3: Flow Layout Beispiel 1

Im nächsten Beispiel wird das FlowLayout vertikal ausgerichtet:

```
container.setLayout(FlowLayout.builder().gap(0).vertical().build());
container.add(BPF.textLabel().setText("Attribute1"));
final ITextControl textField = container.add(BPF.textField());
textField.setText("This is the most common attribute");
```

Die folgende Abbildung zeigt das Ergebnis:

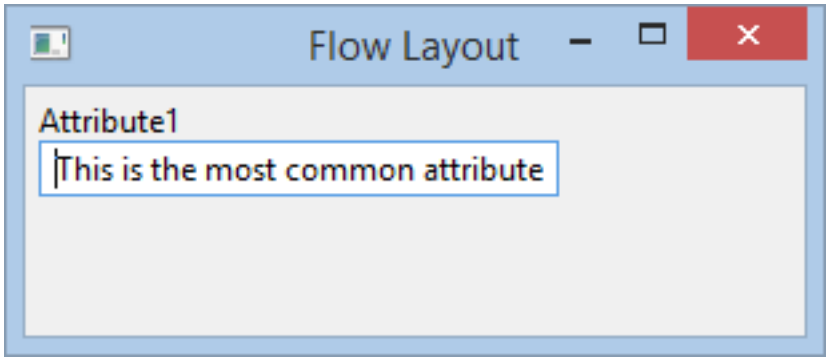


Abbildung 3.4: Flow Layout Beispiel 2

3.9.4 Border Layout

Ein Border Layout teilt einen Container in fünf mögliche Bereiche auf: Center, Top, Bottom, Left und Right. In jedem Bereich kann sich genau ein Control befinden. Die folgende Abbildung zeigt ein typisches Border Layout:

In der Mitte (center) befindet sich eine TextArea, oben, rechts und links je eine Toolbar und unten ein TextFeld.

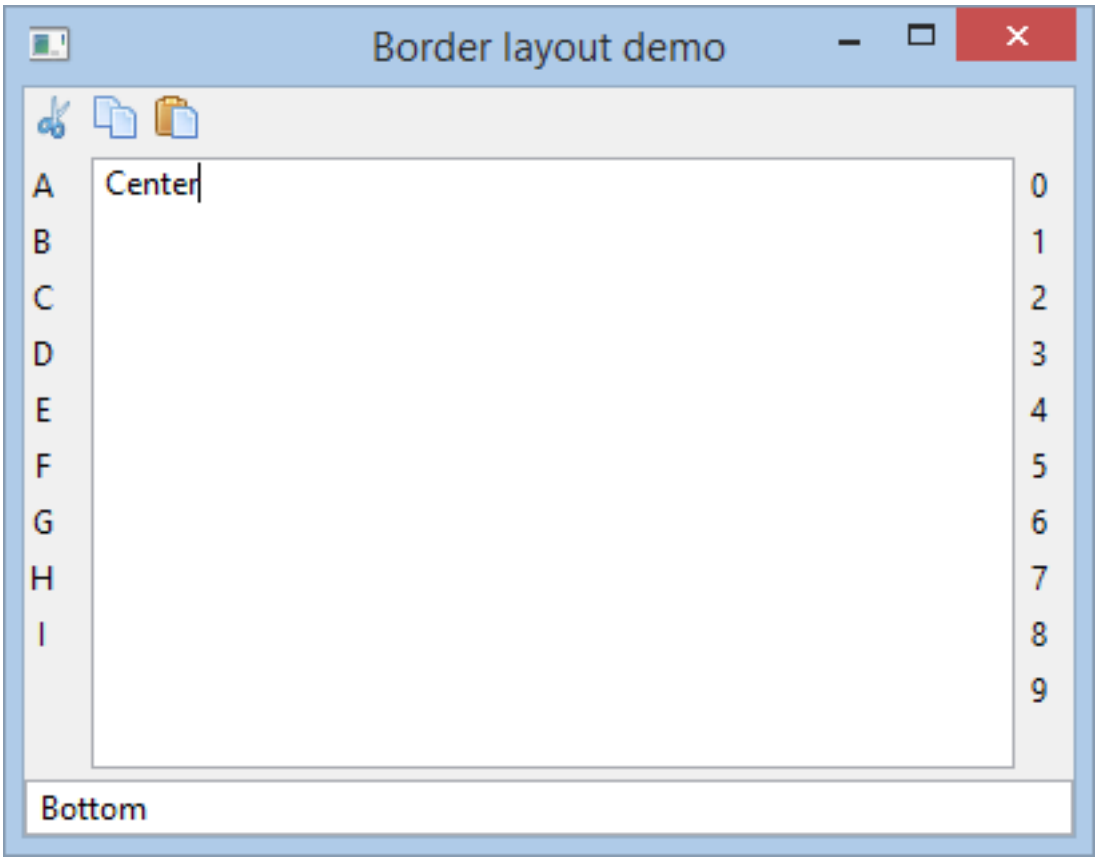


Abbildung 3.5: Border Layout Beispiel

Die Accessor Klasse `org.jowidgets.api.layout.BorderLayout` liefert einen Zugriff auf ein Border Layout. Sie hat folgende Methoden:

```
public static ILayoutFactory<ILayouter> get(){...}

public static IBorderLayoutFactoryBuilder builder(){...}
```

Ein `IBorderLayoutFactoryBuilder` hat die folgenden Methoden:

```
IBorderLayoutFactoryBuilder margin(int margin);

IBorderLayoutFactoryBuilder gap(int gap);

IBorderLayoutFactoryBuilder gapX(int gapX);

IBorderLayoutFactoryBuilder gapY(int gapY);

IBorderLayoutFactoryBuilder marginLeft(int marginLeft);

IBorderLayoutFactoryBuilder marginRight(int marginRight);

IBorderLayoutFactoryBuilder marginTop(int marginTop);

IBorderLayoutFactoryBuilder marginBottom(int marginBottom);

ILayoutFactory<ILayouter> build();
```

Der `margin` definiert den äußeren Abstand zur `ClientArea` des Containers. Er kann separat für recht, links, oben und unten oder für alle Seiten zusammen gesetzt werden. Der default `margin` ist `0`. Der `gap` definiert den Abstand zwischen den einzelnen Bereichen. Er kann separat für die x-Richtung und y-Richtung oder für beide zusammen gesetzt werden. Der default `gap` ist 4. Die Methode `build()` liefert eine neue `ILayoutFactory` zurück.

Um festzulegen, in welchem Bereich ein Control hinzugefügt wird, muss auf diesem der LayoutConstraint `BorderLayoutConstraints` gesetzt werden. Das folgende Beispiel verdeutlicht dies:

```
1  IToolBar top = container.add(BPF.toolBar(), BorderLayout.TOP);
2  IToolBar left = container.add(BPF.toolBar().setVertical(), BorderLayout.LEFT);
3  ITextArea center = container.add(BPF.textArea(), BorderLayout.CENTER);
4  IToolBar right = container.add(BPF.toolBar().setVertical(), BorderLayout.RIGHT);
5  ITextControl bottom = container.add(BPF.textField(), BorderLayout.BOTTOM);
```

3.9.5 Fill Layout

Ein Fill Layout zeichnet ausschließlich das erste (sichtbare) Control eines Containers. Dabei wird unabhängig von der `MinSize`, `PreferredSize` oder `MaxSize` die komplette `ClientArea` abzüglich des `margin` für das Control verwendet.

Die Accessor Klasse `org.jowidgets.api.layout.BorderLayout` liefert einen Zugriff auf ein Fill Layout. Sie hat folgende Methoden:

```
public static ILayoutFactory<ILayouter> get(){...}

public static IFillLayoutFactoryBuilder builder(){...}
```

Ein `IFillLayoutFactoryBuilder` hat die folgenden Methoden:

```
IFillLayoutFactoryBuilder margin(int margin);

IFillLayoutFactoryBuilder marginLeft(int marginLeft);

IFillLayoutFactoryBuilder marginRight(int marginRight);

IFillLayoutFactoryBuilder marginTop(int marginTop);

IFillLayoutFactoryBuilder marginBottom(int marginBottom);

ILayoutFactory<ILayouter> build();
```

Der `margin` definiert den äußeren Abstand zur `ClientArea` des Containers. Er kann separat für recht, links, oben und unten oder für alle Seiten zusammen gesetzt werden. Der default `margin` ist `0`. Die Methode `build()` liefert eine neue `ILayoutFactory` zurück.

Folgendes Beispiel demonstriert Verwendung:

```
1  container.setLayout(FillLayout.builder().margin(5).build());
2  final ITextArea textArea = container.add(BPF.textArea());
3  textArea.setText("Some text in this text area");
```

Die folgende Abbildung zeigt das Ergebnis:

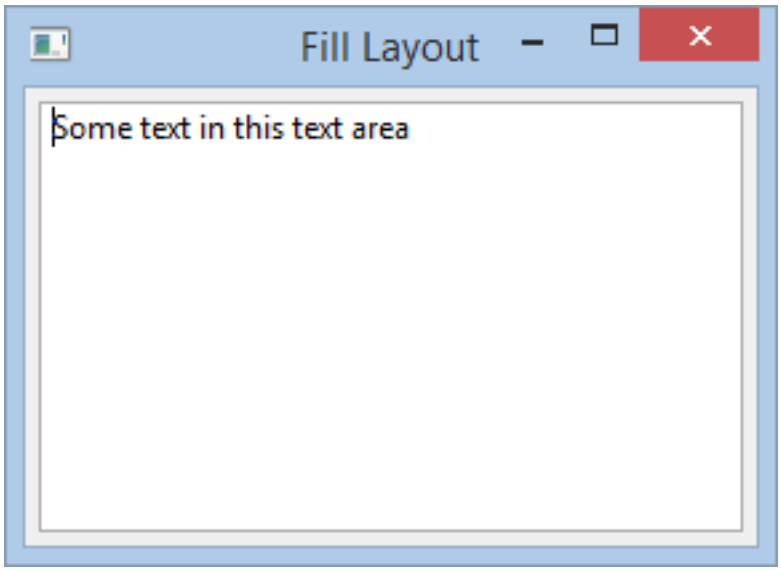


Abbildung 3.6: Fill Layout Beispiel 1

3.9.6 Cached Fill Layout

Das Cached Fill Layout wurde entworfen, um das Layouten komplexer Controls zu optimieren. Immer wenn sich zum Beispiel die Größe eines Fensters oder eines Bereichs innerhalb eines Split Composites ändert, wird auf einem Layouter die `invalidate()` Methode aufgerufen. Wenn man keine Kenntnis über die darunter liegenden Controls hat, ist dieses Vorgehen auch sinnvoll, denn durch das Ändern der Größe könnte sich auch das Layout ändern. Allerdings ist dieses Vorgehen auch *teuer* und unter Umständen werden dabei die immer gleichen `PreferredSize` Werte berechnet.

Es gibt Situationen, in denen man weiß, dass das Ändern der Größe keinen Einfluss auf die `PreferredSize` der Kind Controls hat. In diesem Fall kann man ein Cached Fill Layout verwenden. Dies berechnet die `MinSize`, `PreferredSize` und `MaxSize` bei einem `invalidate()` nicht neu. Ansonsten ist es mit dem Fill Layout zu vergleichen, da auch hier die gesamte `ClientArea` ausgenutzt wird.

Die Accessor Klasse `org.jowidgets.api.layout.CachedFillLayout` liefert einen Zugriff auf ein Cached Fill Layout. Sie hat folgende Methode:

```
public static ILayoutFactory<ICachedFillLayout> get(){...}
```

Um den Cache explizit zu löschen, kann auf dem `ICachedFillLayout` die folgende Methode verwendet werden:

```
void clearCache();
```

3.9.7 Mib Layout

Im Rahmen einer Bacheloarbeit wurde Mig Layout (http://www.miglayout.com/) für jowidgets portiert. Die Portierung implementiert die `ILayouter` Schnittstelle. Es wurden die meisten Funktionen portiert. Zudem wurde auch die original MigLayout Demo Applikation auf Basis der Portierung umgesetzt, der Source Code findet sich hier. Die folgende Abbildung zeigt die Demo Applikation:

3.9.7.1 Mib Layout Verwendung in jowidgets

Um Mib Layout in jowidgets zu verwenden, wird empfohlen, vorab den Mig Layout Quick Start Guide zu studieren.

Die Accessor Klasse `org.jowidgets.api.layout.miglayout.MigLayout` liefert einen Zugriff auf das Mib Layout. Sie hat folgende Methoden:

```
public static ILayoutFactory<IMigLayout> get(){...}

public static ILayoutFactory<IMigLayout> create(
    final String layoutConstraints) {...}

public static ILayoutFactory<IMigLayout> create(
    final String columnConstraints,
    final String rowConstraints) {...}

public static ILayoutFactory<IMigLayout> create(
    final String layoutConstraints,
    final String columnConstraints,
    final String rowConstraints) {...}

public static ILayoutFactory<IMigLayout> create(
    final MigLayoutDescriptor descriptor) {...}

public static IMigLayoutFactoryBuilder builder() {...}
```

Ein `IMigLayoutFactoryBuilder` hat die folgenden Methoden:

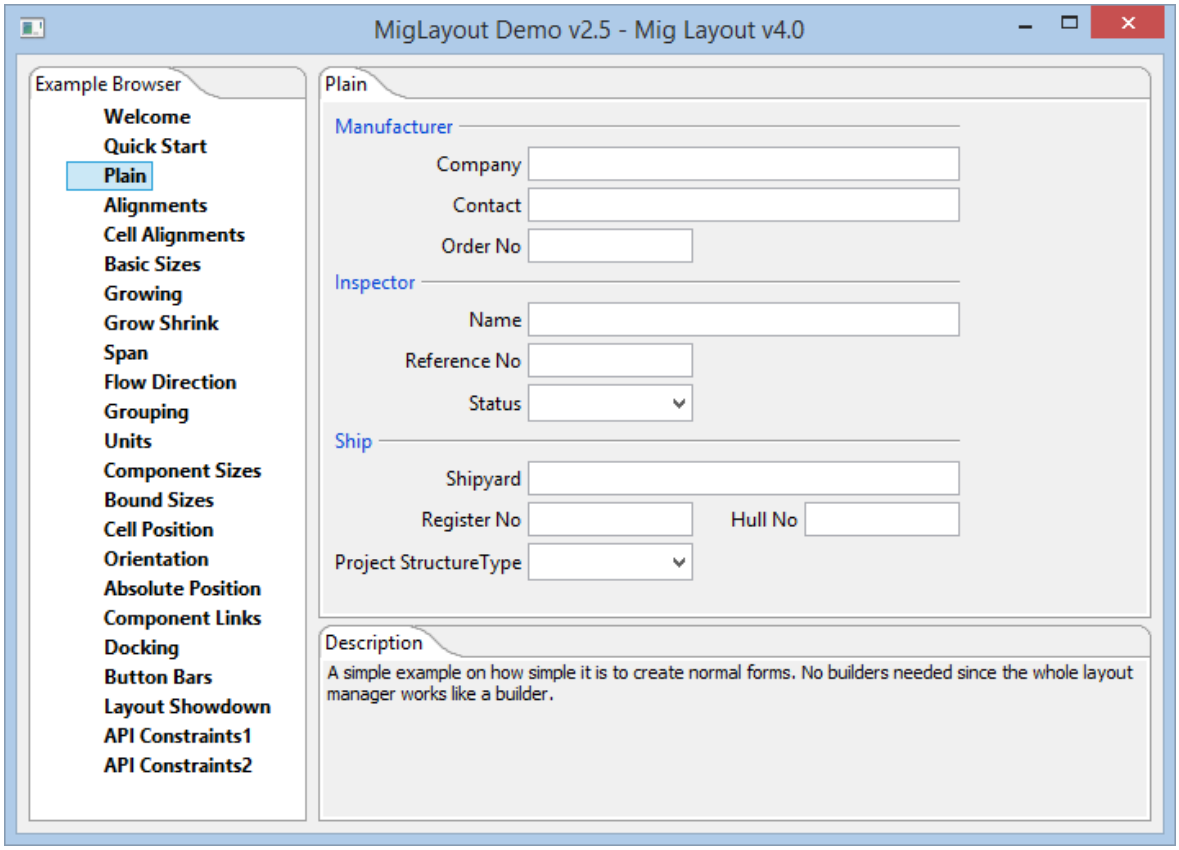


Abbildung 3.7: Mib Layout Demo Applikation

```
IMigLayoutFactoryBuilder descriptor(MigLayoutDescriptor descriptor);

IMigLayoutFactoryBuilder columnConstraints(String constraints);

IMigLayoutFactoryBuilder constraints(String constraints);

IMigLayoutFactoryBuilder rowConstraints(IAC constraints);

IMigLayoutFactoryBuilder columnConstraints(IAC constraints);

IMigLayoutFactoryBuilder constraints(ILC constraints);

ILayoutFactory<IMigLayout> build();
```

Constraints können wie im original Mig Layout mit Hilfe von Strings oder durch Builder (siehe Mib Layout Constraints Builder) erzeugt werden. Die Methode `build()` liefert eine neue `ILayoutFactory<IMigLayout>` zurück. Die Schnittstelle `IMiglayout` hat die folgenden Methoden:

```
void setLayoutConstraints(Object constraints);

Object getLayoutConstraints();

void setColumnConstraints(Object constraints);

Object getColumnConstraints();

void setRowConstraints(Object constraints);

Object getRowConstraints();

void setConstraintMap(Map<IControl, Object> map);

Map<IControl, Object> getConstraintMap();

boolean isManagingComponent(IControl control);
```

Die Methoden sind identisch zur Klasse `net.miginfocom.swt.MigLayout`.

3.9.7.2 Mib Layout Constraints Builder

Die Accessor Klasse org.jowidgets.api.layout.miglayout.LC kann für die Builder basierte Erzeugung von `LayoutConstraints` verwendet werden. Sie hat die folgende Methode:

```
public static ILC create() {...}
```

Die Schnittstelle `org.jowidgets.api.layout.miglayout.ILC` hat die gleichen Methoden wie die original Mig Layout Klasse `net.miginfocom.layout.LC`.

Die Accessor Klasse org.jowidgets.api.layout.miglayout.AC kann für die Builder basierte Erzeugung von `AxisConstraints` (also `ColumnConstraints` und `RowConstraints`) verwendet werden. Sie hat die folgende Methode:

```
public static IAC create() {...}
```

Die Schnittstelle `org.jowidgets.api.layout.miglayout.IAC` hat die gleichen Methoden wie die original Mig Layout Klasse `net.miginfocom.layout.AC`.

Die Accessor Klasse org.jowidgets.api.layout.miglayout.CC kann für die Builder basierte Erzeugung von `ComponentConstraints` verwendet werden. Sie hat die folgende Methode:

```
public static ICC create() {...}
```

Die Schnittstelle `org.jowidgets.api.layout.miglayout.ICC` hat die gleichen Methoden wie die original Mig Layout Klasse `net.miginfocom.layout.CC`.

3.9.7.3 Mib Layout Platform Defaults

Die Accessor Klasse `org.jowidgets.api.layout.miglayout.PlatformDefaults` bietet den Zugriff auf die Portierung der Klasse `net.miginfocom.layout.PlatformDefaults`.

3.9.7.4 Mib Layout Beispiel

Im folgenden wurde das Beispiel aus dem Abschnitt Mig Layout (nativ) mit Mib Layout umgesetzt:

```
1  container.setLayout(MigLayout.create("wrap", "[][grow, 0::]", "[][]"));
2
3  final String textFieldCC = "growx, w 0::";
4
5  //row 0, column 0
6  container.add(BPF.textLabel("Field 1"));
7
8  //row 0, column 1
9  container.add(BPF.textField(), textFieldCC);
10
11 //row 1 (autowrap has wraped to row 1), column 0
12 container.add(BPF.textLabel("Field 2"));
13
14 //row 1, column 1
15 container.add(BPF.textField(), textFieldCC);
```

Der einzige Unterschied findet sich in Zeile 1 bei der Definition des Layouts, der Rest ist identisch. Die folgende Abbildung zeigt das Ergebnis:

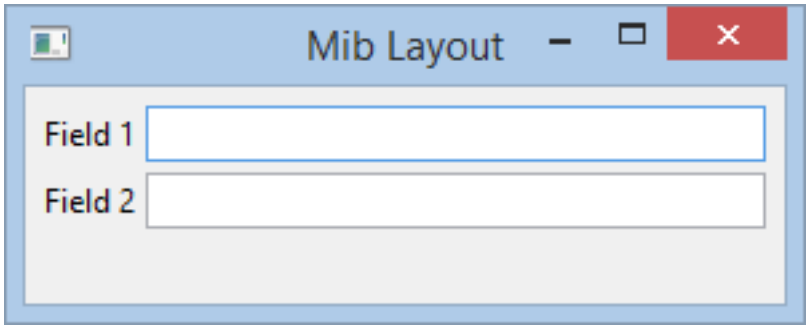


Abbildung 3.8: Mib Layout Beispiel

Im folgenden Beispiel wird das gleiche wie oben mit Hilfe der Constraints Builder umgesetzt:

```
container.setLayout(
    MigLayout.builder()
        .constraints(LC.create().wrap())
        .columnConstraints(AC.create().index(1).grow().size("0::")).build());

final ICC textFieldCC = CC.create().growX().width("0::");

//row 0, column 0
container.add(BPF.textLabel("Field 1"));

//row 0, column 1
container.add(BPF.textField(), textFieldCC);

//row 1 (autowrap has wraped to row 1), column 0
container.add(BPF.textLabel("Field 2"));

//row 1, column 1
container.add(BPF.textField(), textFieldCC);
```

3.9.8 Null Layout

Der Null Layout Layouter implementiert eine *leere* `layout()` Methode. Die Methoden `getMinSize()`, `getPreferredSize()` und `getMaxSize()` liefern die aktuelle Größe des Containers. Bei einem NullLayout müssen die Position und die Größe der Controls daher manuell gesetzt werden.

Die Accessor Klasse `org.jowidgets.api.layout.NullLayout` liefert einen Zugriff auf ein Null Layout. Sie hat die folgende Methode:

```
public static ILayoutFactory<ILayouter> get(){...}
```

Das folgende Beispiel zeigt die Verwendung eines Null Layout:

```
container.setLayout(NullLayout.get());

final int x = 10;
final int y = 10;

for (int i = 0; i < 5; i++) {
    final IButton button = container.add(BPF.button());
    button.setPosition(x + i * 20, y + i * 40);
    button.setText("Button A - " + i);
    button.setSize(button.getPreferredSize());
}

for (int i = 0; i < 5; i++) {
    final IButton button = container.add(BPF.button());
    button.setPosition(x + 160 + i * 20, y + (4 - i) * 40);
    button.setText("Button B - " + i);
    button.setSize(200, 30);
}
```

Die folgende Abbildung zeigt das Ergebnis:

3.9.9 Preferred Size Layout

Bei einem Preferred Size Layout wird in der `layout()` Methode nur die `PreferredSize` der Controls gesetzt. Die Position muss wie beim Null Layout manuell gesetzt werden.

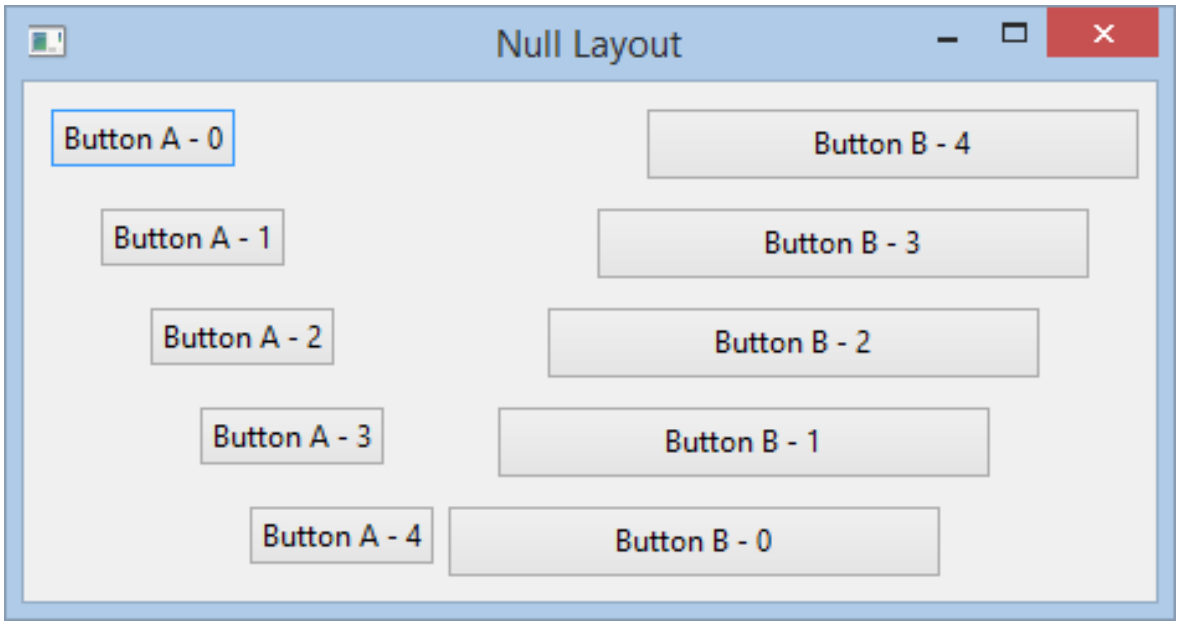


Abbildung 3.9: Null Layout Beispiel

Die Accessor Klasse `org.jowidgets.api.layout.PreferredSizeLayout` liefert einen Zugriff auf ein Preferred Size Layout. Sie hat die folgende Methode:

```
public static ILayoutFactory<ILayouter> get(){...}
```

Das folgende Beispiel zeigt die Verwendung eines Preferred Size Layout:

```
1  container.setLayout(PreferredSizeLayout.get());
2
3  final int x = 10;
4  final int y = 10;
5
6  for (int i = 0; i < 5; i++) {
7      final IButton button = container.add(BPF.button());
8      button.setPosition(x + i * 20, y + i * 40);
9      button.setText("Button A - " + i);
10 }
11
12 for (int i = 0; i < 5; i++) {
13     final IButton button = container.add(BPF.button());
14     button.setPosition(x + 160 + i * 20, y + (4 - i) * 40);
15     button.setText("Button B - " + i);
16 }
```

Die folgende Abbildung zeigt das Ergebnis:

3.10 Menüs und Items

Der folgende Abschnitt gibt eine Einführung in die Verwendung von nativen Menüs und Menü Items in jowidgets. Menüs können auch mit Hilfe von Menü und Item Models erstellt werden, was einige

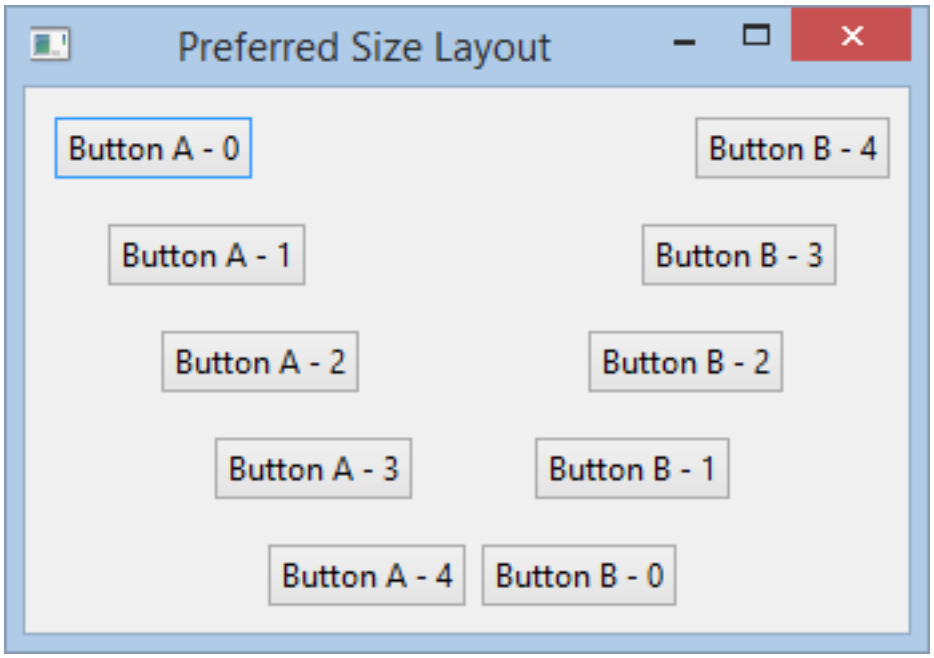


Abbildung 3.10: Preferred Size Layout Beispiel

Vorteile mit sich bringt. Der hier beschriebene native Ansatz ist mit dem anderer UI Frameworks vergleichbar.

3.10.1 Menu Bar

Eine Menu Bar ist eine *Menüleiste* für ein Frame. Folgende Abbildung zeigt ein Frame mit einer Menu Bar, welche ein **File** und ein **Edit** Menü enthält:

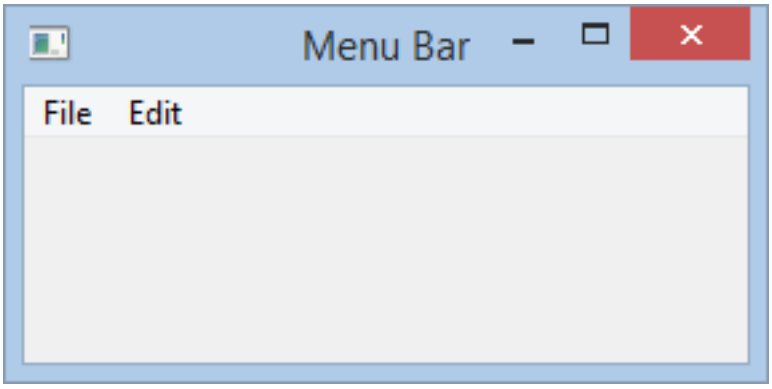


Abbildung 3.11: Menu Bar Beispiel

Eine Menu Bar erhält man für ein Frame mit Hilfe der folgende Methode:

```
IMenuBar createMenuBar();
```

Falls noch keine Menu Bar existiert, wird eine neue erzeugt und zurückgegeben. Andernfalls wir die zuletzt erzeugte (welche noch nicht disposed wurde) zurückgegeben. Es folgt eine kurze Beschreibung der Menu Bar Methoden:

3.10.1.1 Menu Bar Model

Mit Hilfe der folgenden Methoden kann das Model gesetzt und ausgelesen werden. Siehe auch Menü und Item Models.

```
    void setModel(IMenuBarModel model);

    IMenuBarModel getModel();
```

3.10.1.2 Hinzufügen von Menüs

Mit Hilfe der folgenden Methoden kann ein Main Menu hinzugefügt werden:

```
    IMainMenu addMenu(IMainMenuDescriptor descriptor);

    IMainMenu addMenu(int index, IMainMenuDescriptor descriptor);

    IMainMenu addMenu(String name);

    IMainMenu addMenu(String name, char mnemonic);

    IMainMenu addMenu(int index, String name);
```

Die ersten beiden Methoden verwenden einen Descriptor (BluePrint) zum Hinzufügen. Bei den anderen Methoden handelt es sich um Convenience Methoden, mit welchen der Menu Name und Mnemonic direkt angeben werden kann. Wird ein `index` angegeben, wird das Menü an der entsprechenden Stelle eingefügt, ansonsten am Ende.

Das folgende Beispiel soll die Verwendung verdeutlichen:

```
1   final IMenuBar menuBar = frame.createMenuBar();
2
3   final IMainMenu fileMenu = menuBar.addMenu(BPF.mainMenu().setText("File"));
4   final IMainMenu editMenu = menuBar.addMenu(BPF.mainMenu().setText("Edit"));
```

Mit Hilfe der Convenience Methoden kann man das auch kürzer schreiben:

```
1   final IMenuBar menuBar = frame.createMenuBar();
2
3   final IMainMenu fileMenu = menuBar.addMenu("File");
4   final IMainMenu editMenu = menuBar.addMenu("Edit");
```

Folgender Aufruf fügt ein Menü zwischen **File** und **Edit** ein:

```
1   final IMainMenu searchMenu = menuBar.addMenu(1, "Search");
```

Die folgende Abbildung zeigt das Gesamtergebnis:

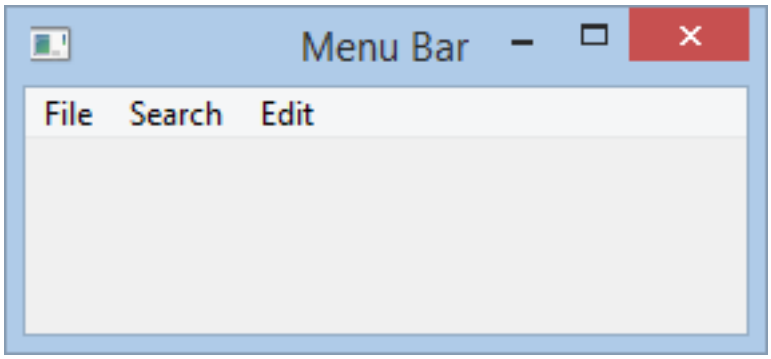


Abbildung 3.12: Menu Bar Beispiel 2

3.10.1.3 Entfernen von Menüs

Ein Menü kann entweder mit Hilfe der folgenden Methoden entfernt werden:

```
boolean remove(IMenu menu);

void remove(final int index);

void removeAll();
```

oder indem auf dem Menü welches entfernt werden soll, die dispose() Methode aufgerufen wird.

Der Aufruf:

```
menuBar.remove(searchMenu);
```

hat also den gleichen Effekt wie:

```
searchMenu.dispose();
```

3.10.1.4 Zugriff auf die Menüs einer Menu Bar

Mit Hilfe der folgenden Methode erhält man eine Liste der vorhandenen Menüs.

```
List<IMenu> getMenus();
```

Dabei handelt es sich um eine nicht modifizierbare Kopie der derzeit vorhandenen Menüs.

3.10.2 Die Schnittstelle IMenu

Die Schnittstelle IMenu liefert die Basisfunktionen für das Main Menu das Sub Menu sowie das Popup Menu. Es folgt eine kurze Beschreibung der wichtigsten Methoden:

3.10.2.1 Menu Model

Mit Hilfe der folgenden Methoden kann das Model gesetzt und ausgelesen werden. Siehe auch Menü und Item Models.

```
void setModel(IMenuModel model);

IMenuModel getModel();
```

3.10.2.2 Hinzufügen von Items

Mit Hilfe der folgenden Methoden können Items zu einem Menü hinzugefügt werden:

```
<WIDGET_TYPE extends IMenuItem> WIDGET_TYPE
    addItem(IWidgetDescriptor<? extends WIDGET_TYPE> descriptor);

<WIDGET_TYPE extends IMenuItem> WIDGET_TYPE
    addItem(int index, IWidgetDescriptor<? extends WIDGET_TYPE> descriptor);

IActionMenuItem addAction(IAction action);

IActionMenuItem addAction(int index, IAction action);

IMenuItem addSeparator();

IMenuItem addSeparator(int index);
```

Die ersten beiden Methoden verwenden dazu einen Descriptor (BluePrint). Folgende Items können so hinzugefügt werden:

- Action Menü Item
- Checked Menu Item
- Radio Menu Item
- Separator Menu Item
- Sub Menu

Die Methoden `addAction()` fügen ein Action Menu Item hinzu, welches an die übergebene `IAction` (siehe Actions and Commands) gebunden wird. Mit Hilfe von `addSeparator()` kann ein Separator Menu Item mit verkürzter Schreibweise hinzugefügt werden. Wird ein `index` angegeben, wird das Item an der entsprechenden Stelle eingefügt, ansonsten am Ende.

Das folgende Beispiel demonstriert die Verwendung:

```
1  IMainMenu menu = menuBar.addMenu("Menu");
2  IActionMenuItem action1 = menu.addItem(BPF.menuItem("Action1"));
3  IActionMenuItem action2 = menu.addItem(BPF.menuItem("Action1"));
4  menu.addItem(BPF.menuSeparator());
5  ISelectableMenuItem opt1 = menu.addItem(BPF.checkedMenuItem("Option1").setSelected(true));
6  ISelectableMenuItem opt2 = menu.addItem(BPF.checkedMenuItem("Option2"));
7  menu.addSeparator();
8  ISelectableMenuItem radio1 = menu.addItem(BPF.radioMenuItem("Radio1").setSelected(true));
9  ISelectableMenuItem radio2 = menu.addItem(BPF.radioMenuItem("Radio2"));
10 ISelectableMenuItem radio3 = menu.addItem(BPF.radioMenuItem("Radio3"));
11 menu.addSeparator();
12 ISubMenu subMenu = menu.addItem(BPF.subMenu("Sub Menu"));
13 IActionMenuItem subaction subMenu.addItem(BPF.menuItem("Subaction"));
```

In Zeile 4 wird eine Separator mit Hilfe des BluePrint hinzugefügt, in Zeile 7 und 11 mit Hilfe der Convenience Methode `addSeparator()`.

Das Ergebnis sieht wie folgt aus:

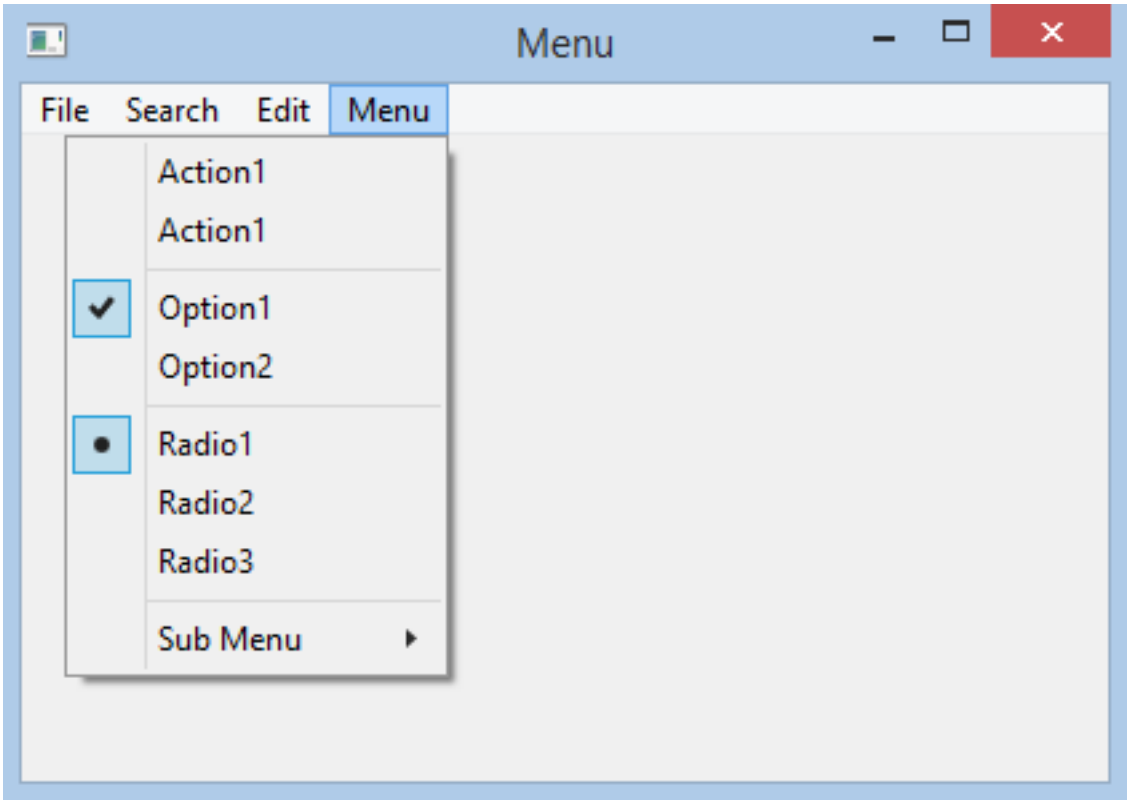


Abbildung 3.13: Menu Beispiel 1

3.10.3 Die Schnittstelle IMenuItem

Die Schnittstelle `IMenuItem` liefert die Basisfunktionen für alle Menu Items. Dazu zählen das Action Menü Item, Checked Menu Item, Radio Menu Item, Separator Menu Item und das Sub Menu. Ein `IMenuItem` ist von `IItem` und somit von `IWidget` abgeleitet. Ein `IMenuItem` hat die folgenden weiteren Methoden:

```
IMenuItemModel getModel();

void setModel(IMenuItemModel model);

void setMnemonic(char mnemonic);
```

Mit Hilfe von `getModel()` und `setModel()` wird das Model gesetzt oder geholt (siehe auch Menü und Item Models). Der Mnemonic definiert das Tastenkürzel, mit welchen das Item in Kombination mit der Taste ALT (z.B. unter Windows) geöffnet bzw. ausgeführt werden kann. Auf manchen Plattformen wird das Mnemonic Zeichen unterstrichen dargestellt, falls es im Label Text vorkommt.

3.10.4 Main Menu

Ein Main Menu ist ein einzelnes Hauptmenü in einer Menu Bar. Die Schnittstelle `IMainMenu` hat neben den von `IMenu` und `IWidget` geerbten Methoden die folgenden weiteren:

```
void setText(String text);

void setMnemonic(char mnemonic);
```

Beim `text` handelt es sich um den Label Text des Menüs. Der Mnemonic definiert das Tastenkürzel, mit welchen das Menü in Kombination mit der Taste ALT (z.B. unter Windows) geöffnet werden kann. Auf manchen Plattformen wird das Mnemonic Zeichen unterstrichen dargestellt, falls es im Menü Label Text vorkommt.

3.10.4.1 Main Menu BluePrint

Ein Main Menu kann (u.A.) mit Hilfe eines `IMainMenuBluePrint` erzeugt werden. Die Klasse `BPF` liefert die folgenden Methoden für die Erzeugung eines BluePrint:

```
public static IMainMenuBluePrint mainMenu(){...}

public static IMainMenuBluePrint mainMenu(final String text){...}
```

Die zweite Methode ermöglicht das gleichzeitige setzen des Label Textes auf dem BluePrint bei der Erzeugung.

Ein `IMainMenuBluePrint` hat die folgenden Methoden zur Konfiguration:

```
IMainMenuBluePrint setText(String text);

IMainMenuBluePrint setMnemonic(Character mnemonic);
```

Diese definieren, analog zu den Methoden auf `IMainMenu` den Label Text und das Mnemonic.

3.10.5 Sub Menu

Ein Sub Menu ist ein Untermenü eines `IMenu` und ist somit `IMenu` und `IMenuItem` zugleich. Die Schnittstelle `ISubMenu` hat neben den von `IMenu`, `IMenuItem`, `IItem`, `IWidget` und `ISubMenu` geerbten **keine** weiteren Methoden.

3.10.5.1 Sub Menu BluePrint

Ein Sub Menu kann (u.A.) mit Hilfe eines `ISubMenuBluePrint` erzeugt werden. Die Klasse `BPF` liefert die folgenden Methoden für die Erzeugung eines BluePrint:

```
public static ISubMenuBluePrint subMenu() {...}

public static ISubMenuBluePrint subMenu(final String text) {...}
```

Die zweite Methode ermöglicht das gleichzeitige setzen des Label Textes auf dem BluePrint bei der Erzeugung.

Ein `ISubMenuBluePrint` hat die folgenden Methoden zur Konfiguration:

```
ISubMenuBluePrint setText(String text);

ISubMenuBluePrint setToolTipText(String toolTipText);

ISubMenuBluePrint setIcon(IImageConstant icon);

ISubMenuBluePrint setMnemonic(Character mnemonic);
```

Mit den ersten drei Methoden kann, analog zu einen `IItem` der Label Text, das Tooltip und das Icon gesetzt werden. Mit Hilfe der Methode `setMnemonic()` lässt sich das Mnemonic (vgl. `IMenuItem`) festgelegt.

3.10.5.2 Beispiel

Das folgende Beispiel zeigt die Verwendung von Sub Menus:

```
1  final ISubMenu subMenu1 = menu.addItem(BPF.subMenu("Submenu 1"));
2  final ISubMenu subMenu2 = menu.addItem(BPF.subMenu("Submenu 2"));
3
4  subMenu1.addItem(BPF.menuItem("Item1"));
5  subMenu1.addItem(BPF.menuItem("Item2"));
6  final ISubMenu subSubMenu1 = subMenu1.addItem(BPF.subMenu("Subsubmenu1"));
7
8  final ISubMenu subSubMenu2 = subMenu1.addItem(BPF.subMenu("Subsubmenu2"));
9  subSubMenu2.addItem(BPF.menuItem("Item1"));
10 subSubMenu2.addItem(BPF.menuItem("Item2"));
```

Die folgende Abbildung zeigt das Ergebnis:

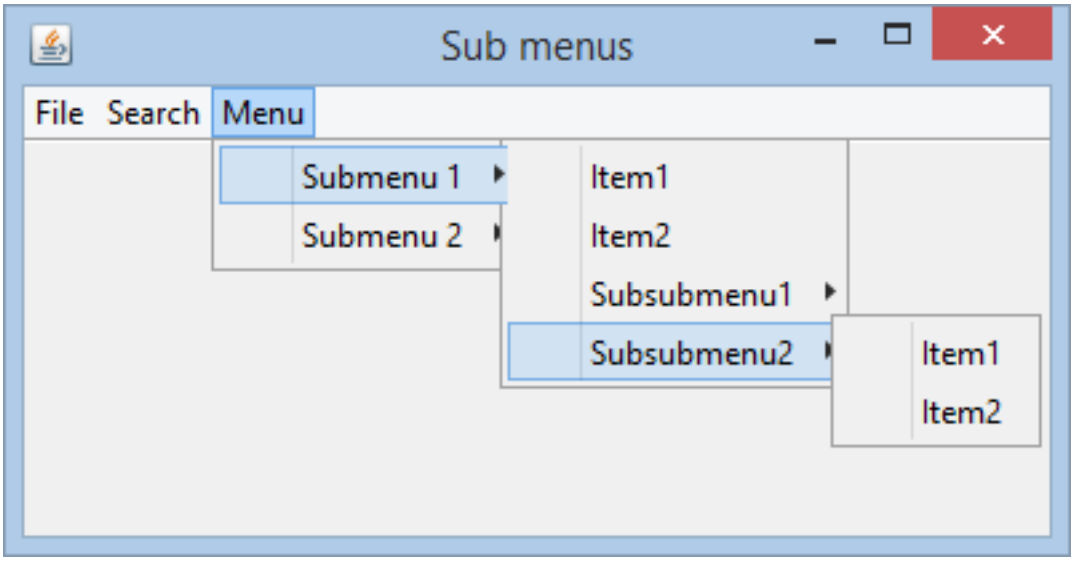


Abbildung 3.14: Sub Menu Beispiel

3.10.6 Popup Menu

Ein Popup Menu ist ein Kontexmenü, welches von einer Komponente mit Hilfe der folgenden Methode erzeugt werden kann:

```
IPopupMenu createPopupMenu();
```

Dabei können für eine Komponente beliebig viele Popup Menüs erzeugt werden.

Die Schnittstelle `IPopupMenu` hat neben den von `IMenu` und `IWidget` geerbten Methoden die folgende weitere zur Anzeige des Menüs:

```
void show(Position position);
```

Die `position` muss dabei bezüglich des Koordinatensystems angegeben werden, welches das PopupMenu erzeugt hat und darf nicht `null` sein.

Folgendes Beispiel zeigt die Verwendung eines PopMenüs:

```
1  //create the popup menu
2  IPopupMenu menu = frame.createPopupMenu();
3
4  //add items to the menu
5  IActionMenuItem action1 = menu.addItem(BPF.menuItem("Action1"));
6  IActionMenuItem action2 = menu.addItem(BPF.menuItem("Action1"));
7  menu.addSeparator();
8  ISelectableMenuItem option1 = menu.addItem(BPF.checkedMenuItem("Option1").setSelected(true));
9  ISelectableMenuItem option2 = menu.addItem(BPF.checkedMenuItem("Option2"));
10 menu.addSeparator();
11 ISelectableMenuItem radio1 = menu.addItem(BPF.radioMenuItem("Radio1").setSelected(true));
12 ISelectableMenuItem radio2 = menu.addItem(BPF.radioMenuItem("Radio2"));
13 ISelectableMenuItem radio3 = menu.addItem(BPF.radioMenuItem("Radio3"));
14 menu.addSeparator();
15 ISubMenu subMenu = menu.addItem(BPF.subMenu("Sub Menu"));
16 subMenu.addItem(BPF.menuItem("Subaction"));
17
18 //add a popup detection listener that shows the menu
19 frame.addPopupDetectionListener(new IPopupDetectionListener() {
20     @Override
21     public void popupDetected(final Position position) {
22         menu.show(position);
23     }
24 });
```

In Zeile 2 wird ein neues PopupMenu erzeugt. In den Zeilen 5 - 16 werden diesem Items hinzugefügt. In Zeile 19 wird ein PopupDetectionListener hinzugefügt, der das Menü bei einem PopupEvent sichtbar macht. Die folgende Abbildung zeigt das Ergebnis:

3.10.7 Action Menu Item

Ein Action Menu Item ist ein Menu Item, welches beim *Anklicken* ein Action Event auslöst. Die folgende Abbildung zeigt ein Menu mit zwei Action Menu Items:

Neben denen von IMenuItem, Item und IWidget geerbten Methoden hat die Schnittstelle `IActionMenuItem` die folgenden weiteren Funktionen:

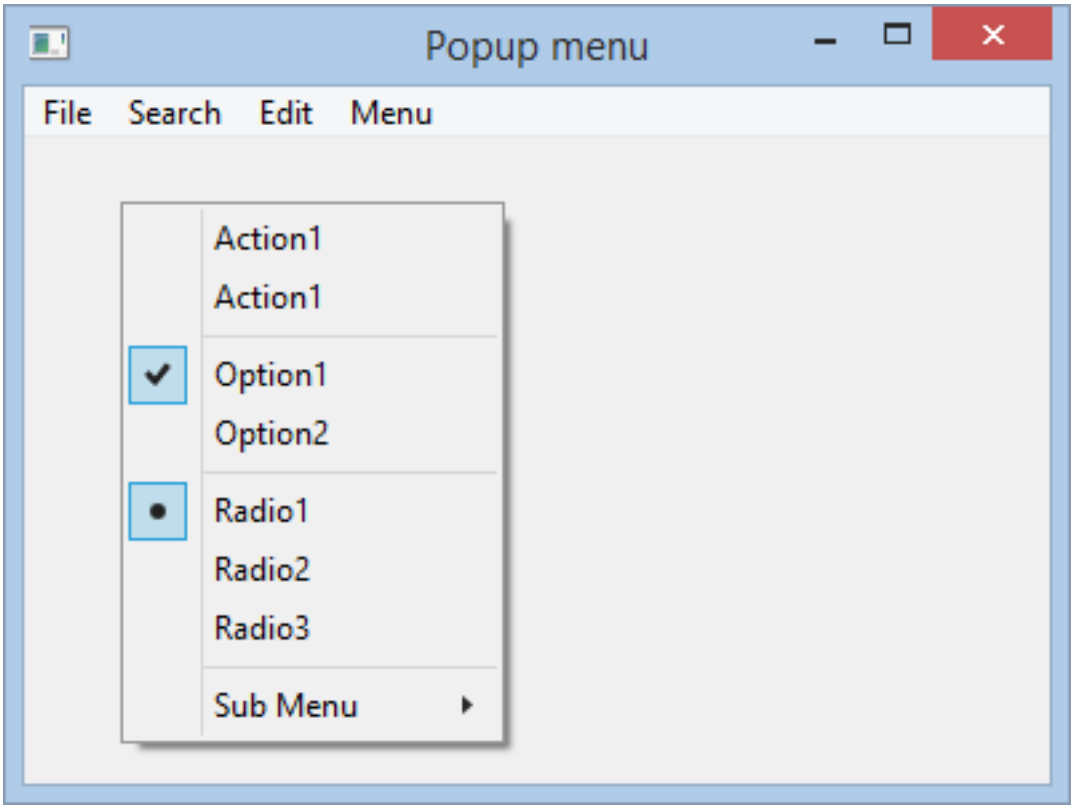


Abbildung 3.15: Popup Menu Beispiel

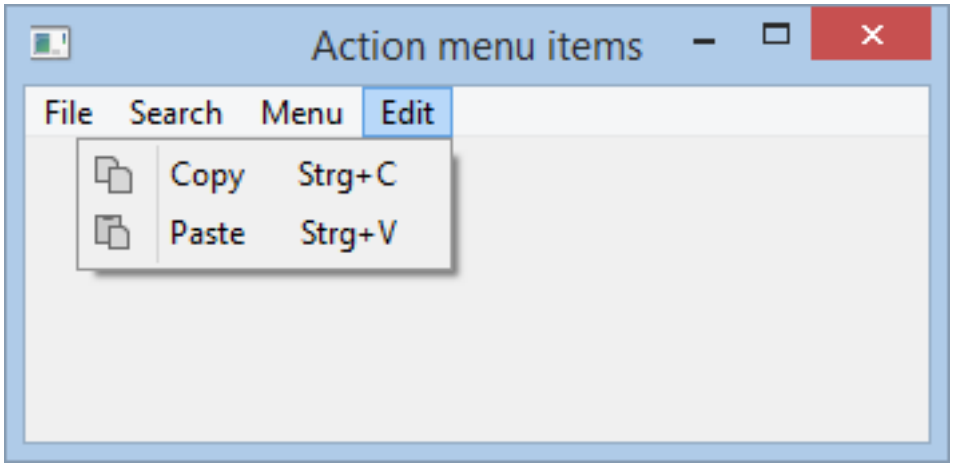


Abbildung 3.16: Action Menu Item Beispiel

3.10.7.1 Menu Model

Mit Hilfe der folgenden Methoden kann das Model gesetzt und ausgelesen werden. Siehe auch Menü und Item Models.

```
void setModel(IActionItemModel model);

IActionItemModel getModel();
```

3.10.7.2 Action Events

Mit Hilfe der folgenden Methoden kann man sich als Listener registrieren, um Action Events zu erhalten:

```
void addActionListener(IActionListener actionListener);

void removeActionListener(IActionListener actionListener);
```

Ein `IActionListener` hat die folgende Methode:

```
void actionPerformed();
```

Diese wird aufgerufen, wenn das Item die Aktion ausführen soll, zum Beispiel durch *Anklicken* oder die Verwendung eines Tastaturkürzels. Anmerkung: Dies wird als Action Event bezeichnet, auch wenn der `IActionListener` kein explizites Event Objekt übergeben bekommt.

3.10.7.3 Action Binding

Mit der folgenden Methode kann eine `IAction` an das Item gebunden werden:

```
void setAction(IAction action);
```

Weitere Details finden sich im Abschnitt Actions und Commands

3.10.7.4 Tastaturkürzel

Mit Hilfe der folgenden Methode kann das Tastaturkürzel (Key Accelerator) festgelegt werden, mit welchem die Aktion (ohne Maus) ausgeführt werden soll:

```
void setAccelerator(Accelerator accelerator);
```

Ist das zugehörige Item in einem Main Menu eines aktiven Frames, wird das Tastaturkürzel automatisch ausgewertet und ein Action Event gefeuert, wenn die entsprechende Tastenkombination gedrückt wird.

Bei Action Items in Popup Menüs ist das **nicht der Fall**. Dort muss man sich, z.B. mit Hilfe eines KeyListener, selbst um das Auslösen der Aktionen kümmern. Die Utility Klasse MenuModelKeyBinding kann dabei unterstützen.

3.10.7.5 Action Item BluePrint

Ein Action Item kann (u.A.) mit Hilfe eines `IActionMenuItemBluePrint` erzeugt werden. Die Klasse `BPF` liefert die folgenden Methoden für die Erzeugung eines BluePrint:

```
public static IActionMenuItemBluePrint menuItem() {...}

public static IActionMenuItemBluePrint menuItem(final String text) {...}
```

Die zweite Methode ermöglicht das gleichzeitige setzen des Label Textes auf dem BluePrint bei der Erzeugung.

Ein `IActionMenuItemBluePrint` hat die folgenden Methoden zur Konfiguration:

```
IActionMenuItemBluePrint setText(String text);

IActionMenuItemBluePrint setToolTipText(String toolTipText);

IActionMenuItemBluePrint setIcon(IImageConstant icon);

IActionMenuItemBluePrint setMnemonic(Character mnemonic);

IActionMenuItemBluePrint setAccelerator(Accelerator accelerator);
```

Diese können analog zu den Methoden der Schnittstelle `IActionMenuItem` verwendet werden.

Das folgende Beispiel demonstriert die Verwendung von Action Menu Items:

```
1  final IActionMenuItemBluePrint copyItemBp = BPF.menuItem();
2  copyItemBp
3      .setText("Copy")
4      .setToolTipText("Copies the selection into the clipboard")
5      .setIcon(IconsSmall.COPY)
6      .setAccelerator(new Accelerator(VirtualKey.C, Modifier.CTRL));
7
8  final IActionMenuItem copyItem = editMenu.addItem(copyItemBp);
9  copyItem.addActionListener(new IActionListener() {
10     @Override
11     public void actionPerformed() {
12         System.out.println("Perform copy");
13     }
14 });
15
16 final IActionMenuItemBluePrint pasteItemBp = BPF.menuItem();
17 pasteItemBp
18     .setText("Paste")
19     .setToolTipText("Pastes the clipboard into the selection")
20     .setIcon(IconsSmall.PASTE)
21     .setAccelerator(new Accelerator(VirtualKey.V, Modifier.CTRL));
22
23 final IActionMenuItem pasteItem = editMenu.addItem(pasteItemBp);
24 pasteItem.addActionListener(new IActionListener() {
25     @Override
26     public void actionPerformed() {
27         System.out.println("Perform paste");
28     }
29 });
```

Bemerkung: Für größeren Anwendungen wird anstatt der obigen Vorgehensweise die Verwendung von Actions empfohlen.

3.10.8 Die Schnittstelle ISelectableMenuItem

Die Schnittstelle `ISelectableMenuItem` liefert die Funktionen für das Checked Menu Item und Radio Menu Item. Ein `ISelectableMenuItem` ist von `IMenuItem` und somit auch von `IItem` und `IWidget` abgeleitet.

Es folgt eine kurze Beschreibung der wichtigsten Methoden:

3.10.8.1 Menu Model

Mit Hilfe der folgenden Methoden kann das Model gesetzt und ausgelesen werden. Siehe auch Menü und Item Models.

```
ISelectableMenuItemModel getModel();

void setModel(ISelectableMenuItemModel model);
```

3.10.8.2 Selected State

Der `selected` State gibt an, ob das Item ausgewählt ist, oder nicht. Bei einem Checked Menu Item ist das der Fall, wenn die Checkbox *angehackt* ist. Bei einem Radio Menu Item, wenn der Radio Button gedrückt ist. Mit Hilfe der folgenden Methoden kann der `selected` State gesetzt und ausgelesen werden:

```
boolean isSelected();

void setSelected(boolean selected);
```

Um sich über Änderungen des `selected` State informieren zu lassen, kann ein `IItemStateListener` verwendet werden:

```
void addItemListener(final IItemStateListener listener);

void removeItemListener(final IItemStateListener listener);
```

Dieser hat die folgende Methode:

```
void itemStateChanged();
```

3.10.8.3 Tastaturkürzel

Mit Hilfe der folgenden Methode kann das Tastaturkürzel (Key Accelerator) festgelegt werden:

```
void setAccelerator(Accelerator accelerator);
```

Ist das zugehörige Item in einem Main Menu eines aktiven Frames, wird das Tastaturkürzel automatisch ausgewertet und ein Action Event gefeuert, wenn die entsprechende Tastenkombination gedrückt wird.

Bei Items in Popup Menüs ist das **nicht der Fall**. Dort muss man sich, z.B. mit Hilfe eines KeyListener, selbst um das Auslösen der Aktionen kümmern.

Bei einem Checked Menu Item wird durch das Tastaturkürzel der Selected State umgeschalten (toggle). Bei einem Radio Menu Item wird durch das Tastaturkürzel der `selected` State auf `true` gesetzt, falls er `false` ist. Ist das Item bereits selektiert, wird der Tastaturkürzel ignoriert.

3.10.9 Checked Menu Item

Ein Checked Menu Item ist ein Optionsfeld (Checkbox) innerhalb eines Menüs. Im Vergleich zu einem Radio Menu Item handelt es sich dabei um eine Einzeloption. Die folgende Abbildung zeigt ein Menu mit zwei Checked Menu Items, wobei die erste Option ausgewählt (`selected`) ist. Beide Optionen können mit Hilfe eines Tastaturkürzel umgeschalten werden.

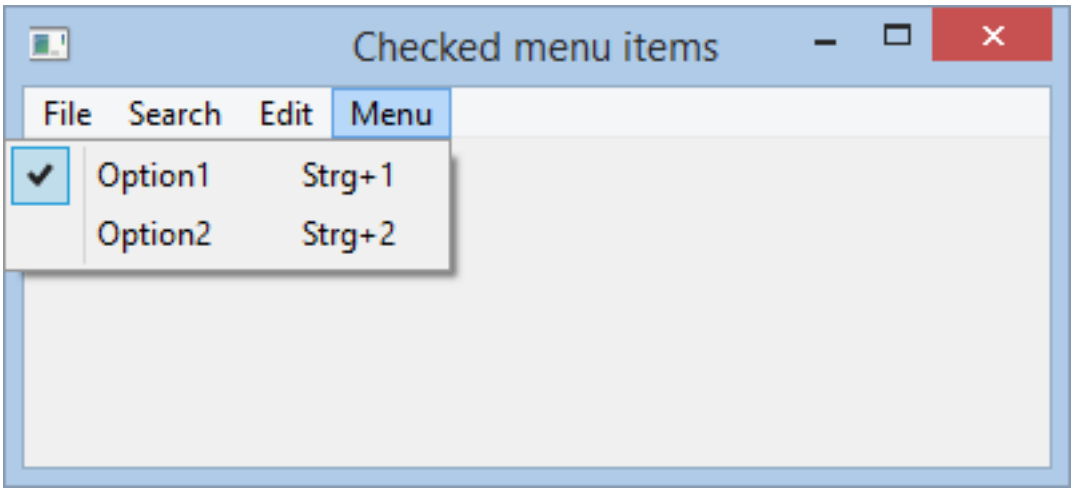


Abbildung 3.17: Checked Menu Item Beispiel

Eine Checked Menu Item implementiert die Schnittstelle ISelectableMenuItem und liefert keine zusätzlichen Methoden.

3.10.9.1 Checked Menu Item BluePrint

Ein Checked Menu Item kann (u.A.) mit Hilfe eines `ICheckedMenuItemBluePrint` erzeugt werden. Die Klasse `BPF` liefert die folgenden Methoden für die Erzeugung eines BluePrint:

```
public static ICheckedMenuItemBluePrint checkedMenuItem() {...}

public static ICheckedMenuItemBluePrint checkedMenuItem(final String text) {...}
```

Die zweite Methode ermöglicht das gleichzeitige setzen des Label Textes auf dem BluePrint bei der Erzeugung.

Ein `ICheckedMenuItemBluePrint` hat die folgenden Methoden zur Konfiguration:

```
ICheckedMenuItemBluePrint setText(String text);

ICheckedMenuItemBluePrint setToolTipText(String toolTipText);

ICheckedMenuItemBluePrint setMnemonic(Character mnemonic);

ICheckedMenuItemBluePrint setAccelerator(Accelerator accelerator);

ICheckedMenuItemBluePrint setSelected(boolean selected);
```

Diese können analog zu den Methoden der Schnittstelle `ISelectableMenuItem` verwendet werden.

3.10.9.2 Beispiel

Das folgende Beispiel demonstriert die Verwendung

```
    //first option
    final ICheckedMenuItemBluePrint option1Bp = BPF.checkedMenuItem();
    option1Bp
        .setText("Option1")
        .setToolTipText("Check this if option1 is desired")
        .setAccelerator(new Accelerator(VirtualKey.DIGIT_1, Modifier.CTRL))
        .setSelected(true);

    final ISelectableMenuItem option1 = menu.addItem(option1Bp);
    option1.addItemListener(new IItemStateListener() {
        @Override
        public void itemStateChanged() {
            System.out.println("Option1 changed: " + option1.isSelected());
        }
    });

    //second option
    final ICheckedMenuItemBluePrint option2Bp = BPF.checkedMenuItem();
    option2Bp
        .setText("Option2")
        .setToolTipText("Check this if option2 is desired")
        .setAccelerator(new Accelerator(VirtualKey.DIGIT_2, Modifier.CTRL));

    final ISelectableMenuItem option2 = menu.addItem(option2Bp);
    option2.addItemListener(new IItemStateListener() {
        @Override
        public void itemStateChanged() {
            System.out.println("Option2 changed: " + option2.isSelected());
        }
    });
```

3.10.10 Radio Menu Item

Mit Hilfe eines Radio Menu Item kann eine Auswahl innerhalb einer Radio Item Group getroffen werden. Innerhalb einer Radion Item Group ist maximal ein Radio Item gleichzeitig ausgewählt.

Alle direkt benachbarten Radion Menu Items eines Menüs gehören zu einer Gruppe. Mit Hilfe von Separator Menu Items können somit innerhalb eines Menüs mehrere Gruppen umgesetzt werden. Ein explizites setzen von Gruppen ist **nicht** möglich [6]! Die folgende Abbildung zeigt ein Menu mit drei Radio Menu Item Gruppen, wobei jeweils eine Option ausgewählt (`selected`) ist.

Eine Radio Menu Item implementiert die Schnittstelle ISelectableMenuItem und liefert keine zusätzlichen Methoden.

3.10.10.1 Radio Menu Item BluePrint

Ein Radio Menu Item kann (u.A.) mit Hilfe eines `IRadioMenuItemBluePrint` erzeugt werden. Die Klasse `BPF` liefert die folgenden Methoden für die Erzeugung eines BluePrint:

[6] Dies war eine bewußte Design Entscheidung. Radio Gruppen zu definieren, welche der Nutzer nicht als Gruppe wahrnimmt, weil die Items nicht zusammenhängend dargestellt werden, hat keine praktische Relevanz. Demgegenüber sollte für den Standardfall kein zusätzlicher Implementierungsaufwand durch die Zuweisung zu Gruppen notwendig sein.

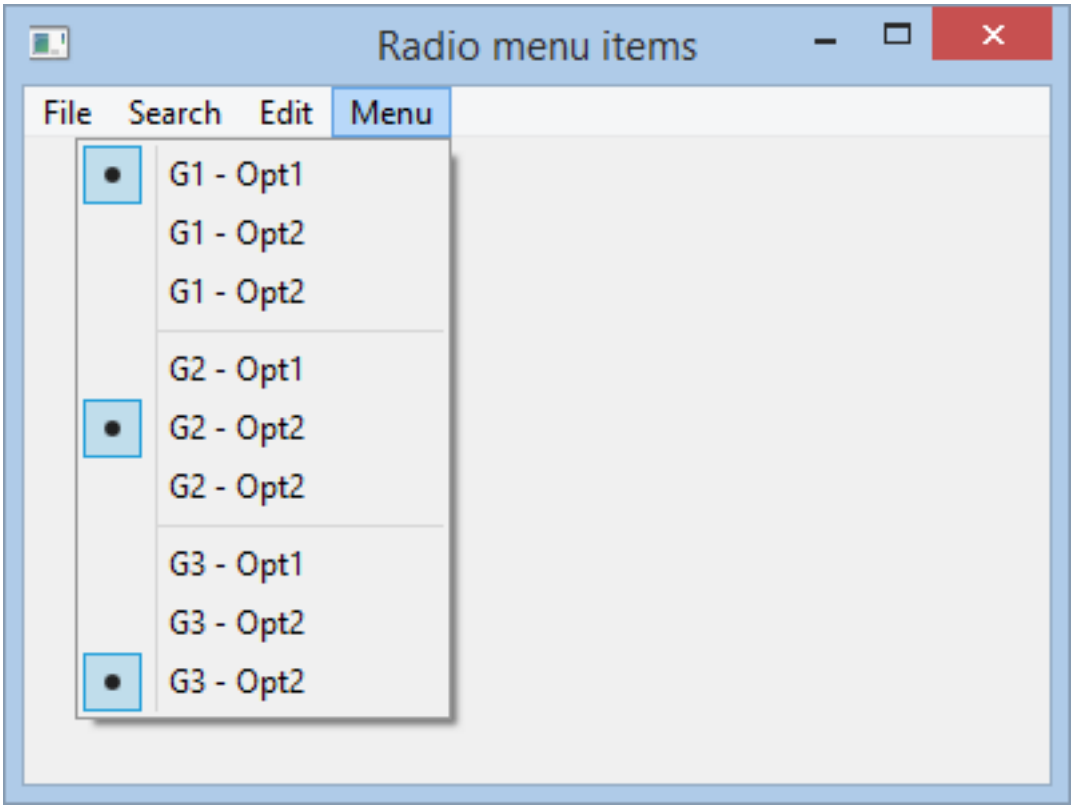


Abbildung 3.18: Radio Menu Item Beispiel

```
public static IRadioMenuItemBluePrint checkedMenuItem() {...}

public static IRadioMenuItemBluePrint checkedMenuItem(final String text) {...}
```

Die zweite Methode ermöglicht das gleichzeitige setzen des Label Textes auf dem BluePrint bei der Erzeugung.

Ein `IRadioMenuItemBluePrint` hat die folgenden Methoden zur Konfiguration:

```
IRadioMenuItemBluePrint setText(String text);

IRadioMenuItemBluePrint setToolTipText(String toolTipText);

IRadioMenuItemBluePrint setMnemonic(Character mnemonic);

IRadioMenuItemBluePrint setAccelerator(Accelerator accelerator);

IRadioMenuItemBluePrint setSelected(boolean selected);
```

Diese können analog zu den Methoden der Schnittstelle `ISelectableMenuItem` verwendet werden.

3.10.10.2 Beispiel

Das folgende Beispiel demonstriert die Verwendung:

```
1  //radio group1
2  ISelectableMenuItem g1Opt1 = menu.addItem(BPF.radioMenuItem("G1 - Opt1").setSelected(true));
3  ISelectableMenuItem g1Opt2 = menu.addItem(BPF.radioMenuItem("G1 - Opt2"));
4  ISelectableMenuItem g1Opt3 = menu.addItem(BPF.radioMenuItem("G1 - Opt2"));
5
6  //radio group2
7  menu.addSeparator();
8  ISelectableMenuItem g2Opt1 = menu.addItem(BPF.radioMenuItem("G2 - Opt1"));
9  ISelectableMenuItem g2Opt2 = menu.addItem(BPF.radioMenuItem("G2 - Opt2").setSelected(true));
10 ISelectableMenuItem g2Opt3 = menu.addItem(BPF.radioMenuItem("G2 - Opt2"));
11
12 //radio group3
13 menu.addSeparator();
14 ISelectableMenuItem g3Opt1 = menu.addItem(BPF.radioMenuItem("G3 - Opt1"));
15 ISelectableMenuItem g3Opt2 = menu.addItem(BPF.radioMenuItem("G3 - Opt2"));
16 ISelectableMenuItem g3Opt3 = menu.addItem(BPF.radioMenuItem("G3 - Opt2").setSelected(true));
```

3.10.11 Separator Menu Item

Ein Menu Separator kann zur visuellen Gruppierung von Items innerhalb eines Menüs verwendet werden.

3.10.11.1 Separator Menu Item BluePrint

Ein Separator Menu Item kann (u.A.) mit Hilfe eines `ISeparatorMenuItemBluePrint` erzeugt werden. Die Klasse `BPF` liefert die folgenden Methoden für die Erzeugung eines BluePrint:

```
public static ISeparatorMenuItemBluePrint menuSeparator() {...}
```

3.10.11.2 Beispiel

Das folgende Beispiel demonstriert die Verwendung:

```
1  menu.addItem(BPF.menuItem("Item1"));
2  menu.addItem(BPF.menuItem("Item2"));
3  menu.addItem(BPF.menuItem("Item3"));
4
5  menu.addItem(BPF.menuSeparator());
6  menu.addItem(BPF.menuItem("Item4"));
7  menu.addItem(BPF.menuItem("Item5"));
8
9  menu.addSeparator();
10 menu.addItem(BPF.menuItem("Item6"));
11 menu.addItem(BPF.menuItem("Item7"));
12 menu.addItem(BPF.menuItem("Item8"));
13 menu.addItem(BPF.menuItem("Item9"));
```

In Zeile 5 wird der Separator mit Hilfe eines BluePrint erzeugt, in Zeile 9 mit Hilfe der Convenience Methode `addSeparator()` der Schnittstelle `IMenu`.

Die folgende Abbildung zeigt das Ergebnis:

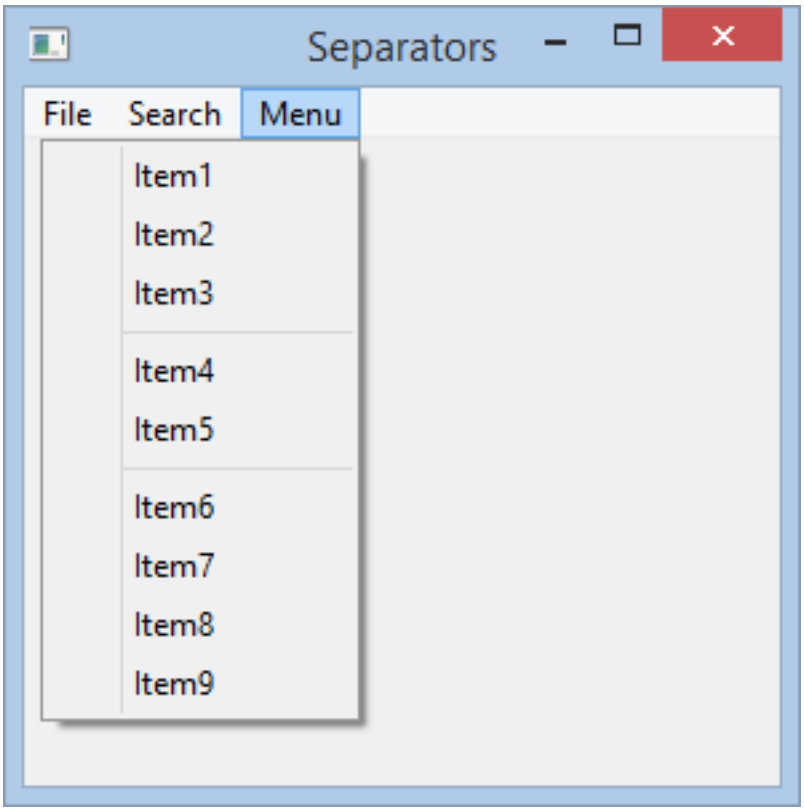


Abbildung 3.19: Menu Separator Item Beispiel

3.11 Menü und Item Models

Im Abschnitt Menus und Items wurde die native Verwendung von Menus und Menu Items in jowidgets beschrieben. Dort existiert eine Menü oder Menü Item Instanz genau ein Mal an einer Stelle. Eine Wiederverwendung auf Instanzebene ist **nicht** möglich[7].

Auf Klassenebene gibt es bestimmte Einschränkungen bei der Wiederverwendung. Angenommen man möchte ein Menü einer Menu Bar oder Teile davon, wie zum Beispiel ein Checked Menu Item, noch mal an einer anderen Stelle, zum Beispiel in einen Popup Menu haben, kommt man nicht umher, das gleiche Menü oder Item ein zweites mal zu erzeugen. Will man das Menü nachträglich anpassen, zum Beispiel weil ein Plugin eine *Menu Contribution* macht, muss man beide Instanzen nachträglich anpassen. Für das Checked Menu Item müssten beide Instanzen, z.B. über ein Presentation Model, aneinander gebunden werden.

Menu und Items Models wurden entworfen, um Menüs und Menü Items zu beschreiben bzw. deren aktuellen Zustand widerzuspiegeln. Ein solches Model (eine Instanz) kann dann an (mehrere) konkrete Items gebunden werden. Änderungen auf dem Model, wie zum Beispiel das Einfügen neuer Menü Items oder das Ändern des `selected` State eines Checked Menu Item, werden dann an allen Stellen synchron gehalten.

Item Models haben eine eindeutige ID. Dadurch ist es zum Beispiel möglich, Menu Contributions wie `addBefore(item, idWhereToAdd)` oder `addAfter(item, idWhereToAdd)` umzusetzen, ohne eine Referenz auf das Item (*where to add*) haben zu müssen [8].

Item Models haben einen `visible` Status. Dadurch ist es zum Beispiel möglich, einzelne Aktionen auszublenden, ohne dass das Item Model aus seinem Menü entfernt werden muss (wodurch sich die Struktur ändern würde, was im Zusammenhang mit Menü Contributions wieder zu Problemen führen könnte). Man kann für Action Items Sichbarkeitsaspekte injizieren, was u.A. im Zusammenhang mit Rechtemanagement hilfreich ist.

Item Models beschränken sich nicht ausschließlich auf Menüs. So ist es für bestimmte Items möglich, diese gleichzeitig in einer Toolbar und in einem Menu zu haben. Beispielsweise wird ein Checked Item Model in einem Menu als Checked Menu Item und in einer Toolbar als ToolBarToggleButton dargestellt. Ein Action Item Model wird in einer Toolbar mit Hilfe eines ToolBarButton und in einem Menu als Action Menu Item angezeigt. Ein Menu Model kann sowohl für ein Main Menu, ein Sub Menu als auch für ein Toolbar Menu verwendet werden.

3.11.1 Einführendes Beispiel

Die Verwendung von Item Models soll vorab anhand eines Beispiels verdeutlicht werden (das komplette `ItemModelSnipped` findet sich hier).

Das Rahmenprogramm sieht wie folgt aus:

```
1  public final class ItemModelSnipped implements IApplication {
2
3      @Override
4      public void start(final IApplicationLifecycle lifecycle) {
5
6          //create a root frame
7          final IFrameBluePrint frameBp = BPF.frame();
8          frameBp.setSize(new Dimension(400, 300)).setTitle("Menu and Item Models");
9          final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);
```

[7] In Swt und Swing ist das auch nicht möglich!

[8] Der Label Text sollte dafür auf keinen Fall verwendet werden da dieser internationalisiert sein kann

```
10
11          //Create the menu bar
12          final IMenuBarModel menuBar = frame.getMenuBarModel();
13
14          //Use a border layout, add toolbar and composite
15          frame.setLayout(BorderLayout.builder().gap(0).build());
16          final IToolBarModel toolBar = frame.add(BPF.toolBar(), BorderLayout.TOP).getModel();
17          final IComposite composite = frame.add(BPF.composite().setBorder(), BorderLayout.CENTER);
18
19          //set the root frame visible
20          frame.setVisible(true);
21      }
22  }
```

Es wird ein Frame verwendet, welches mittels eines Border Layout eine Toolbar und ein Composite darstellt. Zudem wird eine Menu Bar erzeugt.

Der folgende Code beinhaltet die Erzeugung der Items und Menüs:

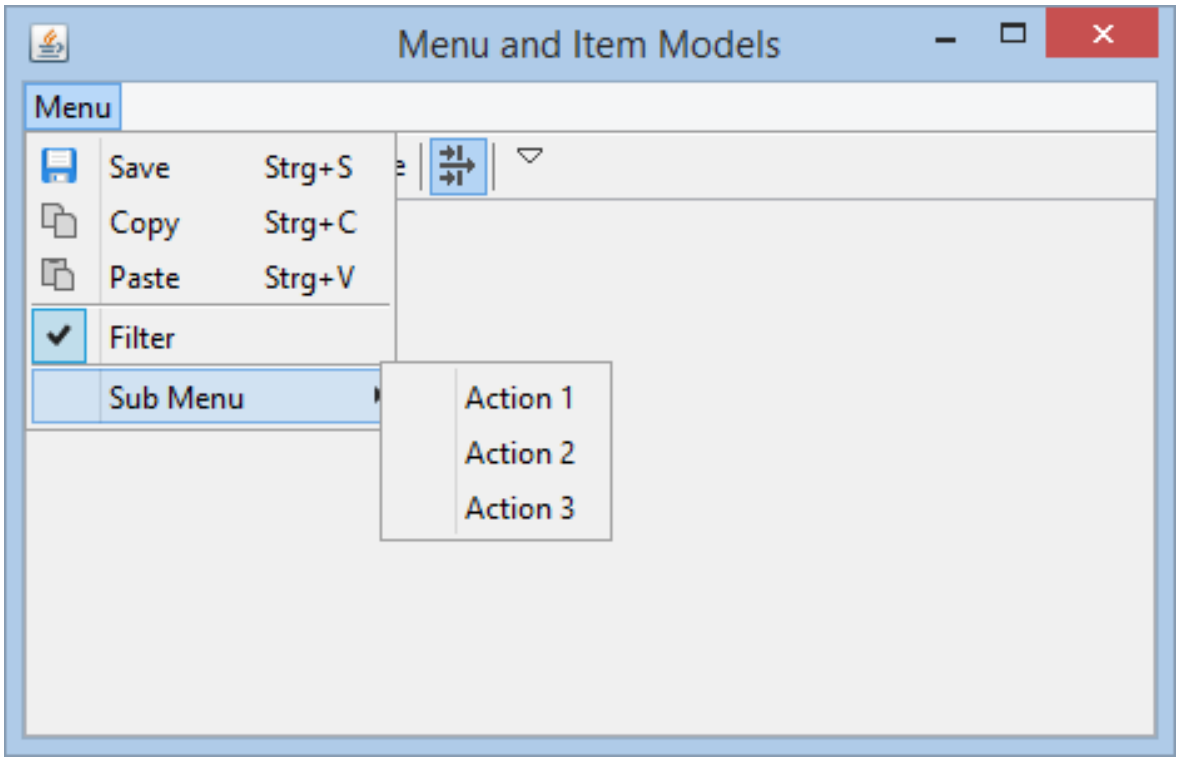
```
1       //create a checked item for filter
2       final CheckedItemModel filter = new CheckedItemModel("Filter", IconsSmall.FILTER);
3       filter.setSelected(true);
4
5       //create save action (with constructor)
6       final ActionItemModel save = new ActionItemModel("Save", IconsSmall.DISK);
7       save.setAccelerator(VirtualKey.S, Modifier.CTRL);
8
9       //create copy action (with constructor)
10      final ActionItemModel copy = new ActionItemModel("Copy", IconsSmall.COPY);
11      copy.setAccelerator(VirtualKey.C, Modifier.CTRL);
12
13      //create paste action (with builder)
14      final IActionItemModel paste
15          = ActionItemModel
16              .builder()
17              .setText("Paste")
18              .setIcon(IconsSmall.PASTE)
19              .setAccelerator(VirtualKey.V,Modifier.CTRL)
20              .build();
21
22      //create a menu and add items
23      final MenuModel menu = new MenuModel("Menu");
24      menu.addItem(save);
25      menu.addItem(copy);
26      menu.addItem(paste);
27      menu.addSeparator();
28      menu.addItem(filter);
29      menu.addSeparator();
30
31      //create a sub menu and add some items
32      final IMenuModel subMenu = menu.addMenu("Sub Menu");
33      final IActionItemModel action1 = subMenu.addActionItem("Action 1");
34      final IActionItemModel action2 = subMenu.addActionItem("Action 2");
35      final IActionItemModel action3 = subMenu.addActionItem("Action 3");
36
37      //add the menu to the menu bar
38      menuBar.addMenu(menu);
39
40      //sets the menu as popup menu on the composite
41      composite.setPopupMenu(menu);
42
43      //add some actions and items to the toolbar
44      toolBar.addItem(save);
```

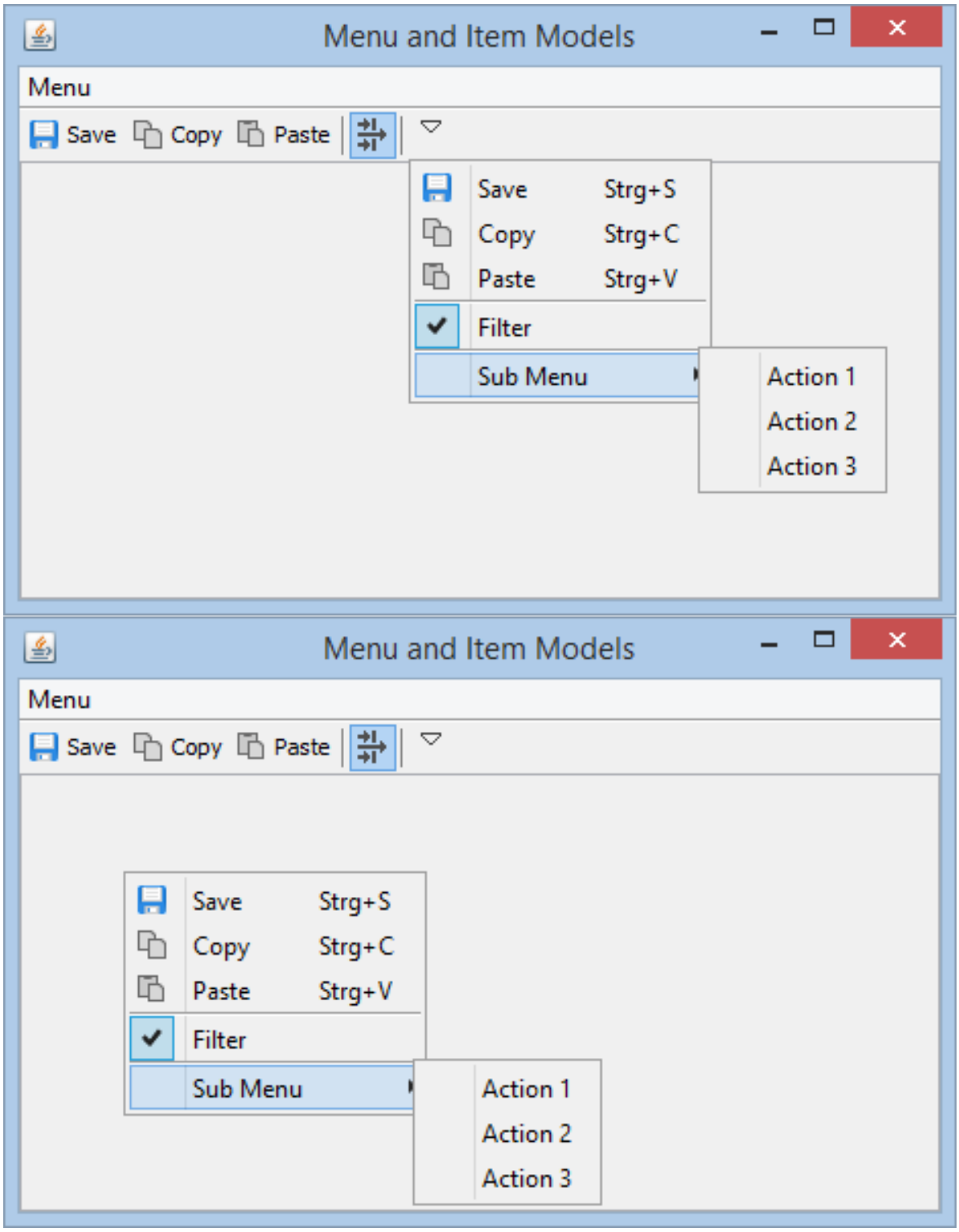
```
45      toolBar.addItem(copy);
46      toolBar.addItem(paste);
47      toolBar.addSeparator();
48      toolBar.addItem(filter);
49      toolBar.addSeparator();
50
51      //add menu to the toolbar
52      toolBar.addItem(menu);
```

Anschließend werden zu einigen Items Listener hinzugefügt, welche deren Funktion mit Hilfe einer Konsolenausgabe belegen:

```
1       //add listeners to the items
2       filter.addItemListener(new SysoutSelectionListener(filter));
3       save.addActionListener(new SysoutActionListener(save));
4       copy.addActionListener(new SysoutActionListener(copy));
5       paste.addActionListener(new SysoutActionListener(paste));
6       action1.addActionListener(new SysoutActionListener(action1));
7       action2.addActionListener(new SysoutActionListener(action2));
8       action3.addActionListener(new SysoutActionListener(action3));
```

Die folgenden Abbildungen zeigen das Ergebnis, jeweils mit geöffnetem Main Menu, ToolBar Menu und Popup Menu:





Deaktiviert man in der Toolbar nun zum Beispiel den *Filter* Toggle Button, wird auch das entsprechende Checked Menu Item in den drei vorhandenen Menüs deaktiviert und auf dem `filter` Item ein Selection Event *gefeuert*.

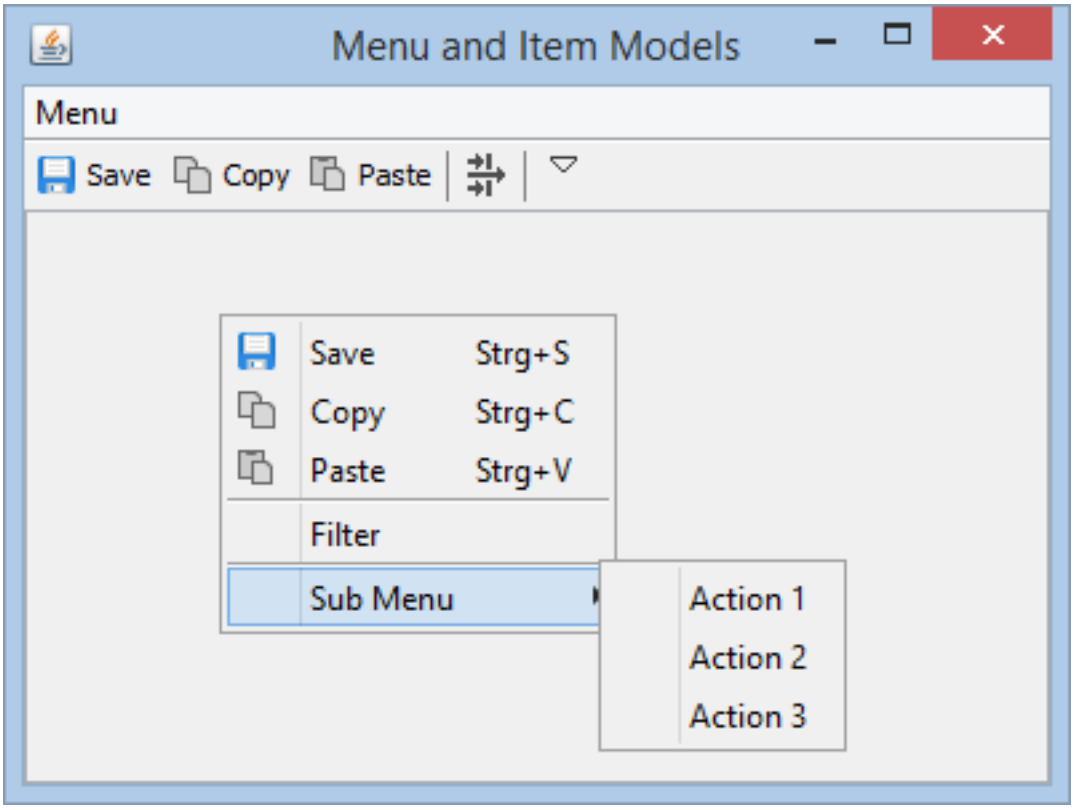


Abbildung 3.20: ItemModelSnipped - Checked Item Binding

Im folgenden wird nun *Action2* ausgeblendet:

```
action2.setVisible(false);
```

Die Abbildung zeigt das Ergebnis:

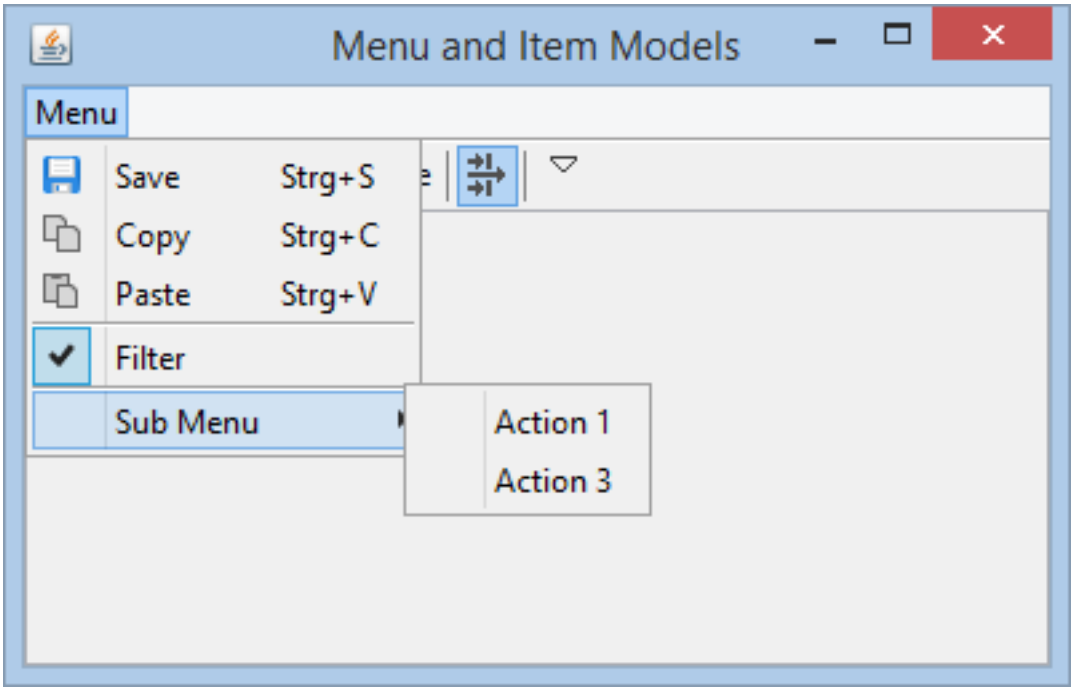


Abbildung 3.21: ItemModelSnipped - Sichtbarkeit

Action2 wird in den anderen Menüs selbstverständlich auch nicht mehr angezeigt. Mit Hilfe von `setVisible(false)` wird der gleiche visuelle Effekt erzielt, als hätte man:

```
subMenu.removeItem(action2);
```

ausgeführt. Allerdings müsste man hier zum wieder sichtbar machen das Folgende ausführen:

```
subMenu.addItem(1, action2);
```

was bedeutet, dass man den Index kennen muss, wo das Item eingefügt werden soll, während man im ersten Fall mit

```
action2.setVisible(true);
```

auskommt.

Anmerkung: An welchem Index man eine Action wieder sichtbar machen müsste, ist bei mehr als einem ausgeblendeten Item nicht mehr trivial, und ohne genaue Kenntnis aller Menüeinträge und deren Sichtbarkeit nicht lösbar.

3.11.2 Menu Bar Model

Ein Menu Bar Model ist ein Model für eine Menu Bar. Es folgt eine Beschreibung der wichtigsten Methoden der Schnittstelle `IMenuBarModel`:

3.11.2.1 Hinzufügen von existierenden Menu Models

Mit Hilfe der folgenden Methoden können bereits existierende Menu Models hinzugefügt werden:

```
void addMenu(IMenuModel menu);

void addMenu(int index, IMenuModel menu);
```

3.11.2.2 Hinzufügen von Menu Models mit Hilfe von Menu Model Buildern

Mit den folgenden Methoden lassen sich Menus mit Hilfe von Menu Model Buildern hinzufügen:

```
IMenuModel addMenu(IMenuModelBuilder menuBuilder);

IMenuModel addMenu(int index, IMenuModelBuilder menuBuilder);
```

Dabei wird beim Hinzufügen die `build()` Methode aufgerufen, das gebaute Menu Model hinzugefügt und zurückgegeben. Damit können zum Beispiel Konstrukte der folgenden Art realisiert werden:

```
final IMenuModel menu1 = menuBar.addMenu(
        MenuModel
            .builder()
            .setId(MENU_1_ID)
            .setText("Menu1")
            .setMnemonic('1'));

final IMenuModel menu2 = menuBar.addMenu(
        MenuModel
            .builder()
            .setId(MENU_2_ID)
            .setText("Menu2")
            .setMnemonic('2'));
```

3.11.2.3 Erstellen und Hinzufügen von Menüs mit einem Aufruf

Die folgenden Methoden erlauben das erstellen und Hinzufügen von Menüs mit einem Aufruf:

```
IMenuModel addMenu();

IMenuModel addMenu(String text);
```

Dabei wird ein neues Menu Model erzeugt, hinzugefügt und zurückgegeben. Das folgende Beispiel demonstriert die Verwendung:

```
final IMenuModel menu1 = menuBar.addMenu("Menu1");
final IMenuModel menu2 = menuBar.addMenu("Menu2");
final IMenuModel menu3 = menuBar.addMenu("Menu3");
```

3.11.2.4 Hinzufügen von Menüs relativ zu anderen Menüs

Die folgenden Methoden können verwendet werden, um ein Item an eine bestimmte Stelle in einer Menu Bar hinzuzufügen:

```
    void addBefore(IMenuModel newMenu, String id);

    void addAfter(IMenuModel newMenu, String id);
```

Der `id` gibt dabei das Menu an, bezüglich welcher das neue Menü hinzugefügt werden soll. Ist dieses nicht vorhanden, wird eine `IllegalArgumentException` geworfen. Mittels der Methode `findMenuById(String id)` kann vorab überprüft werden, ob ein solches Menü vorhanden ist.

Die Verwendung soll anhand eines Beispiels demonstriert werden. Der folgende Code erzeugt eine Menu Bar mit drei Menüs wobei das letzte ein Hilfe Menu mit fester id ist:

```
1   final IMenuModel menu1 = menuBar.addMenu("Menu1");
2   final IMenuModel menu2 = menuBar.addMenu("Menu2");
3
4   IMenuModel helpMenu = menuBar.addMenu(
5           MenuModel
6               .builder()
7               .setId(HELP_MENU_ID)
8               .setText("Help"));
```

An einer anderen Stelle könnte man eine Contribution zur Menu Bar wie folgt machen:

```
1   final IMenuModel customMenu = new MenuModel("Custom menu");
2   menuBar.addBefore(customMenu, HELP_MENU_ID);
```

Die folgende Abbildung zeigt das Ergebnis:

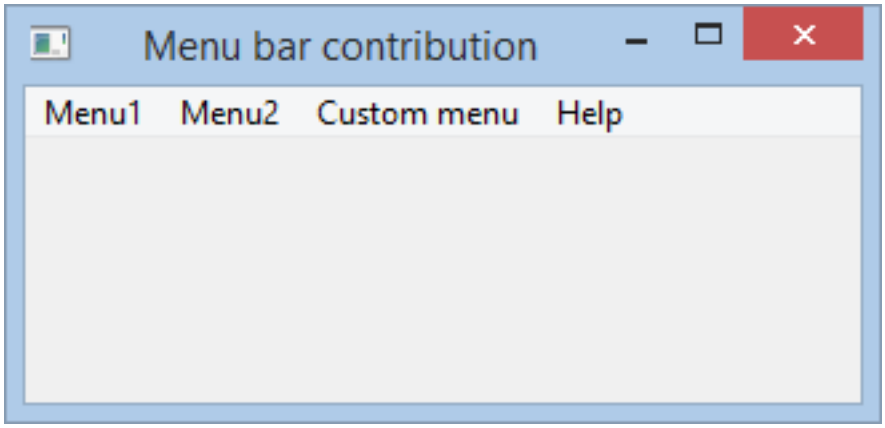


Abbildung 3.22: Menu Bar Contribution Beispiel

3.11.2.5 Hinzufügen der Menüs einer anderen Menu Bar

Mit Hilfe der folgenden Methode werden alle Menüs der übergebenen Menu Bar hinzugefügt:

```
void addMenusOfModel(IMenuBarModel model);
```

Dabei wird eine Referenz der Items hinzugefügt und keine Kopie!

3.11.2.6 Entfernen von Menüs

Mit Hilfe der folgenden Methoden können Menüs aus einer Menu Bar entfernt werden:

```
void removeMenu(final IMenuModel item);

void removeMenu(int index);

void removeAllMenus();
```

3.11.2.7 ListModelListener

Um sich über das Hinzufügen oder Entfernen von Menüs informieren zu lassen, kann ein `IListModelListener` verwendet werden:

```
void addListModelListener(IListModelListener listener);

void removeListModelListener(IListModelListener listener);
```

Dieser hat die folgenden Methoden:

```
void afterChildAdded(int index);

void beforeChildRemove(int index);

void afterChildRemoved(int index);
```

3.11.2.8 Zugriff auf die Menüs einer Menu Bar

Die folgende Methode liefert eine nicht modifizierbare Kopie der aktuell vorhandenen Menüs der Menu Bar.

```
List<IMenuModel> getMenus();
```

Um ein Menu Anhand einer id zu finden, kann die folgende Methode verwendet werden:

```
IMenuModel findMenuById(String id);
```

Existiert kein solches Menü in der Menu Bar, wird null zurückgegeben.

3.11.2.9 Kopieren einer Menu Bar

Mit Hilfe der folgenden Methode kann eine Kopie (*Deep Copy*) einer Menu Bar erzeugt werden. Die Kopie hat dabei die *gleichen* Menüs wie das Original (jedoch nicht die *Selben*). Registrierte Listener werden nicht mitkopiert. Dadurch sind bereits gebundene Items nicht an die Kopie gebunden.

```
IMenuBarModel createCopy();
```

3.11.2.10 Menu Bar Model Instanzen

Die Klasse `org.jowidgets.tools.model.item.MenuBarModel` liefert eine Implementierung der `IMenuBarModel` Schnittstelle. Eine neu Instanz kann entweder mittels `new` wie folgt erzeugt werden:

```
IMenuBarModel menuBar = new MenuBarModel();
```

oder man verwendet die statische `create()` Methode:

```
IMenuBarModel menuBar = MenuBarModel.create()
```

Man kann eine Instanz aber auch direkt von einem Frame mittels

```
IMenuBarModel getMenuBarModel();
```

oder von einer Menu Bar mittels

```
IMenuBarModel getModel();
```

erhalten.

Um ein existierendes Menu Bar Model auf einem Frame zu setzen, kann die folgende Methode verwendet werden:

```
void setMenuBar(IMenuBarModel model);
```

3.11.3 Die Schnittstelle IItemModel

Die Schnittstelle IItemModel liefert die Basisfunktionen für alle Item Models. Es werden nicht für jeden Item Typ alle Attribute ausgewertet, bzw. an ein konkretes Item gebunden. So kann zum Beispiel der `enabled` State nur für Action Items gesetzt werden, weil andere Items ein `disable` nicht ermöglichen. Es folgt eine kurze Beschreibung der wichtigsten Methoden:

3.11.3.1 Die Item ID

Mit Hilfe der folgenden Methode erhält man die id eines Items:

```
String getId();
```

Die id darf weder `null` noch ein Leerstring sein. Die id sollte möglichst global eindeutig sein. Wird keine id definiert, wird per default eine UUID verwendet. Die Item id wird für die Implementierung von `equals()` und `hashCode()` herangezogen. Zwei Items sind genau dann gleich, wenn ihre id's gleich sind. Es ist nicht möglich, zwei Items mit der gleichen id zu einem Menü hinzuzufügen.

3.11.3.2 Label, Tooltip und Icon

Mit Hilfe der folgenden Methoden können der Label Text, das Tooltip und das Icon gesetzt und ausgelesen werden:

```
void setText(final String text);

String getText();

void setToolTipText(String toolTipText);

String getToolTipText();

void setIcon(IImageConstant icon);

IImageConstant getIcon();
```

3.11.3.3 Tastatursteuerung

Mit den folgenden Methoden erhält man Zugriff auf Mnemonic und Key Accelerator:

```
void setAccelerator(Accelerator accelerator);

void setAccelerator(final VirtualKey key, final Modifier... modifier);

Accelerator getAccelerator();

void setMnemonic(Character mnemonic);

void setMnemonic(char mnemonic);

Character getMnemonic();
```

3.11.3.4 Der Enabled State

Mit den folgenden Methoden kann der `enabled` State gesetzt und ausgelesen werden:

```
void setEnabled(boolean enabled);

boolean isEnabled();
```

Aktionen, welche nicht enabled sind, können nicht ausgeführt werden, und werden in der Regel *ausgegraut* dargestellt.

3.11.3.5 Der Visible State

Mit Hilfe der folgenden Methoden kann der `visible` State gesetzt und ausgelesen werden:

```
void setVisible(boolean visible);

boolean isVisible();
```

Items welche nicht `visible` sind, werden im zugehörigen Menü nicht angezeigt, ohne das das Item aus dem Menü Model entfernt werden muss.

3.11.3.6 ItemModelListener

Mit Hilfe der folgenden Methoden kann ein `IItemModelListener` registriert oder deregistriert werden:

```
void addItemModelListener(IItemModelListener listener);

void removeItemModelListener(IItemModelListener listener);
```

Dieser hat die folgende Methode:

```
void itemChanged(IItemModel item);
```

Diese wird immer aufgerufen, wenn ein Item geändert wurde, wie zum Beispiel der Label Text, das Icon oder der `enabled` oder `visible` State.

3.11.3.7 Kopieren von Item Models

Mit Hilfe der folgenden Methode kann eine Kopie (*Deep Copy*) eines Item Models erzeugt werden. Die Kopie hat dabei die *gleichen* Items wie das Original (jedoch nicht die *Selben*). Registrierte Listener werden nicht mitkopiert. Dadurch sind bereits gebundene Items nicht an die Kopie gebunden.

```
IItemModel createCopy();
```

3.11.4 Die Schnittstelle IItemModelBuilder

Jedes Item Model kann mit Hilfe eines Builders erzeugt werden. Die Schnittstelle `IItemModelBuilder` liefert die gemeinsamen Methoden aller Item Model Builder. Sie hat die folgenden Methoden:

```
BUILDER_TYPE setId(String id);

BUILDER_TYPE setText(final String text);

BUILDER_TYPE setToolTipText(String toolTipText);

BUILDER_TYPE setIcon(IImageConstant icon);

BUILDER_TYPE setAccelerator(Accelerator accelerator);

BUILDER_TYPE setAccelerator(final VirtualKey key, final Modifier... modifier);

BUILDER_TYPE setAccelerator(final char key, final Modifier... modifier);

BUILDER_TYPE setMnemonic(Character mnemonic);

BUILDER_TYPE setMnemonic(char mnemonic);

BUILDER_TYPE setEnabled(boolean enabled);

ITEM_TYPE build();
```

Der Builder Type, welchen die oberen Methoden als Rückgabewert haben, ist dabei der konkrete Typ des Builders, welchen man verwendet. Dadurch können die Methoden, wie beim Builder Pattern üblich, verkettet aufgerufen werden. Die Semantik der Methoden ist analog zur Schnittstelle `IItemModel`.

Die Methode `build()` liefert ein neues Item zurück. Der Typ hängt vom konkreten Builder ab. Zum Beispiel gibt ein `IMenuModelBuilder` ein `IMenuModel` zurück.

Das folgende Beispiel soll die Verwendung demonstrieren:

```
1     final IActionItemModel pasteActionItem
2         = ActionItemModel
3             .builder()
4             .setText("Paste")
5             .setIcon(IconsSmall.PASTE)
6             .setAccelerator(VirtualKey.V,Modifier.CTRL)
7             .build();
```

3.11.5 Menu Model

Ein Menu Model ist ein Model für Menüs. Dazu zählen die Main Menüs in einer Menu Bar, Sub Menüs, Popup Menüs oder Toolbar Menüs. Es folgt eine Beschreibung der zusätzlich zu `IItemModel` vorhandenen Methoden der Schnittstelle `IMenuModel`:

3.11.5.1 Hinzufügen von existierenden Items

Mit Hilfe der folgenden Methoden können bereits existierende Items hinzugefügt werden:

```
    void addItem(final IMenuItemModel item);

    void addItem(final int index, final IMenuItemModel item);
```

3.11.5.2 Hinzufügen von Items mit Hilfe von Item Model Buildern

Mit den folgenden Methoden lassen sich Items mit Hilfe von Item Model Buildern hinzufügen:

```
    <MODEL_TYPE extends IMenuItemModel, BUILDER_TYPE extends IItemModelBuilder<?, MODEL_TYPE>>
        MODEL_TYPE addItem(final BUILDER_TYPE itemBuilder);

    <MODEL_TYPE extends IMenuItemModel, BUILDER_TYPE extends IItemModelBuilder<?, MODEL_TYPE>>
        MODEL_TYPE addItem(int index, final BUILDER_TYPE itemBuilder);
```

Dabei wird beim Hinzufügen die `build()` Methode aufgerufen, das gebaute Item hinzugefügt und zurückgegeben. Damit können zum Beispiel Konstrukte der folgenden Art realisiert werden:

```
1     IActionItemModel action = menu.addItem(ActionItemModel.builder().setText("Action"));
2     IMenuModel subMenu = menu.addItem(MenuModel.builder().setText("SubMenu"));
```

3.11.5.3 Erstellen und Hinzufügen von Items mit einem Aufruf

Die folgenden Methoden erlauben das Erstellen und Hinzufügen von Items mit einem Aufruf:

```
    IActionItemModel addAction(IAction action);

    IActionItemModel addAction(final int index, IAction action);

    IActionItemModel addActionItem();
```

```
IActionItemModel addActionItem(String text);

IActionItemModel addActionItem(String text, String toolTipText);

IActionItemModel addActionItem(String text, IImageConstant icon);

IActionItemModel addActionItem(String text, String toolTipText, IImageConstant icon);

ICheckedItemModel addCheckedItem();

ICheckedItemModel addCheckedItem(String text);

ICheckedItemModel addCheckedItem(String text, String toolTipText);

ICheckedItemModel addCheckedItem(String text, IImageConstant icon);

ICheckedItemModel addCheckedItem(String text, String toolTipText, IImageConstant icon);

IRadioItemModel addRadioItem();

IRadioItemModel addRadioItem(String text);

IRadioItemModel addRadioItem(String text, String toolTipText);

IRadioItemModel addRadioItem(String text, IImageConstant icon);

IRadioItemModel addRadioItem(String text, String toolTipText, IImageConstant icon);

ISeparatorItemModel addSeparator();

ISeparatorItemModel addSeparator(String id);

ISeparatorItemModel addSeparator(int index);

IMenuModel addMenu();

IMenuModel addMenu(String text);

IMenuModel addMenu(String text, String toolTipText);

IMenuModel addMenu(String text, IImageConstant icon);

IMenuModel addMenu(String text, String toolTipText, IImageConstant icon);
```

Dabei wird (mit Hilfe der Parameter) ein neues Item Model erzeugt, hinzugefügt und zurückgegeben. Das folgende Beispiel demonstriert die Verwendung:

```
1  MenuModel menu = new MenuModel("Menu");
2
3  IActionItemModel action1 = menu.addActionItem("Action1");
4  IActionItemModel action2 = menu.addActionItem("Action2");
5  IActionItemModel action3 = menu.addActionItem("Action3");
6
7  menu.addSeparator();
8  ICheckedItemModel option1 = menu.addCheckedItem("Option1");
9  ICheckedItemModel option2 = menu.addCheckedItem("Option2");
10 option2.setSelected(true);
11
12 menu.addSeparator();
13 IRadioItemModel radio1 = menu.addRadioItem("Radio1");
14 radio1.setSelected(true);
15 IRadioItemModel radio2 = menu.addRadioItem("Radio2");
16 IRadioItemModel radio3 = menu.addRadioItem("Radio3");
```

```
17
18      IMenuModel subMenu = menu.addMenu("Submenu");
19      IActionItemModel action4 = subMenu.addActionItem("Action4");
20      IActionItemModel action5 = subMenu.addActionItem("Action5");
```

Die folgende Abbildung zeigt das Ergebnis:

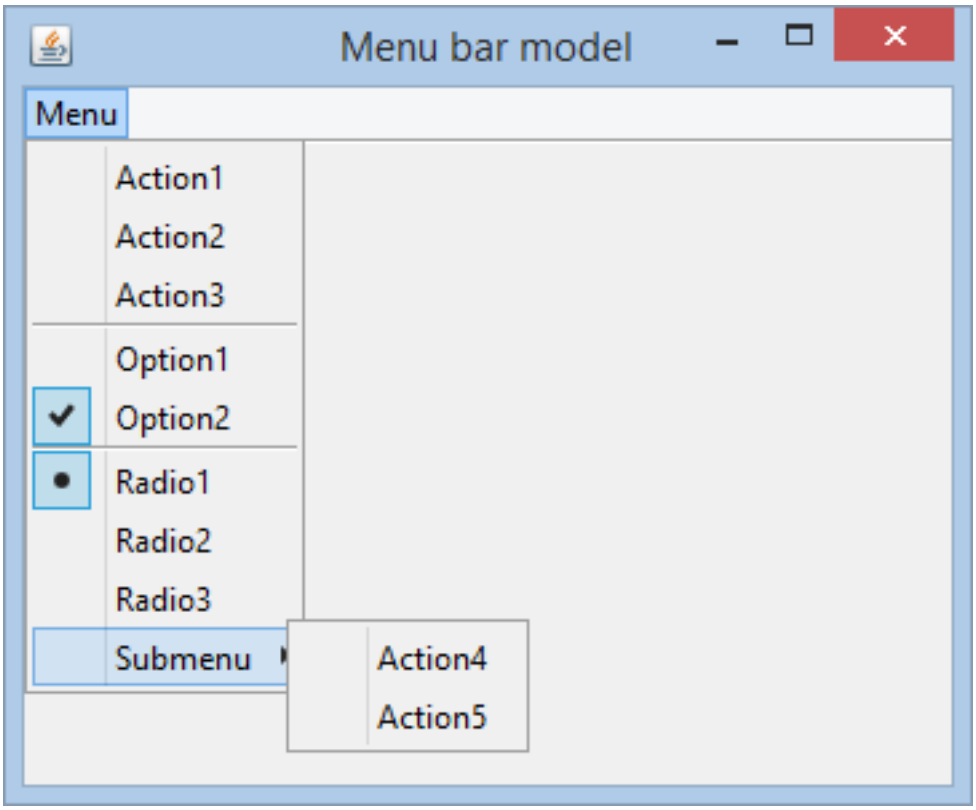


Abbildung 3.23: Menu Model Beispiel

3.11.5.4 Hinzufügen von Items relativ zu anderen Items

Die folgenden Methoden können verwendet werden, um ein Item an eine bestimmte Stelle in einem Menu (inklusive Sub Menüs) hinzuzufügen:

```
    void addBefore(IMenuItemModel newItem, String... idPath);

    void addAfter(IMenuItemModel newItem, String... idPath);
```

Der `idPath` gibt dabei den Pfad der id´s an, über welche die Einfügestelle lokalisiert werden kann. Ist diese nicht vorhanden, wird eine `IllegalArgumentException` geworfen. Mittels der Methode `findItemByPath(String... idPath)` kann vorab überprüft werden, ob der Pfad vorhanden ist.

Die Verwendung soll anhand eines Beispiels demonstriert werden. Der folgende Code erzeugt ein Menü und fügt einen Separator hinzu (Zeile 3), zu dem man Custom Actions hinzugefügen können soll:

```
final MenuModel menu = new MenuModel("Menu");
menu.addItem(save);
menu.addSeparator(CUSTOM_ACTIONS);
menu.addSeparator(EDIT_ACTIONS);
menu.addItem(copy);
menu.addItem(paste);
```

An einer anderen Stelle könnte man eine Contribution zum Menü wie folgt machen:

```
final ActionItemModel customAction = new ActionItemModel("Custom action");
menu.addAfter(customAction, CUSTOM_ACTIONS);
```

Die folgende Abbildung zeigt das Ergebnis:

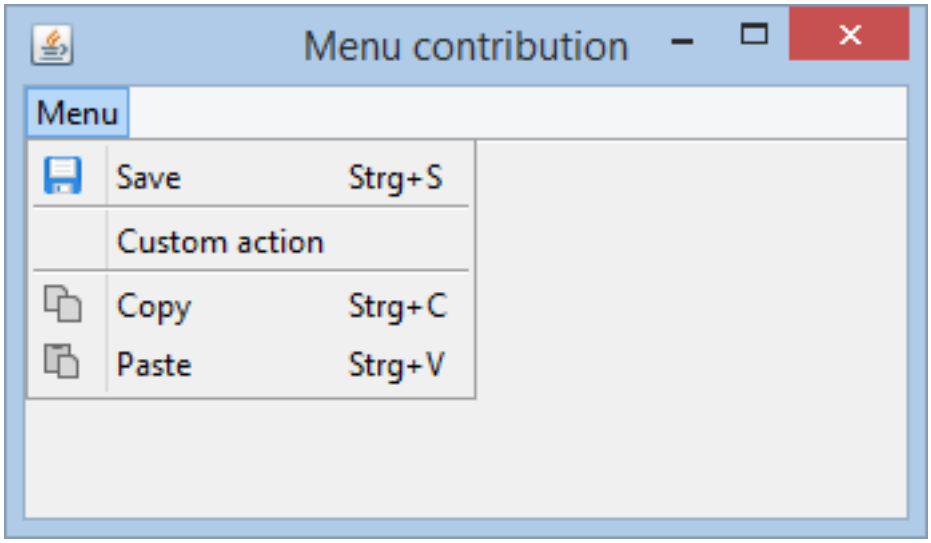


Abbildung 3.24: Menu Contribution Beispiel

3.11.5.5 Hinzufügen von Items eines anderen Menüs

Mit Hilfe der folgenden Methode werden alle Items der übergebenen Menüs hinzugefügt:

```
void addItemsOfModel(IMenuModel menuModel);
```

Dabei wird eine Referenz der Items hinzugefügt und keine Kopie!

3.11.5.6 Entfernen von Items

Mit Hilfe der folgenden Methoden können Items aus einem Menü entfernt werden:

```
void removeItem(final IMenuItemModel item);

void removeItem(int index);

void removeAllItems();
```

3.11.5.7 ListModelListener

Um sich über das Hinzufügen oder Entfernen von Items informieren zu lassen, kann ein `IListModelListener` verwendet werden:

```
void addListModelListener(IListModelListener listener);

void removeListModelListener(IListModelListener listener);
```

Dieser hat die folgenden Methoden:

```
void afterChildAdded(int index);

void beforeChildRemove(int index);

void afterChildRemoved(int index);
```

3.11.5.8 Zugriff auf die Items eines Menüs

Die folgende Methode liefert eine nicht modifizierbare Kopie der aktuell vorhandenen Menu Item Models des Menu Model.

```
List<IMenuItemModel> getChildren();
```

Um ein Item Anhand eines id Pfades zu finden, kann die folgende Methode verwendet werden:

```
IMenuItemModel findItemByPath(String... idPath);
```

Existiert kein solches Item im Menü, wird null zurückgegeben.

3.11.5.9 Action Decorator

Mit Hilfe der folgenden Methoden kann ein Decorator hinzugefügt werden, der alle Actions des Menüs (inklusive aller Untermenüs) dekoriert:

```
void addDecorator(IDecorator<IAction> decorator);

void removeDecorator(IDecorator<IAction> decorator);
```

Die Dekorierung betrifft alle derzeit vorhanden und zukünftig hinzugefügten Actions. Für weitere Details zu Actions siehe auch Actions und Commands.

3.11.5.10 Menu Model Builder

Die Schnittstelle `IMenuModelBuilder` ist von `IItemModelBuilder` abgeleitet und liefert einen konkreten Builder für Menu Models. Sie hat keine zusätzlichen Methoden. Eine Instanz erhält man von der Klasse `org.jowidgets.tools.model.item.MenuModel`.

3.11.5.11 Menu Model Instanzen

Die Klasse `org.jowidgets.tools.model.item.MenuModel` liefert zum Einen statische Methoden für die Erzeugung eines `IMenuModelBuilder`. Zum Anderen implementiert die Klasse die Schnittstelle `IMenuModel`. Das folgende Beispiel zeigt die Verwendung des Builders:

```
final IMenuModel menu
        = MenuModel
            .builder()
            .setText("Edit menu")
            .setIcon(IconsSmall.EDIT)
            .build();
```

Mit Hilfe einer Instantiierung mittels `new` kann das gleiche so erreicht werden:

```
IMenuModel menu = new MenuModel("Edit menu", IconsSmall.EDIT);
```

3.11.6 Action Item Model

Ein Action Item Model ist ein Model für Items welche Aktionen auslösen. Dazu zählen Action Menu Items und Tool Bar Buttons. Die Schnittstelle `IActionItemModel` ist von `IItemModel` abgeleitet. Es folgt eine Beschreibung der wichtigsten weiteren Methoden:

3.11.6.1 Action binding

Mit Hilfe der folgenden Methoden kann eine Action an ein Action Item Model gebunden, sowie die gebundene Action abgefragt werden:

```
void setAction(IAction action);

IAction getAction();
```

Ein Action Item Model muss nicht zwingend an eine Action gebunden sein, so dass `null` sowohl gesetzt werden kann, als auch als Rückgabewert möglich ist. Für weitere Details zu Actions siehe auch Actions und Commands.

3.11.6.2 Action Dekorierung

Die folgenden Methoden können verwendet werden, um einen Action Dekorierer hinzuzufügen oder zu entfernen.

```
void addDecorator(IDecorator<IAction> decorator);

void removeDecorator(IDecorator<IAction> decorator);
```

Dieser wird verwendet, um eine gebundene Action zu dekorieren. Für weitere Details zu Actions siehe auch Actions und Commands.

3.11.6.3 ActionListener

Mit Hilfe der folgenden Methoden kann ein `IActionListener` hinzugefügt oder entfernt werden:

```
void addActionListener(final IActionListener actionListener);

void removeActionListener(final IActionListener actionListener);
```

Dieser hat die folgende Methode:

```
void actionPerformed();
```

Die Methode wird aufgerufen, wenn auf einem gebundenen Item eine Aktion (z.B. durch Klicken oder Tastaturkürzel) ausgelöst wurde.

3.11.6.4 Action Item Model Builder

Die Schnittstelle `IActionItemModelBuilder` ist von `IItemModelBuilder` abgeleitet und liefert einen konkreten Builder für Action Item Models. Sie hat folgende zusätzliche Methoden.

```
IActionItemModelBuilder setAction(IAction action);

IActionItemModelBuilder addVisibilityAspect(IActionItemVisibilityAspect visibilityAspect);
```

Die erste Methode dient zum Binding einer Action, die zweite Methode fügt einen Sichtbarkeitsaspket hinzu.

Eine Instanz erhält man von der Klasse `org.jowidgets.tools.model.item.ActionItemModel`.

3.11.6.5 Action Item Model Instanzen

Die Klasse `org.jowidgets.tools.model.item.ActionItemModel` liefert zum Einen statische Methoden für die Erzeugung eines `IActionItemModelBuilder`. Zum Anderen implementiert die Klasse die Schnittstelle `IActionItemModel`. Das folgende Beispiel zeigt die Verwendung des Builders:

```
final IActionItemModel paste
    = ActionItemModel
        .builder()
        .setText("Paste")
        .setIcon(IconsSmall.PASTE)
        .setAccelerator(VirtualKey.V,Modifier.CTRL)
        .build();
```

Mit Hilfe einer Instantiierung mittels `new` kann das gleiche so erreicht werden:

```
final IActionItemModel paste = new ActionItemModel("Paste", IconsSmall.PASTE);
paste.setAccelerator(VirtualKey.V, Modifier.CTRL);
```

3.11.7 Die Schnittstelle ISelectableMenuItemModel

Die Schnittstelle `ISelectableItemModel` liefert die Funktionen für ein Checked Item Model und ein Radio Item Model. Die Schnittstelle ist von `IItemModel` abgeleitet. Es folgt eine Beschreibung der weiteren Methoden:

3.11.7.1 Der Selected State

Mit Hilfe der folgenden Methoden kann der `selected` State gesetzt und ausgelesen werden:

```
boolean isSelected();

void setSelected(boolean selected);
```

Um sich über Änderungen des `selected` State informieren zu lassen, kann ein `IItemStateListener` verwendet werden:

```
void addItemListener(final IItemStateListener listener);

void removeItemListener(final IItemStateListener listener);
```

Dieser hat die folgende Methode:

```
void itemStateChanged();
```

3.11.8 Checked Item Model

Ein Checked Item Model ist ein Model für Items welche eine unabhängige Option anzeigen. Dazu zählen Checked Menu Items und Toolbar Toggle Buttons. Die Schnittstelle `ICheckedItemModel` ist von `ISelectableMenuItemModel` abgeleitet und hat keine weiteren Methoden.

3.11.8.1 Checked Item Model Builder

Die Schnittstelle `ICheckedItemModelBuilder` ist von `IItemModelBuilder` abgeleitet und liefert einen konkreten Builder für Checked Item Models. Sie hat die folgende zusätzliche Methode:

```
ICheckedItemModelBuilder setSelected(boolean selected);
```

Eine Instanz erhält man von der Klasse `org.jowidgets.tools.model.item.CheckedItemModel`.

3.11.8.2 Checked Item Model Instanzen

Die Klasse `org.jowidgets.tools.model.item.CheckedItemModel` liefert zum Einen statische Methoden für die Erzeugung eines `ICheckedItemModelBuilder`. Zum Anderen implementiert die Klasse die Schnittstelle `ICheckedItemModel`. Das folgende Beispiel zeigt die Verwendung des Builders:

```
1  final ICheckedItemModel filter
2          = CheckedItemModel
3              .builder()
4              .setText("Filter")
5              .setToolTipText("Indicates if filter is active or not")
6              .setIcon(IconsSmall.FILTER)
7              .setAccelerator(VirtualKey.F, Modifier.ALT)
8              .build();
```

Mit Hilfe einer Instantiierung mittels `new` kann das gleiche so erreicht werden:

```
    final ICheckedItemModel filter = new CheckedItemModel(
            "Filter",
            "Indicates if filter is used or not",
            IconsSmall.FILTER);
    filter.setAccelerator(VirtualKey.F, Modifier.ALT);
```

3.11.9 Radio Item Model

Ein Radio Item Model ist ein Model für Items welche eine Option innerhalb einer Optionsliste anzeigen. Dazu zählt das Radio Menu Item. Die Schnittstelle `IRadioItemModel` ist von `ISelectableMenuItemModel` abgeleitet und hat keine weiteren Methoden.

3.11.9.1 Radio Item Model Builder

Die Schnittstelle `IRadioItemModelBuilder` ist von `IItemModelBuilder` abgeleitet und liefert einen konkreten Builder für Radio Item Models. Sie hat die folgende zusätzliche Methode:

```
    IRadioItemModelBuilder setSelected(boolean selected);
```

Eine Instanz erhält man von der Klasse `org.jowidgets.tools.model.item.RadioItemModel`.

3.11.9.2 Radio Item Model Instanzen

Die Klasse `org.jowidgets.tools.model.item.RadioItemModel` liefert zum Einen statische Methoden für die Erzeugung eines `IRadioItemModelBuilder`. Zum Anderen implementiert die Klasse die Schnittstelle `IRadioItemModel`. Das folgende Beispiel zeigt die Verwendung des Builders:

```
    final IRadioItemModel low
            = RadioItemModel
                .builder()
                .setText("Low latency")
                .setToolTipText("Uses low latency which may lead to high workload")
                .setAccelerator(VirtualKey.L, Modifier.CTRL)
                .build();

    final IRadioItemModel med
            = RadioItemModel
                .builder()
                .setText("Medium latency")
                .setToolTipText("Uses medium latency which may lead to balanced workload")
                .setAccelerator(VirtualKey.M, Modifier.CTRL)
                .setSelected(true)
                .build();

    final IRadioItemModel high
            = RadioItemModel
                .builder()
                .setText("High latency")
                .setToolTipText("Uses high latency which may lead to low workload")
                .setAccelerator(VirtualKey.H, Modifier.CTRL)
                .build();
```

Mit Hilfe einer Instantiierung mittels `new` kann das gleiche so erreicht werden:

```
    final IRadioItemModel low = new RadioItemModel(
            "Low latency",
            "Uses low latency which may lead to high workload");
    low.setAccelerator(VirtualKey.L, Modifier.CTRL);

    final IRadioItemModel med = new RadioItemModel(
            "Medium latency",
            "Uses medium latency which may lead to balanced workload");
    med.setAccelerator(VirtualKey.M, Modifier.CTRL);

    final IRadioItemModel high = new RadioItemModel(
            "High latency",
            "Uses high latency which may lead to low workload");
    high.setAccelerator(VirtualKey.H, Modifier.CTRL);
```

3.11.10 Separator Item Model

Ein Separator Item Model ist ein Model für Separator Items. Dazu zählen Separator Menu Items und Toolbar Separators. Die Schnittstelle `ISeparatorItemModel` ist von `IItemModel` abgeleitet und hat keine weiteren Methoden.

3.11.10.1 Separator Item Model Builder

Die Schnittstelle `ISeparatorItemModelBuilder` ist von `IItemModelBuilder` abgeleitet und liefert einen konkreten Builder für Separator Item Models. Sie hat keine zusätzlichen Methoden. Eine Instanz erhält man von der Klasse `org.jowidgets.tools.model.item.ISeparatorItemModel`.

3.11.10.2 Separator Item Model Instanzen

Die Klasse `org.jowidgets.tools.model.item.SeparatorItemModel` liefert zum Einen statische Methoden für die Erzeugung eines `ISeparatorItemModelBuilder`. Zum Anderen implementiert die Klasse die Schnittstelle `ISeparatorItemModel`. Das folgende Beispiel zeigt die Verwendung des Builders:

```
    ISeparatorItemModel separator
            = SeparatorItemModel
                .builder()
                .setId(CUSTOM_ACTIONS)
                .build();
```

Mit Hilfe einer Instantiierung mittels `new` kann das gleiche so erreicht werden:

```
    ISeparatorItemModel separator = new SeparatorItemModel(CUSTOM_ACTIONS);
```

3.11.11 Menu Model Key Binding

Die Key Accelerators in einem Popup Menüs werden, im Vergleich zu denen in einer Menu Bar, nicht automatisch gebunden. Die Klasse `org.jowidgets.tools.model.item.MenuModelKeyBinding` bietet verschiedenen Möglichkeiten, die Key Accelerator der Action Item Models eines Menu Models an ein `IComponent` zu binden.

Die folgende statische Methode führt ein Binding eines Menu Model mit einem Component durch:

```
public static MenuModelKeyBinding bind(final IMenuModel menu, final IComponent component) {...}
```

Tritt auf dem `component` (oder auf einem seiner Kinder, falls es ein IContainer ist) ein Key Event auf, wird geprüft, ob dafür eine Aktion existiert, und wenn ja, wird diese ausgeführt. Wird das übergebene `component` disposed, wird automatisch auch das Binding disposed. Es werden auch die Aktionen gebunden, welche erst nach dem Aufruf von `bind()` hinzugefügt wurden.

Das folgende Snipped zeigt die Verwendung (das vollständige Snipped findet sich hier).

```
1  //create a root frame
2  final IFrameBluePrint frameBp = BPF.frame();
3  frameBp.setSize(new Dimension(400, 300)).setTitle("Menu model key binding");
4  final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);
5  frame.setLayout(FillLayout.get());
6
7  //create a popup menu with some actions
8  final MenuModel popup = new MenuModel();
9
10 final IActionItemModel action1 = popup.addActionItem("Action1");
11 action1.setAccelerator(VirtualKey.DIGIT_1, Modifier.CTRL);
12
13 final IActionItemModel action2 = popup.addActionItem("Action2");
14 action2.setAccelerator(VirtualKey.DIGIT_2, Modifier.CTRL);
15
16 final IActionItemModel action3 = popup.addActionItem("Action3");
17 action3.setAccelerator(VirtualKey.DIGIT_3, Modifier.CTRL);
18
19 //set the popup menu for the frame
20 frame.setPopupMenu(popup);
21
22 //do the key binding to the frame (recursive)
23 MenuModelKeyBinding.bind(popup, frame);
24
25 //set the root frame visible
26 frame.setVisible(true);
27
28 //add some actions after binding to show that they are bound too
29 final IMenuModel submenu = popup.addMenu("Submenu");
30
31 final IActionItemModel action4 = submenu.addActionItem("Action4");
32 action4.setAccelerator(VirtualKey.DIGIT_4, Modifier.CTRL);
33
34 final IActionItemModel action5 = submenu.addActionItem("Action5");
35 action5.setAccelerator(VirtualKey.DIGIT_5, Modifier.CTRL);
36
37 //add listeners to the items
38 action1.addActionListener(new SysoutActionListener(action1));
39 action2.addActionListener(new SysoutActionListener(action2));
40 action3.addActionListener(new SysoutActionListener(action3));
41 action4.addActionListener(new SysoutActionListener(action4));
42 action5.addActionListener(new SysoutActionListener(action5));
```

In Zeile 23 wird das Binding durchgeführt.

3.12 Actions und Commands

Eine `IAction` dient dazu, eine Nutzeraktion zu kapseln. Dabei soll sowohl der Action Code als eine Action Instanz wiederverwendet werden können. Actions können zu einem Menu hinzugefügt werden oder an ein Button oder Toolbar Button gebunden werden.

3.12.1 Die Schnittstelle IAction

Die Schnittstelle `IAction` liefert die Methoden, welche jowidgets benötigt, um Aktionen anzuzeigen, zu binden und auszuführen. Sie stellt somit in gewisser Weise die SPI[9] für Aktionen dar. Wenn man Command Actions verwendet, muss man diese Schnittstelle nicht selbst implementieren. Dennoch sollen die Methoden zum besseren Verständnis kurz beschrieben werden:

3.12.1.1 Action Description

Die folgenden Methoden liefern die beschreibenden Attribute der Action für die Anzeige und das Key Binding:

```
String getText();

String getToolTipText();

IImageConstant getIcon();

Character getMnemonic();

Accelerator getAccelerator();
```

3.12.1.2 Enabled State

Die folgende Methode liefert den `enabled` State. Ein Action, welche nicht `enabled` ist, kann nicht ausgeführt werden und wird *ausgegraut* dargestellt.

```
boolean isEnabled();
```

3.12.1.3 Execute

Die folgende Methode wird aufgerufen, wenn die Action ausgeführt werden soll:

```
void execute(IExecutionContext executionContext) throws Exception;
```

3.12.1.4 Exception Handler

Eine `IAction` Implementierung kann einen `IExceptionHandler` liefern:

```
IExceptionHandler getExceptionHandler();
```

Dieser wird verwendet, wenn bei der `execute()` Methode eine Exception auftritt. Wird `null` zurückgegeben, wird der `UncaughtExceptionHandler` des aktuellen Threads aufgerufen.

[9] Service Provider Interface

3.12.1.5 Change Observable

Wird eine Action *mutable* entworfen, was bedeutet, dass sich die Attribute `text`, `tooltipText`, `icon` oder `enabled` State zur Laufzeit ändern können, dann liefert die folgenden Methode ein `IActionObservable` zurück, auf dem man sich als Listener über Änderungen informieren kann:

```
IActionChangeObservable getActionChangeObservable();
```

Ist die Action *immutable* entworfen, kann die Implementierung `null` zurückliefern. Die Schnittstelle `IActionObservable` hat die folgenden Methoden:

```
void addActionChangeListener(IActionChangeListener listener);

void removeActionChangeListener(IActionChangeListener listener);
```

Ein `IActionChangeListener` hat die folgenden Methoden:

```
void textChanged();

void toolTipTextChanged();

void iconChanged();

void enabledChanged();
```

Der Accelerator Key und Mnemonic Key sind somit immer *immutable* zu entwerfen. Nur die oben angegebenen Attribute können sich zur Laufzeit ändern.

3.12.2 Die Schnittstelle ICommand

Eine Command Action teilt eine Action in den beschreiben Anteil wie Label Text, Tooltip Text oder Icon, und die Business Logik auf. Ein `ICommand` stellt dabei die Business Logik dar. Diese hat drei Aspekte:

- **Enabled Checking** prüft, ob die Aktion ausführbar ist.
- **Execution** führt die eigentliche Aktion aus
- **Exception Handling** behandelt Ausnahmen

Die Schnittstelle `ICommand` wird häufig selbst implementiert, um die Business Logik abzubilden. Diese hat die folgenden Methoden:

```
ICommandExecutor getCommandExecutor();

IExceptionHandler getExceptionHandler();

IEnabledChecker getEnabledChecker();
```

Dabei kann jede der Methoden `null` zurückliefern, wenn der jeweilige Aspekt nicht unterstützt wird (einen Enabled Checker oder Exception Handler anzubieten, ohne einen Executor zu haben ist allerdings nicht unbedingt sinnvoll).

3.12.2.1 Die Schnittstelle ICommandExecutor

Die Schnittstelle `ICommandExecutor` hat die folgende Methode:

```
    void execute(IExecutionContext executionContext) throws Exception;
```

Diese wird aufgerufen, wenn der Command ausgeführt werden soll. Die Schnittstelle `IExecutionContext` liefert dabei die folgenden Methoden:

```
    IAction getAction();

    IWidget getSource();

    <VALUE_TYPE> VALUE_TYPE getValue(final ITypedKey<VALUE_TYPE> key);
```

Damit kann man eine Referenz auf die auslösende Action, das Widget, für welches die Action ausgeführt wurde, und weitere Properties erhalten. Über die ausführende Action kann man sich Beispielsweise das Action Label und Icon beschaffen, um diese etwa in einem Dialog anzuzeigen. Das folgende Beispiel soll das verdeutlichen:

```
1  public final class ExampleExecutor implements ICommandExecutor {
2
3      @Override
4      public void execute(final IExecutionContext executionContext) throws Exception {
5          final IAction action = executionContext.getAction();
6          final String title = action.getText();
7          final IImageConstant icon = action.getIcon();
8          final String message = "Execution was successful";
9          MessagePane.showInfo(title, icon, message);
10     }
11
12 }
```

Das dies ein häufiger Anwendungsfall ist, unterstützt die Accessor Klasse `MessagePane` auch das direkte Übergeben des `executionContext`. Die folgende Implementierung hat daher den gleichen Effekt:

```
1  public final class ExampleExecutor implements ICommandExecutor {
2
3      @Override
4      public void execute(final IExecutionContext executionContext) throws Exception {
5          final String message = "Execution was successful";
6          MessagePane.showInfo(executionContext, message);
7      }
8
9  }
```

3.12.2.2 Die Schnittstelle IExceptionHandler

Die Schnittstelle `IExceptionHandler` hat die folgende Methode:

```
    void handleException(
        IExecutionContext executionContext,
        final Exception exception) throws Exception;
```

Auch hier wird der `IExecutionContext` der auslösenden Action übergeben. Zudem bekommt man die aufgetretene Exception übergeben. Wenn man diese nicht selbst behandeln kann, kann man sie erneut werfen. Sie wird dann als nächstes vom Exception Handler der Action behandelt, falls ein solcher existiert.

3.12.2.3 Die Schnittstelle IEnabledChecker

Die Schnittstelle `IEnabledChecker` liefert die folgenden Methoden:

```
IEnabledState getEnabledState();

void addChangeListener(IChangeListener listener);

void removeChangeListener(IChangeListener listener);
```

Die Methode `getEnabledState()` liefert die Information, ob die Aktion ausführbar ist. Immer wenn sich der EnabledState ändert, müssen die registrierten Listener darüber informiert werden.

Die Schnittstelle `IEnabledState` hat die folgenden Methoden:

```
boolean isEnabled();

String getReason();
```

Das bedeutet, es wird nicht nur die Information geliefert, ob eine Aktion ausführbar ist, sondern auch der Grund warum nicht. Die Methode `getReason()` sollte einen internationalisierten String zurückliefern, welche dem Nutzer Auskunft darüber gibt, warum die Aktion nicht ausführbar ist. Beispiele sind:

- **Speichern** - "Es gibt keine Änderungen"
- **Speichern** - "Es existiert bereits ein Datensatz mit gleicher Artikelnummer"
- **Undo** - "Es gibt keine Änderungen"
- **Nachricht versenden** - "Die Nachricht hat keinen Betreff"
- **Löschen** - "Fehlendes Recht"

Es ist jedoch auch erlaubt `null` oder einen `Leerstring` zurück zu liefern.

Die Default Implementierung der Command Action **tauscht**, wenn der `EnabledState` disabled und der `reason` nicht leer ist, **das Tooltip** des gebundenen MenuItem, Button oder ToolbarButton **gegen den `reason` Text aus**.

Dies hat sich in großen und komplexen Enterprise Anwendungen als äußerst nützlich herausgestellt und wurde von Kunden mehrfach gelobt. So wurde sogar die Aussage getätigt, das man dieses Feature in vielen anderen Applikationen vermisse, seit dem man mit dieser Applikation arbeiten würde.

Wenn man bei Google den Text *"why is that button greyed out in"* eingibt, liefert einem die Autovervollständigung unzählige Fortführungen dieses Satzes, woraus sich vermuten lässt, dass sich Nutzer diese Frage häufig zu stellen scheinen. Da der Entwickler den Grund für das Ausgrauen in der Regel kennt, wäre es doch auch hilfreich, diesen an den Nutzer weiter zu geben, um die Usability zu erhöhen.

Die Klasse `EnabledState` liefert folgende statische Methode zur Erzeugung eines `disabled` State:

```
public static EnabledState disabled(final String reason) {...}
```

Sowie die folgenden Konstanten:

```
public static final EnabledState ENABLED = new EnabledState();

public static final EnabledState DISABLED = new EnabledState(false, null);
```

Das folgende Beispiel demonstriert die Implementierung eines `IEnabledChecker`:

```
1  public final class ModifiedEnabledChecker extends AbstractEnabledChecker {
2
3      private final IInputComponent<?> inputComponent;
4
5      private ModifiedEnabledChecker(final IInputComponent<?> inputComponent) {
6          this.inputComponent = inputComponent;
7
8          inputComponent.addInputListener(new IInputListener() {
9              @Override
10             public void inputChanged() {
11                 fireEnabledStateChanged();
12             }
13         });
14     }
15
16     @Override
17     public IEnabledState getEnabledState() {
18         if (!inputComponent.hasModifications()) {
19             return EnabledState.disabled("There is no modification");
20         }
21         else {
22             return EnabledState.ENABLED;
23         }
24     }
25
26 }
```

Dieser erlaubt die Ausführung nur, wenn die referenzierte Input Component Modifikationen hat. Für die Implementierung wird von der abstrakten Klasse `AbstractEnabledChecker` abgeleitet. Dadurch spart man sich die Implementierung Methoden `addChangeListener()` und `removeChangeListener()`. Um Änderungen anzuzeigen muss nur die Methode `fireEnabledStateChanged()` aufgerufen werden (Zeile 11).

Mit Hilfe der Klasse `EnabledChecker` könnte man das gleiche wie oben auch so erreichen:

```
1      final EnabledChecker enabledChecker = new EnabledChecker();
2
3      inputComponent.addInputListener(new IInputListener() {
4          @Override
5          public void inputChanged() {
6              if (!inputComponent.hasModifications()) {
7                  enabledChecker.setDisabled("There is no modification");
8              }
9              else {
10                 enabledChecker.setEnabled();
11             }
12         }
13     });
```

Je nach Anwendungsfall kann die eine oder andere Variante besser geeignet sein.

3.12.3 Die Schnittstelle ICommandAction

Die Schnittstelle `ICommandAction` erweitert die Schnittstelle `IAction` um den Aspekt von Commands. Jowidgets liefert dafür eine Defaultimplementierung, so dass die Schnittstelle `ICommandAction` normalerweise nicht selbst implementiert wird, sondern nur die zugehörigen Commands. Es folgt eine Beschreibung der zusätzlich zu `IAction` vorhandenen Methoden:

3.12.3.1 Descriptor Methoden

Mit Hilfe der folgenden Methoden können `text`, `tooltipText` und `icon` geändert werden.

```
    void setText(String text);

    void setToolTipText(final String toolTipText);

    void setIcon(IImageConstant icon);
```

3.12.3.2 Enabled State

Mit der folgenden Methode kann der *gloable* `enabled` State geändert werden:

```
    void setEnabled(boolean enabled);
```

Wird ein `IEnabledChecker` verwendet, wird dieser nur ausgewertet, wenn der globale `enabled` State `true` ist.

3.12.3.3 Setzen, Ändern und Auslesen des Command

Mit Hilfe der folgenden Methoden kann der Command gesetzt und geändert werden:

```
    void setCommand(ICommand command);

    void setCommand(ICommandExecutor executor);

    void setCommand(ICommandExecutor executor, IEnabledChecker enabledChecker);

    void setCommand(ICommandExecutor executor, IExceptionHandler exceptionHandler);

    void setCommand(
        ICommandExecutor executor,
        IEnabledChecker enabledChecker,
        IExceptionHandler exceptionHandler);
```

Die erste Methode verwendet dazu die `ICommand` Schnittstelle, die anderen Methoden erlauben das Setzen eines Command mit Hilfe der einzelnen Aspekte (Executor, EnabledChecker, ExceptionHandler). Dabei werden immer alle Aspekte neu gesetzt (die nicht angegebenen werden Aspkete werden zu `null`). Ein Command kann explizit auf `null` gesetzt werden. Die Action wird dadurch automatisch `disabled`.

Die folgende Methode liefert das derzeit gesetzte Command:

```
    ICommand getCommand();
```

3.12.3.4 Exception Handling

Für alle Exceptions, welche nicht durch den Command behandelt wurden, kann ein Action Exception-Handler wie folgt gesetzt werden:

```
    void setActionExceptionHandler(IExceptionHandler exceptionHandler);
```

3.12.3.5 Action Builder

Um eine Implementierung der Schnittstelle ICommandAction zu erhalten, kann ein `IActionBuilder` verwendet werden. Diesen erhält man u.A. von der Accessor Klasse `org.jowidgets.api.command.Action` mit Hilfe der statischen Methode:

```
    public static IActionBuilder builder() {...}
```

Die Schnittstelle `IActionBuilder` hat die folgenden Methoden:

```
    IActionBuilder setText(String text);

    IActionBuilder setToolTipText(String toolTipText);

    IActionBuilder setIcon(IImageConstant icon);

    IActionBuilder setMnemonic(Character mnemonic);

    IActionBuilder setMnemonic(char mnemonic);

    IActionBuilder setAccelerator(Accelerator accelerator);

    IActionBuilder setAccelerator(char key, Modifier... modifier);

    IActionBuilder setAccelerator(VirtualKey virtualKey, Modifier... modifier);

    IActionBuilder setEnabled(boolean enabled);

    IActionBuilder setCommand(ICommand command);

    IActionBuilder setCommand(ICommandExecutor command);

    IActionBuilder setCommand(ICommandExecutor command, IEnabledChecker executableStateChecker);

    IActionBuilder setCommand(ICommandExecutor command, IExceptionHandler exceptionHandler);

    IActionBuilder setCommand(
        ICommandExecutor command,
        IEnabledChecker enabledChecker,
        IExceptionHandler exceptionHandler);

    IActionBuilder setActionExceptionHandler(IExceptionHandler exceptionHandler);

    ICommandAction build();
```

Die Parameter haben die gleiche Semantik wie die der Schnittstelle `ICommandAction`.

Das folgende Beispiel zeigt die Verwendung:

```
1  public final class SaveActionFactory {
2
3      private SaveActionFactory() {}
4
5      public static IAction create(final IDataModel dataModel) {
6          final IActionBuilder builder = Action.builder();
7
8          builder.setText("Save");
9          builder.setToolTipText("Saves the text");
10         builder.setAccelerator(VirtualKey.S, Modifier.CTRL);
11         builder.setIcon(IconsSmall.DISK);
12
```

```
13          builder.setCommand(SaveCommandFactory.create(dataModel));
14
15          return builder.build();
16      }
17  }
```

Anmerkung: Im obigen Beispiel wird bewusst eine `IAction` und keine `ICommandAction` zurückgegeben. Dadurch wird signalisiert, dass die Action nicht durch den Nutzer der Methode nachträglich modifiziert werden soll.

3.12.4 Command Action Snipped

Im folgenden soll die Verwendung von Command Actions noch einmal anhand eines Beispiels verdeutlicht werden. Die folgenden Abbildung zeigen vorab fertige Ergebnis:

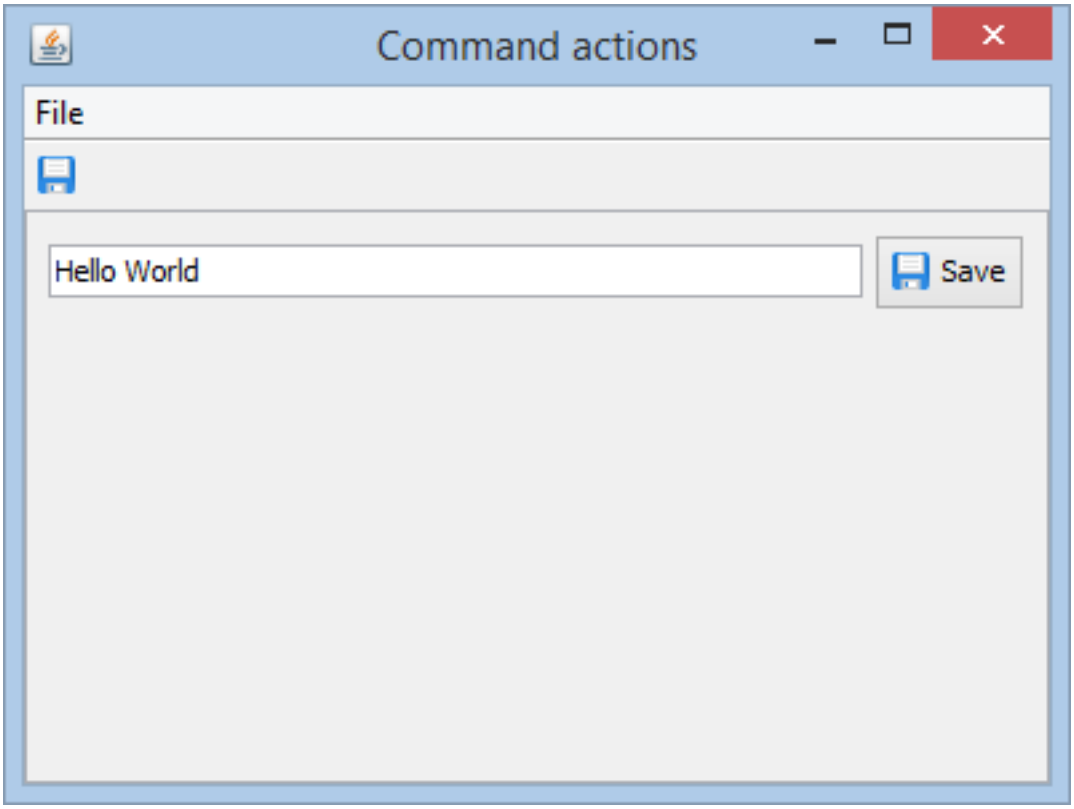


Abbildung 3.25: Command Action Snipped - Screenshot 1

In dem Textfeld wurde der Text *"Hello World"* durch den Benutzer eingegeben. Dadurch ist der Save Button neben dem Text Feld sowie in der Toolbar aktiviert. Zudem befindet sich die Action auch noch im File Menu (nicht auf dem Screenshot sichtbar).

Nachdem die Save Action ausgelöst wurde, ist das folgende zu sehen:

Ein Dialog erscheint, dass die Aktion ausgeführt wurde. Wird dieser geschlossen und öffnet man das File Menu, sieht man das folgende:

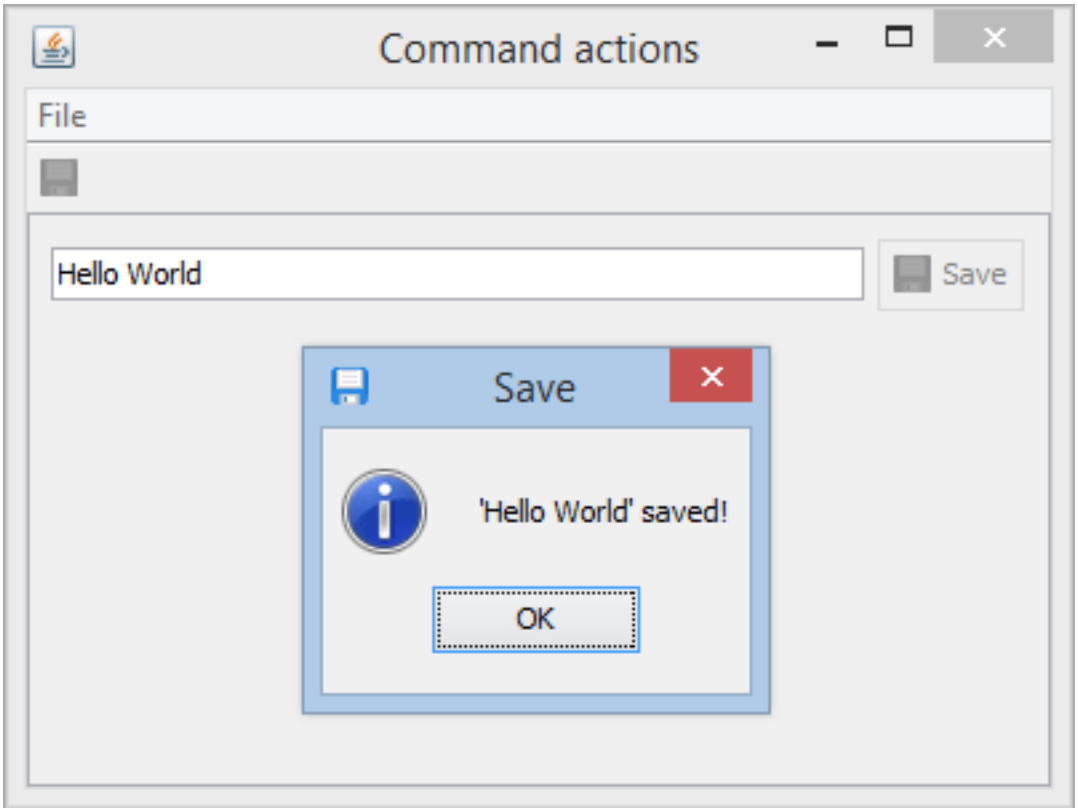


Abbildung 3.26: Command Action Snipped - Screenshot 2

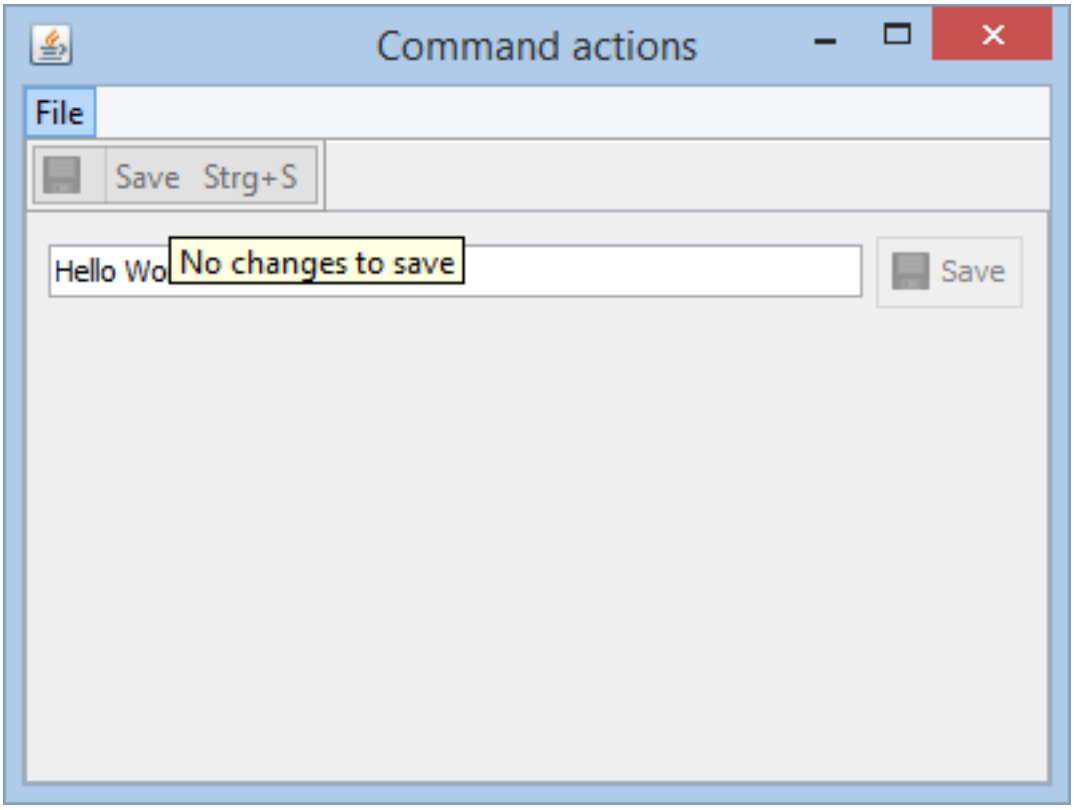


Abbildung 3.27: Command Action Snipped - Screenshot 3

Der Tooltip zu der Save Action im File Menü zeigt den Grund für die Deaktivierung im Tooltip. Die Save Action soll genau dann `enabled` sein, wenn der Text im Textfeld sich seit dem letzten Speichern geändert hat.

Der folgende Code zeigt die Implementierung (das vollständige Snipped inklusive Imports befindet sich hier):

```
public final class CommandActionSnipped implements IApplication {

    @Override
    public void start(final IApplicationLifecycle lifecycle) {

        //create a root frame
        final IFrameBluePrint frameBp = BPF.frame();
        frameBp.setSize(new Dimension(400, 300)).setTitle("Command actions");
        final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);

        //Create the menu bar
        final IMenuBarModel menuBar = frame.getMenuBarModel();

        //Use a border layout
        frame.setLayout(BorderLayout.builder().gap(0).build());

        //add a toolbar to the top
        final IToolBarModel toolBar = frame.add(BPF.toolBar(), BorderLayout.TOP).getModel();

        //add a composite to the center
        final IComposite composite = frame.add(BPF.composite().setBorder(), BorderLayout.CENTER);
        composite.setLayout(new MigLayoutDescriptor("[grow][]", "[]"));

        //add a input field and save button to the composite
        final IInputField<String> inputField = composite.add(BPF.inputFieldString(), "growx");
        final IButton saveButton = composite.add(BPF.button());

        //create save action
        final IAction saveAction = createSaveAction(inputField);

        //create a menu and add save action
        final MenuModel menu = new MenuModel("File");
        menu.addAction(saveAction);

        //add the menu to the menu bar
        menuBar.addMenu(menu);

        //add the action to the toolbar
        toolBar.addAction(saveAction);

        //bind the action to the save button
        saveButton.setAction(saveAction);

        //set the root frame visible
        frame.setVisible(true);
    }

    private static IAction createSaveAction(final IInputComponent<String> inputComponent) {
        final IActionBuilder builder = Action.builder();
        builder.setText("Save");
        builder.setToolTipText("Saves the text");
        builder.setAccelerator(VirtualKey.S, Modifier.CTRL);
        builder.setIcon(IconsSmall.DISK);

        //save command implements ICommandExecutor and IEnabledChecker,
        //so set them both
        final SaveCommand saveCommand = new SaveCommand(inputComponent);
```

```
58          builder.setCommand(saveCommand, saveCommand);
59
60          return builder.build();
61      }
62
63      private static final class SaveCommand
64          extends AbstractEnabledChecker implements ICommandExecutor, IEnabledChecker {
65
66          private final IInputComponent<?> inputComponent;
67
68          private SaveCommand(final IInputComponent<?> inputComponent) {
69              this.inputComponent = inputComponent;
70
71              inputComponent.addInputListener(new IInputListener() {
72                  @Override
73                  public void inputChanged() {
74                      fireEnabledStateChanged();
75                  }
76              });
77          }
78
79          @Override
80          public void execute(final IExecutionContext executionContext) throws Exception {
81              inputComponent.resetModificationState();
82              fireEnabledStateChanged();
83              final String message = "'" + inputComponent.getValue() + "' saved!";
84              MessagePane.showInfo(executionContext, message);
85          }
86
87          @Override
88          public IEnabledState getEnabledState() {
89              if (!inputComponent.hasModifications()) {
90                  return EnabledState.disabled("No changes to save");
91              }
92              else {
93                  return EnabledState.ENABLED;
94              }
95          }
96
97      }
98
99  }
```

Die Klasse `SaveCommand` implementiert sowohl die Schnittstelle `ICommandExecutor` als auch `IEnabledChecker`. Dies ist im Beispiel einfacher, da sich nach dem Zurücksetzen des Modification State in Zeile 81 auch der EnabledState ändert. In Zeile 82 werden daher die registrierten Listener darüber informiert. Es ist bei diesem Vorgehen zu beachten, dass in Zeile 58 die `saveCommand` Instanz doppelt angegeben wird, einmal als `ICommandExecutor` und einmal als `IEnabledChecker`.

Beide Schnittstellen in einer Klasse zu implementieren ist nicht ungewöhnlich. In manchem Fällen lassen sich aber auch `IEnabledChecker` für unterschiedliche Actions wiederverwenden, was eine Trennung der Implementierung nahelegt.

3.12.5 ActionItemVisibilityAspect

Ein `IActionItemVisibilityAspect` definiert die Sichtbarkeit für eine oder mehrere Action Item Models. Mit Hilfe der folgenden Methode kann auf einem `IActionItemModelBuilder` ein `IActionItemVisibilityAspect` für ein einzelnes Action Item Model gesetzt, bzw. hinzugefügt werden:

```
IActionItemModelBuilder addVisibilityAspect(IActionItemVisibilityAspect visibilityAspect);
```

Mit Hilfe eines `IActionItemVisibilityAspectPlugin` kann ein Aspekt zu allen Action Items Models hinzugefügt werden.

3.12.5.1 Die Schnittstelle IActionItemVisibilityAspect

Die Schnittstelle IActionItemVisibilityAspect hat die folgende Methode:

```
IPriorityValue<Boolean, LowHighPriority> getVisibility(IAction action);
```

Dabei hat ein `IPriorityValue<Boolean, LowHighPriority>` folgenden Methoden:

```
Boolean getValue();

LowHighPriority getPriority();
```

Der `value` definiert, ob die Action sichtbar sein soll oder nicht. Die enum `LowHighPriority` hat zwei mögliche Werte:

```
LOW,

HIGH;
```

Man kann also festlegen, mit welcher Priorität die Action sichtbar sein soll oder nicht. Dies kann hilfreich sein, wenn mehr als ein Aspekt definiert ist.

Zum Beispiel könnte ein Aspekt alle Actions ausblenden wollen, für die der angemeldete Nutzer kein Ausführungsrecht in der Rechteverwaltung besitzt. Ein solcher Aspekte würden für `visible == true` die Priorität `LOW` setzen, was soviel bedeutet wie: “Aus meiner Sicht sichtbar, außer jemand anders sagt mit höherer Priorität `visible=false`. Für `visible == false` würde die Priorität `HIGH` verwendet werden, was soviel bedeutet wie:”Egal was andere nach mir sagen, die Action soll nicht sichtbar sein“.

Hat die Sichtbarkeit von zwei Aspekten beide Male die Priorität `HIGH`, wird der Aspekt berücksichtigt, welcher zuerst hinzugefügt wurde. Beim Action Item Visibility Aspect Plugin kann dafür eine `order` angegeben werden.

Um das obere Beispiel zu erweitern, könnte ein Aspekt die Aufgabe haben, bestimmte Actions immer anzuzeigen, auch wenn man kein Recht zur Ausführung hat (z.B. weil keine Lizenz erworben wurde). Dennoch soll der Nutzer, sozusagen als *Teaser*, sehen dass eine bestimmte Aktion existiert, um den Wunsch zu wecken, eine Lizenz zu erwerben. Ein solcher Aspekt würde für `visible == true` die Priorität `HIGH` und für `visible == false` die Priorität `LOW` verwenden.

3.12.5.2 Enabled State Visibility Aspect

Die Klasse `org.jowidgets.tools.model.item.EnabledStateVisibilityAspect` implementiert einen Visibility Aspekt, welcher alle Actions ausblendet, die nicht enabled sind. Der Source Code soll als Beispiel für die Implementierung eines `IActionItemVisibilityAspect` dienen:

```
public final class EnabledStateVisibilityAspect implements IActionItemVisibilityAspect {

    private static final IPriorityValue<Boolean, LowHighPriority> NOT_VISIBLE_HIGH
        = new PriorityValue<Boolean, LowHighPriority>(Boolean.FALSE, LowHighPriority.HIGH);

    private static final IPriorityValue<Boolean, LowHighPriority> VISIBLE_LOW
        = new PriorityValue<Boolean, LowHighPriority>(Boolean.TRUE, LowHighPriority.LOW);

    @Override
    public IPriorityValue<Boolean, LowHighPriority> getVisibility(final IAction action) {
        if (action != null) {
            final boolean enabled = action.isEnabled();
            if (!enabled) {
                return NOT_VISIBLE_HIGH;
            }
            else {
                return VISIBLE_LOW;
            }
        }
        return VISIBLE_LOW;
    }
}
```

3.12.5.3 Secure Action Item Visiblity Aspect

Der folgende Sichtbarkeitsaspekt stammt aus der jo-client-platform, und versteckt alle Actions, für welche der angemeldete Nutzer kein Ausführungsrecht besitzt[10].

```
private final class SecureActionItemVisibilityAspect implements IActionItemVisibilityAspect {

    @Override
    public IPriorityValue<Boolean, LowHighPriority> getVisibility(final IAction action) {
        final ISecureObject<AUTHORIZATION_TYPE> secureObject
            = WrapperUtil.tryToCast(action, ISecureObject.class);
        if (secureObject != null) {
            if (!authorizationChecker.hasAuthorization(secureObject.getAuthorization())) {
                return NOT_VISIBLE_HIGH;
            }
        }
        return null;
    }
}
```

Das zugehörige Plugin findet sich hier

3.12.5.4 Die Schnittstelle IActionItemVisibilityAspectPlugin

Um ein Visibilitätsaspekt global hinzuzufügen, kann das `IActionItemVisibilityAspectPlugin` verwendet werden. Die Schnittstelle hat die folgenden Methoden:

```
    IActionItemVisibilityAspect getVisibilityAspect();

    int getOrder();
```

[10] Dabei handelt es sich um einen reinen Convenience Aspekt. In der Service Schicht wird eine solche Aktion zusätzlich abgelehnt.

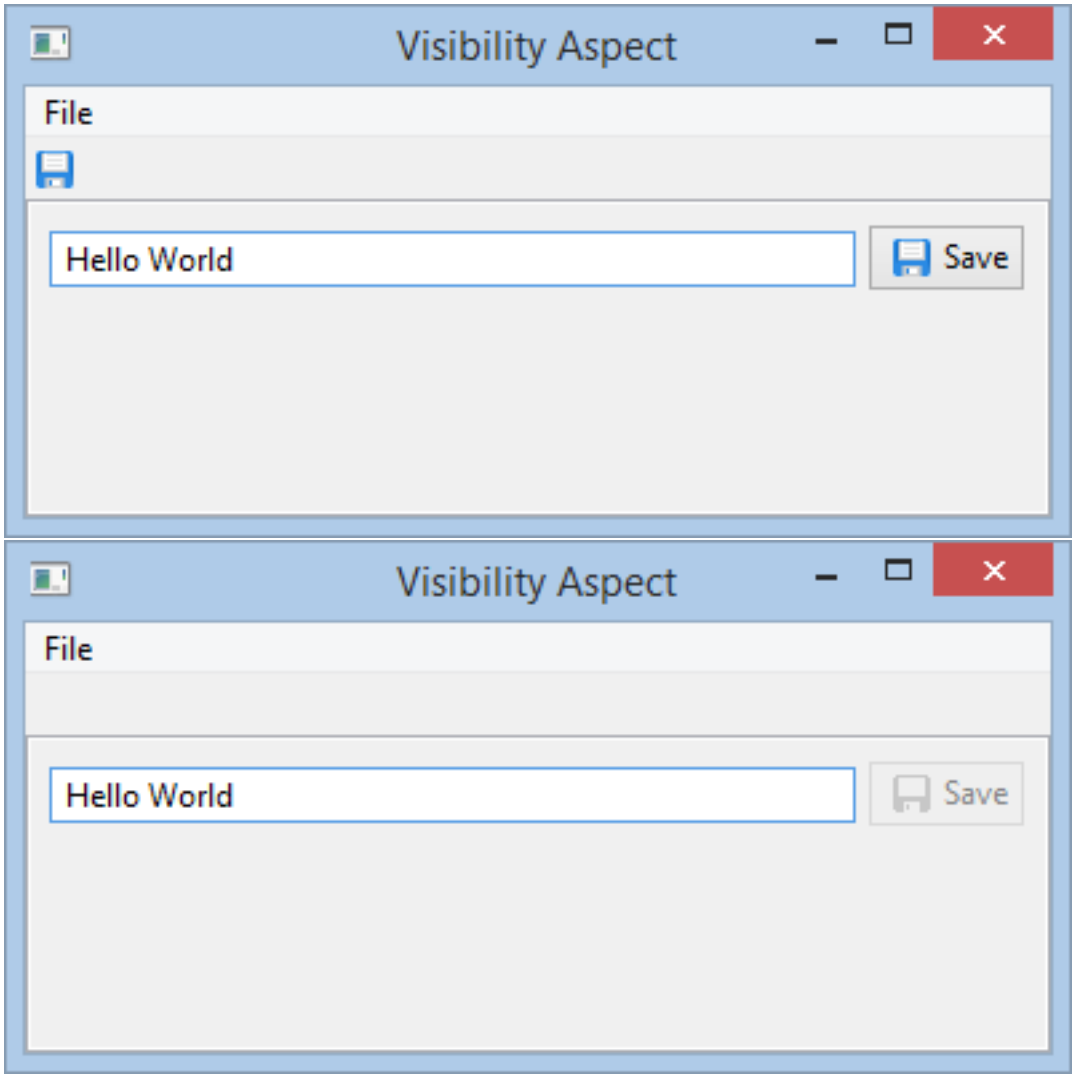
Die erste Methode liefert den Aspekt, die zweite die Order. Alle Plugins werden nach ihrer Order ausgeführt. Um ein Plugin zu registrieren, kann der ServiceLoader Mechanismus verwendet werden, dazu muss eine Datei mit dem Namen `org.jowidgets.api.model.item.IActionItemVisibilityAspectPlugin` in META-INF/services abgelegt werden, welche die Implementierungen auflistet.

Im Command Action Snipped könnte man zum Beispiel das `org.jowidgets.tools.model.item.EnabledStateVisibilityAsp` hinzufügen, dann würde die Save Action nur erscheinen, wenn man speichern kann[11].

Neben der Möglichkeit, das Plugin mit Hilfe eines ServiceLoader zu registrieren, könnte man auch den folgenden Code hinzufügen (z.B. bevor das Frame erzeugt wird):

```
1    ActionItemVisibilityAspectPlugin.registerPlugin(new EnabledStateVisibilityAspectPlugin());
```

Das Ergebnis würde, unabhängig mit welcher Methode das Plugin registriert wurde, dann so aussehen (einmal mit und einmal ohne Save Action):



[11] Dies soll nur der Anschauung dienen. Nach Meinung des Autors ist es besser, den `disabled` Reason anzuzeigen, anstatt den Save Button auszublenden.

Es ist zu beachten, dass der Button nicht automatisch ausgeblendet wurde. Das Ausblenden ist derzeit auf Menüs und Toolbars beschränkt. Um die Funktion selbst zu implementieren, könnte die folgende Methode auf der Klasse `ActionItemVisibilityAspectPlugin` verwendet werden, um die Sichtbarkeit der Action zu erhalten:

```
    public static IPriorityValue<Boolean, LowHighPriority> getVisibility(final IAction action) {...}
```

Diese liefert die Visibilität auf Basis aller registrierten Plugins. Um den Button im Command Action Snipped ebenfalls automatisch auszublenden, könnte man die folgende Methode hinzufügen:

```
1  private void setButtonVisibility(final IButton button, final IAction action) {
2      final IPriorityValue<Boolean, LowHighPriority> visibility
3          = ActionItemVisibilityAspectPlugin.getVisibility(action);
4      if (visibility != null) {
5          button.setVisible(visibility.getValue().booleanValue());
6      }
7      else {
8          button.setVisible(true);
9      }
10 }
```

Und diese wie folgt aufrufen:

```
1      ...
2
3      setButtonVisibility(saveButton, saveAction);
4      saveAction.getActionChangeObservable().addActionChangeListener(new ActionChangeAdapter() {
5          @Override
6          public void enabledChanged() {
7              setButtonVisibility(saveButton, saveAction);
8              frame.layoutLater();
9          }
10     });
11
12     ...
```

3.13 Farben

Farben werden in jowidgets mit Hilfe einer `IColorConstant` gesetzt. Die Schnittstelle hat die folgende Methode:

```
    ColorValue getDefaultValue();
```

Ein `ColorValue` Objekt hat den folgenden Konstruktor:

```
    public ColorValue(final int red, final int green, final int blue) {...}
```

Dabei kann der rot, grün und blau Anteil festgelegt werden. Die Klasse `ColorValue` implementiert die Schnittstelle `IColorConstant` und kann somit zum Setzen von Farben verwendet werden. Das folgende Beispiel soll das demonstrieren:

```
    frame.setBackgroundColor(new ColorValue(7, 106, 3));
```

Die Idee hinter der Schnittstelle `IColorConstant` ist, dass man neben konkreten Farben auch *logische* Farbkonstanten definieren kann. Zudem sollte mit Hilfe einer `IColorRegistry` der `defaultValue` überschrieben werden können. Derzeit ist eine Color Registry jedoch nicht implementiert[12].

Die enum `org.jowidgets.api.color.Colors` definiert einige Farbkonstanten:

```
public enum Colors implements IColorConstant {

    //logical colors

    /**
     * Default foreground color
     */
    DEFAULT(new ColorValue(0, 0, 0)),

    /**
     * Error color
     */
    ERROR(new ColorValue(220, 0, 0)),

    /**
     * Warning color
     */
    WARNING(new ColorValue(209, 124, 34)),

    /**
     * Color for a strong label markup
     */
    STRONG(new ColorValue(0, 70, 213)),

    /**
     * Color for disabled items
     */
    DISABLED(new ColorValue(130, 130, 130)),

    /**
     * Background color for even table rows when using striped rendering
     */
    DEFAULT_TABLE_EVEN_BACKGROUND_COLOR(new ColorValue(222, 235, 235)),

    /**
     * Color for selected backgrounds
     */
    SELECTED_BACKGROUND(new ColorValue(16, 63, 149)),

    //named colors

    BLACK(new ColorValue(0, 0, 0)),

    WHITE(new ColorValue(255, 255, 255)),

    DARK_GREY(new ColorValue(80, 80, 80)),

    GREY(new ColorValue(140, 140, 140)),

    LIGHT_GREY(new ColorValue(225, 225, 225)),

    GREEN(new ColorValue(7, 106, 3));

```

[12] Wobei es sich um eine Erweiterung handeln würde, die sich relativ einfach umsetzen ließe und nicht invasiv wäre.

```
54      private ColorValue colorValue;
55
56      private Colors(final ColorValue colorValue) {
57          this.colorValue = colorValue;
58      }
59
60      @Override
61      public ColorValue getDefaultValue() {
62          return colorValue;
63      }
64
65  }
```

Diese werden hauptsächlich von jowidgets Widgets und von jo-client-platform Widgets verwendet. Nach dem gleichen Schema könnte man sich eigene Color Enums erstellen, zum Beispiel für die Verwendung in einer applikations- oder firmeninternen Bibliothek.

Mit Hilfe der `Colors` Klasse könnte man die Hintergrundfarbe der Frames auch wie folgt setzen:

```
    frame.setBackgroundColor(Colors.GREEN);
```

Mit Hilfe des ColorTableSnipped werden alle Colors in einer Tabelle angezeigt:

3.13.1 Farben unter SWT

Unter SWT werden für Farben Systemresourcen verwendet. Diese werden mit Hilfe des SWT Display erzeugt, und sollten disposed werden, sobald man sie nicht mehr benötigt.

Für ein `ColorValue` wird erst dann eine Systemresource erzeugt, wenn sie tatsächlich benutzt wird und wenn die gleiche Farbe nicht bereits existiert. Dazu wird ein Color Cache verwendet, welcher für `ColorValue` Keys SWT Farben liefert. Um für ein `ColorValue` Objekt eine *gecachte* SWT Farbe aus dem Cache zu entfernen und anschließend zu disposen, kann man auf dem `ColorValue` Objekt die Methode

```
    void release();
```

aufgerufen werden. Dies ist aber nur dann wirklich sinnvoll, wenn viele unterschiedliche Farben angelegt werden, welche man später nicht mehr benötigt, wie zum Beispiel für einen Color Chooser.

Die Farben der weiter vorne beschriebenen Klasse `Colors` nach deren Verwendung wieder zu *releasen* ist zwar möglich, bringt aber nicht wirklich eine nennenswerte Speichereinsparung (alle 13 Farben benötigen zusammen grob überschlagen 130 Byte).

Das folgende Beispiel zeigt eine Animation bei der 32768 unterschiedliche Farben verwendet werden. Diese würden ca. 320 kByte benötigen, weshalb sie direkt nach deren Verwendung wieder aus dem Cache released werden (Zeile 19):

```
1       final IAnimationRunner animationRunner = AnimationRunner.create();
2       animationRunner.setDelay(20, TimeUnit.MILLISECONDS);
3
4       final Runnable animationStep = new Runnable() {
5
6           private int red = 0;
7           private int redIncrement = 1;
8
```

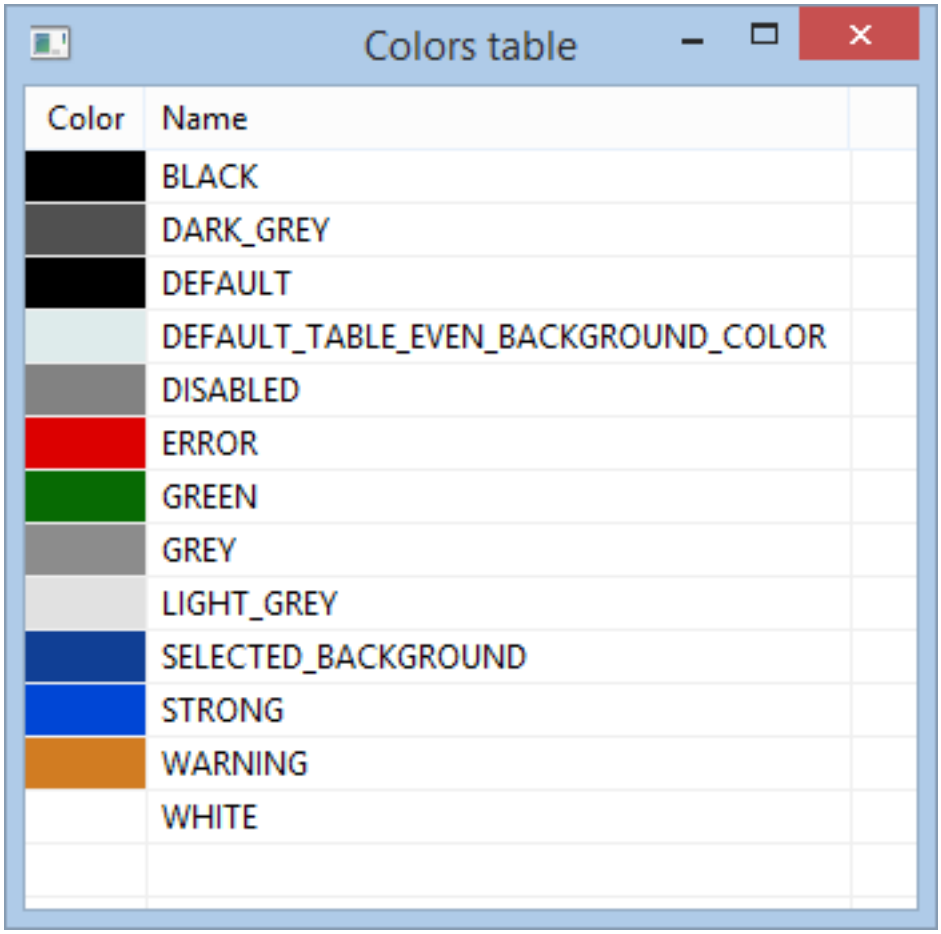


Abbildung 3.28: ColorTableSnipped

```
9         private int green = 0;
10        private int greenIncrement = 2;
11
12        private int blue = 0;
13        private int blueIncrement = 3;
14
15        @Override
16        public void run() {
17            final ColorValue color = new ColorValue(red, green, blue);
18            frame.setBackgroundColor(color);
19            color.release();
20
21            red = red + redIncrement;
22            if (red >= 255) {
23                red = 255;
24                redIncrement = redIncrement * -1;
25            }
26            else if (red <= 0) {
27                red = 0;
28                redIncrement = redIncrement * -1;
29            }
30
31            green = green + greenIncrement;
32            if (green >= 255) {
33                green = 255;
34                greenIncrement = greenIncrement * -1;
35            }
36            else if (green <= 0) {
37                green = 0;
38                greenIncrement = greenIncrement * -1;
39            }
40
41            blue = blue + blueIncrement;
42            if (blue >= 255) {
43                blue = 255;
44                blueIncrement = blueIncrement * -1;
45            }
46            else if (blue <= 0) {
47                blue = 0;
48                blueIncrement = blueIncrement * -1;
49            }
50
51            animationRunner.run(this);
52        }
53    };
54
55    animationRunner.start();
56    animationRunner.run(animationStep);
```

3.14 Images und Icons

Die folgende Abschnitte liefern eine Übersicht über die Verwendung Images und Icons.

Es gibt zum einen die Möglichkeit, Images von der `IImageFactory` erzeugen zu lassen, was dem Vorgehen klassischer UI Frameworks entspricht.

Zudem bietet jowidgets auch die Möglichkeit, Image Konstanten zu definieren, welche über eine Image Registry registriert werden. Dadurch wird sicher gestellt, dass ein Image für eine Konstante nur ein Mal erzeugt wird, unabhängig davon, wie oft man die Konstante verwendet. Die Definition der Konstanten und die Registrierung konkreter Images kann dabei in unterschiedlichen Modulen erfolgen, so dass

zum Beispiel für unterschiedliche Implementierungen einer API auch unterschiedliche Icons verwendet werden können.

Zudem ist es möglich, Images zu einer Image Konstante beliebig auszutauschen. In Software Unternehmen werden oft kommerzielle Icon Bibliotheken eingesetzt, welche von professionellen Grafik Designern erstellt wurden. Jowidgets verwendet für einige Widgets bereits Icon Konstanten als Default Icon. [13] Das Validation Label verwendet z.B. das Icon `OK` als Default Icon für den MessageType OK. Hat man nun in einer firmeninternen Icon Bibliothek ein *besseres* Ok Icon, oder will man aus Gründen der Konsistenz, dass überall das gleiche Ok Icon verwendet wird, kann man das Icon einfach *umregistrieren*. [14]

Mit Hilfe von Image Providern ist auch eine implizite Registrierung von Images möglich. Im Abschnitt: Eigene Icon Bibliotheken mit Hilfe von Image Provider Enums wird gezeigt, wie dadurch sehr einfach eigene Icon Bibliotheken erstellt werden können.

3.14.1 Image Konstanten

Images werden in jowidgets mit Hilfe einer Image Konstante auf einem Widget gesetzt. Die Schnittstelle `IImageConstant` ist die Basisschnittstelle für alle Image Konstanten. Sie hat selbst keine Methoden. [15]

Eine Image Konstante muss über die Image Registry verfügbar sein, um verwendet werden zu können. Dies ist der Fall, wenn:

- Für die Image Konstante explizit ein Image Handle in der Image Registry registriert wurde.
- Sie ein selbstbeschreibender ImageProvider ist.
- Sie implizit von der Image Factory beim Erzeugen eines Images registriert wurde.

3.14.1.1 Beispiel für die Verwendung von Enum Konstanten

Die folgenden Beispiele demonstriert die Verwendung vom Image Konstanten, welche durch Enums definiert sind:

```
label.setIcon(IconsSmall.OK);

button.setIcon(SilkIcons.HELP);

tabItem.setIcon(SilkIcons.APPLICATION);

actionBuilder.setIcon(IconsSmall.DISK);
```

Ein expliziter Zugriff auf die Image Registry ist bei der Verwendung nicht notwendig. Die Adapter Klassen der jeweiligen SPI Implementierung verwenden die Konstante, um aus der Image Registry das registrierte Image Handle zu erhalten.

[13] Die konkreten Default Icons dafür wurden jedoch nicht von einem professionellen Grafik Designer, sondern von einem Informatiker entworfen, zumindest hat der Ersteller der Default Icons, welcher auch gleichzeitig Autor dieses Textes ist, kein Grafikdesign studiert :-)

[14] Man könnte im konkreten Fall das gleiche auch erreichen, indem man die Widget Defaults des Validation Label global überschreibt, allerdings könnte es sein, dass ein Icon von mehreren Widgets (und vielleicht sogar innerhalb der firmeneigenen Widgets) verwendet wird. Das Edit Icon wird zum Beispiel von mehreren Actions der jo-client-platform sowie vom CollectionInputField verwendet.

[15] Die Schnittstelle sollte treffender `IImageKey` heißen. Aufrund der zahlreichen Verwendung von Images in abgeleiteten Projekten wurde bisher jedoch von einem Refactoring abgesehen.

3.14.1.2 Image Konstanten und die ICacheable Schnittstelle

Wenn eine Image Konstante die Schnittstelle `ICacheable` implementiert, wird diese aus der Image Registry entfernt und das zugehörige Image Handle disposed, sobald auf dem `ICacheable` die Methode:

```
void release();
```

aufgerufen wird. Das sollte jedoch nur dann gemacht werden, wenn man sich sicher ist, **dass die Konstante aktuell nicht verwendet wird**. Das *Disposen* von nativen Images, welche in Verwendung sind, kann andernfalls zu Fehlern führen.

3.14.2 Image Handle

Ein Image Handle stellt das native (UI Framework abhängige) Image zur Verfügung. Des Weiteren kennt ein Image Handle seinen Image Descriptor wenn es mit Hilfe eines solchen erzeugt wurde. Die Schnittstelle `IImageHandle` hat die folgenden Methoden:

```
Object getImage();

boolean isInitialized();

IImageDescriptor getImageDescriptor();
```

Die Methode `getImage()` liefert das native Image Objekt. Der Typ kann je nach SPI Implementierung variieren. Das Image wird *lazy*, also beim ersten Aufruf der Methode `getImage()` erzeugt. Durch das Registrieren eines Image Handle in der Image Registry mit Hilfe eines Image Descriptor wird daher noch **keine** Image Resource erzeugt. Die Methode `getImage()` liefert nie `null` zurück. Es könnte jedoch eine Exception geworfen werden, wenn das Image nicht erzeugt werden kann. Dies passiert folglich auch nicht beim Registrieren, sondern bei der ersten Verwendung. Durch die *lazy* Erzeugung von Resourcen kann man große Icon Bibliotheken mit vielen Icons einbinden, und benötigt zur Laufzeit nur den Speicher für die Icons, die man tatsächlich verwendet.

Will man wissen, ob das Image bereits erzeugt wurde, kann man dies mit Hilfe der Methode `isInitialized()` prüfen.

Image Handles, welche mit Hilfe eines Image Descriptor erstellt wurden, liefern diesen mit Hilfe der Methode `getImageDescriptor()` zurück. Die Betriebssystem Message Icons und Buffered Images werden zum Beispiel nicht mit Hilfe eines Image Descriptors erstellt, weshalb die Methode dort `null` zurückliefert.

Normalerweise benötigt man Image Handles nicht direkt. Sie werden indirekt von der SPI Implementierung bei der Image Registry angefragt.

3.14.3 Image Descriptor

Mit Hilfe eines Image Descriptor kann ein Image Handle erzeugt werden. Normalerweise registriert man in der Image Registry nicht direkt ein Image Handle, sondern verwendet für die Registrierung zum Beispiel einen Image Descriptor. Derzeit werden zwei Descriptor Typen unterstützt, `IUrlImageDescriptor` und `IStreamFactoryImageDescriptor`. [16]

[16] Die Schnittstelle `IStreamImageDescriptor` wird auch unterstützt, ist aber wegen des missverständlichen Methodennamens `getInputStream()` deprecated und soll langfristig entfernt werden.

3.14.3.1 Url Image Descriptor

Die Schnittstelle `IUrlImageDescriptor` hat die folgende Methode:

```
URL getImageUrl();
```

Diese liefert die URL, über welche man das Image einlesen kann. Die Methode darf nicht `null` zurückgeben. Die Klasse `UrlImageDescriptor` liefert eine Implementierung der Schnittstelle `IUrlImageDescriptor`.

Das folgende Beispiel zeigt eine Methode, welche eine Image Konstante mit Hilfe eines `UrlImageDescriptor` in der Image Registry registriert:

```
1  final void registerImage(final IImageConstant imageConstant, final String relativePath) {
2      final String path = rootPath + relativePath;
3      final URL url = getClass().getClassLoader().getResource(path);
4      final IImageDescriptor descriptor = new UrlImageDescriptor(url);
5      Toolkit.getImageRegistry().registerImageConstant(imageConstant, descriptor);
6  }
```

3.14.3.2 Stream Factory Image Descriptor

Die Schnittstelle `IStreamFactoryImageDescriptor` hat die folgende Methode:

```
InputStream createInputStream();
```

Der Input Stream wird genau dann erzeugt, wenn ein natives Image benötigt wird. Jeder Aufruf muss einen neuen Input Stream liefern, der insbesondere nicht `null` sein darf. Der Stream wird vom Aufrufer der Methode geschlossen (`close()`), wenn er nicht mehr benötigt wird.

3.14.4 Image Provider

Ein Image Provider ist eine Image Konstante und ein Image Descriptor zugleich. Ein Image Provider muss nicht explizit in der Image Registry registriert werden. Bei der ersten Verwendung wird mit Hilfe des Image Descriptor Anteil ein Image Handle erzeugt und für die Image Konstante registriert. Dadurch spart man sich zum Beispiel das Halten von Schlüsseln in der Image Registry für Icons, welche gar nicht zur Laufzeit verwendet, aber in einer Icon Bibliothek vorhanden sind. Derzeit werden zwei Image Provider Typen unterstützt, der `IImageUrlProvider` und der `IImageStreamFactoryProvider`.

3.14.4.1 Image Url Provider

Die Schnittstelle `IImageUrlProvider` hat die folgende Methode:

```
URL getImageUrl();
```

Die Klasse `ImageUrlProvider` implementiert diese Schnittstelle und zudem auch die Schnittstelle `ICacheable`, wodurch ein `ImageUrlProvider` mittels `release()` aus der Image Registry entfernt werden kann. Das folgende Beispiel demonstriert die Verwendung der Klasse `ImageUrlProvider`:

```
1  final ImageUrlProvider icon = new ImageUrlProvider(iconFile);
2
3  button1.setIcon(icon);
4  button2.setIcon(icon);
5
6  ...
7
8  button1.dispose();
9  button2.dispose();
10
11 //release the icon from the image registry if no longer used
12 icon.release();
```

Der `ImageUrlProvider` wird in Zeile 1 erzeugt und in Zeile 12 wieder released. Da es sich um einen Image Provider handelt, ist ein explizites registrieren in der Image Registry nicht notwendig. Für beide Widgets wird nur ein Image Handle erzeugt, welches bei der ersten Verwendung automatisch in der Image Registry registriert wird. **Man muss sich selbst darum kümmern, die Konstante, wenn sie nicht mehr benötigt wird, aus der Image Registry zu entfernen**. Mit dem folgenden Code würde man das gleiche wie mit `icon.release()` erreichen:

```
Toolkit.getImageRegistry().unRegisterImageConstant(icon);
```

Anmerkung: Gemeinsam genutzte Icons sollten besser in einer dauerhaft gültigen Konstante, wie zu Beispiel einer Enum gehalten, und nur in Ausnahmefällen disposed werden. Durch das Disposen von Icons welche man mehrfach verwendet, können Fehler auftreten, welche unter Umständen nur schlecht reproduzierbar sind. Da man in der Regel (auch in großen Applikationen) nicht Unmengen von unterschiedlichen Icons verwendet, und diese somit auch nicht unverhältnismäßig viel Speicher benötigen, könnte man das Disposen von momentan nicht verwendeten Icons unter Umständen als *verfrühte Optimierung (premature optimization)* im Sinne von Knuth betrachten. Es gibt jedoch Fälle, wo es sinnvoll sein mag aufzuräumen, zum Beispiel wenn mit Hilfe eines Icon Chooser alle verfügbaren Icons einer Bibliothek angezeigt und somit auch geladen wurden. Hier ist es durchaus sinnvoll, diese Icons anschließend wieder aus der Image Registry zu entfernen.

3.14.4.2 Image Stream Factory Provider

Die Schnittstelle `IImageStreamFactoryProvider` hat die folgende Methode:

```
InputStream createInputStream();
```

Der Input Stream wird genau dann erzeugt, wenn ein natives Image benötigt wird. Jeder Aufruf muss einen neuen Input Stream liefern, der insbesondere nicht `null` sein darf. Der Stream wird vom Aufrufer der Methode geschlossen (`close()`), wenn er nicht mehr benötigt wird.

3.14.5 Die Image Registry

Die Image Registry liefert für eine Image Konstante ein Image Handle. In der Regel fragt man das Image Handle nicht selbst ab, sondern verwendet eine Image Konstante zur Definition eines Images oder Icons für ein Widget. Die Adapter Klassen in der SPI Implementierung lösen dann mit Hilfe der Image Registry diese Image Konstante in ein Image Handle auf.

3.14.5.1 Image Registry Instanz

Eine Image Registry Instanz bekommt man vom Toolkit mittels der folgenden Methode:

```
IImageRegistry getImageRegistry();
```

Es folgt eine kurze Auflistung der wichtigsten Methoden der Schnittstelle `IImageRegistry`:

3.14.5.2 Methoden zur Registrierung und zum Austauschen von Image Handles

Die folgenden Methoden dienen der Registrierung eines Image Handle zu einer Image Konstante:

```
1  void registerImageConstant(IImageConstant key, IImageHandle imageHandle);
2
3  void registerImageConstant(IImageConstant key, IImageDescriptor descriptor);
4
5  void registerImageConstant(IImageConstant key, URL url);
6
7  void registerImageConstant(IImageConstant key, IImageConstant substitude);
8
9  void registerImageConstant(IImageConstant key, IImageProvider provider);
```

Die erste Methode macht die Registrierung explizit. Diese wird normalerweise nur von der SPI Implementierung verwendet, welche das Image Handle erzeugt, zum Beispiel mit Hilfe eines Image Descriptors, welchen man mit der Methode in Zeile 3 übergeben kann.

Die Methode in Zeile 5 bietet eine verkürzte Schreibweise zur Verwendung eines UrlImageDescriptor.

Die nächste Methode in Zeile 7 ersetzt ein Image durch ein anderes, welches durch eine Image Konstante (`substitude`) definiert ist. Die Image Konstante für das `substitude` muss zum Zeitpunkt des Aufrufs available sein, was bedeutet, dass bereits ein Image Handle dafür registriert sein muss. [17]. Es ist für die Ausführung der Methode nicht notwendig, dass für den Quell `key` bereits ein Handle registriert ist.

Die letzte Methode in Zeile 9 ersetzt auch eine vorhandene Registrierung. Dazu wird bei Bedarf vorab der Image Provider `provider` (mit seinem eigenen Schlüssel) registriert, wenn er noch nicht vorhanden ist. Das Image, welches der Provider liefert ist dadurch sowohl für die Image Konstante `key` als auch für die Image Konstante des `provider` verfügbar. Es ist für die Ausführung der Methode nicht notwendig, dass für den Quell `key` bereits ein Handle registriert ist.

3.14.5.3 Austauschen von Images

Mit allen oben aufgeführten Methoden kann auch ein bereits registriertes Image Handle durch ein anderes ersetzt (substituiert) werden, jedoch nur, wenn dieses nicht bereits initialisiert wurde (nicht initialisierte aber registrierte oder verfügbare Handles stellen kein Problem dar).

Würde man ein bereits initialisiertes Image aus der Registry entfernen (indem man ein neues Image Handle für den `key` setzt), ohne das bisherige, initialisierte Handle vorab zu disposen, würde man dadurch ein potentielles Speicherleck provozieren, da zum Beispiel unter SWT das native Image nicht mehr disposed werden würde, wodurch das SWT Display eine Referenz darauf behält. Würde man das alte Handle automatisch disposen, würde man einen potentiellen Folgefehler provozieren, dessen Ursache nur schwer zu finden ist, da er sich erst später auswirken kann. Man kann wie folgt prüfen, ob ein Image Handle für eine Image Konstante initialisiert ist:

[17] Oder die Image Konstante des `substitude` muss ein selbstbeschreibender Image Provider sein. Wird dieser jedoch nicht explizit in eine `IImageConstant` *gecastet*, wird in diesem Fall die Signatur in Zeile 9 verwendet.

```
    if (!registry.isImageInitialized(IconsSmall.DISK)) {
        registry.registerImageConstant(IconsSmall.DISK, SilkIcons.DISK);
    }
```

Hinweis: Um das Problem grundsätzlich zu vermeiden, wird empfohlen, Registrierungen und Substituierungen immer mit Hilfe eines Toolkit Interceptor durchzuführen. Dadurch ist sichergestellt, dass die Registrierung stattfindet, bevor ein Image initialisiert werden kann.

Das folgende Beispiel zeigt, wie logische Icons einer Bibliothek durch konkrete ersetzt werden:

```
1   registry.registerImageConstant(IconsSmall.OK, IconLib_16x16.TICK);
2   registry.registerImageConstant(IconsSmall.EDIT, IconLib_16x16.PENCIL);
3   registry.registerImageConstant(IconsSmall.REFRESH, IconLib_16x16.RECYCLE);
4   registry.registerImageConstant(IconsSmall.UNDO, IconLib_16x16.ARROW_UNDO);
5   registry.registerImageConstant(IconsSmall.CANCEL, IconLib_16x16.CANCEL);
6   registry.registerImageConstant(IconsSmall.ERROR, IconLib_16x16.ERROR);
```

3.14.5.4 Zugriff auf ein Image Handle

Die folgende Methode liefert ein Image Handle für eine Image Konstante:

```
    IImageHandle getImageHandle(IImageConstant key);
```

Diese Methode wird in der Regel nur von der SPI Implementierung verwendet, um für eine Image Konstante das registrierte Image Handle zu erhalten.

3.14.5.5 Deregistrierung von Objekten

Die folgenden Methoden können verwendet werden, um Einträge aus der Image Registry zu entfernen.

```
    void unRegisterImageConstant(IImageConstant key);

    void unRegisterImage(IImageCommon image);
```

Dabei werden auch immer die zugehörigen Image Handles disposed, falls diese bereits initialisiert wurden. Man sollte daher nur Images deregistrieren, wenn diese nicht mehr verwendet werden.

Die Methode `unregisterImageConstant()` kann auch für Images verwendet werden.

3.14.5.6 Der Available Status

Mit Hilfe der folgenden Methode kann geprüft werden, ob eine Image Konstante verfügbar ist:

```
    boolean isImageAvailable(IImageConstant key);
```

Dies ist der Fall, wenn entweder ein Image Handle für den `key` registriert, oder die Konstante ein IImageProvider ist.

3.14.5.7 Der Registered Status

Mit Hilfe der folgenden Methode kann geprüft werden, ob für eine Image Konstante ein Image Handle registriert wurde:

```
boolean isImageRegistered(IImageConstant key);
```

Hinweis: Wenn die Methode `false` zurückliefert, folgt daraus **nicht**, dass der `key` auch nicht `available` ist (da es sich um einen `IImageProvider` handeln könnte).

3.14.5.8 Der Initialized Status

Mit Hilfe der folgenden Methode kann geprüft werden, ob für eine Image Konstante ein initialisiertes Image Handle existiert:

```
boolean isImageInitialized(IImageConstant key);
```

Wird `true` zurückgegeben, kann das Image nicht mehr umregistriert werden, ohne es vorher aus der Image Registry zu entfernen.

Hinweis: Die Methode führt die Prüfung durch, ohne dabei einen Image Provider implizit zu registrieren. Der folgende Aufruf ist daher **keine geeignete Alternative**:

```
1  IImageHandle imageHandle = registry.getImageHandle(SilkIcons.ACCEPT);
2  if (imageHandle.isInitialized()){
3      //do something
4  }
```

Durch den Aufruf in Zeile 1 wird, wenn noch kein Image Handle existiert, einer erzeugt, da die Image Konstanten der Silk Icons Bibliothek Image Provider sind.

3.14.6 Icon Bibliotheken

In großen Projekten spart der Einsatz bereits vorhandener Icon Bibliotheken Zeit und somit Geld. Zudem wird gefördert, dass nicht jeder Entwickler eine eigene Instanz eines Image Files erstellt. Man kann (muss aber nicht :-) bei Icon Bibliotheken zwischen logischen und konkreten unterscheiden.

3.14.6.1 Logische Icon Bibliotheken

Logische Icons beschreiben eher wofür das Icon eingesetzt wird, konkrete Icons eher das, was das Icon abbildet, wobei diese Regel nur als grober Leitfaden, den man nicht zu dogmatisch betrachten sollte, zu verstehen ist. Der logischen Konstante `Icons.EDIT` könnte man dann zum Beispiel das konkrete Icon `IconLib_16x16.PENCIL` als Default zuweisen und der logischen Konstante `Icons.Save` das konkrete Icon `IconLib_16x16.DISK`. Logische Icon Bibliotheken liefern eher genau die Icons Konstanten, welche für einen bestimmten Modul-, Produkt- oder Firmenkontext relevant sind, inklusive eines konkreten Default Icon dafür.

Verwendet man bei der Erstellung eigener Widget Bibliotheken logische Konstanten, können die Default Icons später leicht durch andere Icons ausgetauscht werden.

Würde man zum Beispiel für den Edit Button in einem firmeninternen Widget die Konstante `IconLib_16x16.PENCIL` verwenden, anstatt `OdzIcons.EDIT` [18], dann könnte man dieses nicht so einfach umdefinieren. Angenommen in einem Produkt sollen anstatt der `IconLib` die `SmartIcons` verwendet werden, dann würde der folgende Code

```
1     registry.registerImageConstant(IconLib_16x16.PENCIL, SmartIcons_16x16.PENCIL);
```

alle Pencils ändern. Hat man im gleichen Produkte auch eine Komponente zum Zeichnen mit verschiedenen Stiften, könnte dies zu einem unerwünschten Nebeneffekt führen.

Mit dem folgenden Code:

```
1     registry.registerImageConstant(OdzIcons.EDIT, SmartIcons_16x16.PENCIL);
```

würde man nur die Stellen ändern, wo es um Editierung geht.

Der Name der weiter unten beschriebenen Enum `IconsSmall` wurde bewusst so und nicht etwa `Icons_16x16` gewählt. Für bestimmte SPI Implementierungen wie zum Beispiel RWT könnte es durchaus Sinn machen, für diese Icons eine andere Auflösung zu wählen, zum Beispiel wenn man per StyleSheet alles etwas größer darstellt.

3.14.6.2 Konkrete Icon Bibliotheken

Konkrete Icon Bibliotheken enthalten eher eine große Auswahl vieler unterschiedlicher Icons und dienen dadurch sowohl als Pool möglicher Default Icons bei der Erstellung logischer Icon Bibliotheken, also auch als mögliche Substitute für das Ersetzen der logischen Konstanten verschiedenster Module, zum Beispiel auch zur Vereinheitlichung des Look And Feel.

Im Vergleich zu logischen Icons macht es für konkrete Icons durchaus Sinn, die zugehörigen Enums oder Konstanten entsprechend der Auflösung zu benennen, insbesonderer wenn verschiedene Auflösungen verfügbar sind. Für einen SWT oder Swing Client könnte man folgende Icons der Enum `IconsSmall` dann zum Beispiel wie folgt Substituieren:

```
1     registry.registerImageConstant(IconsSmall.OK, IconLib_16x16.TICK);
2     registry.registerImageConstant(IconsSmall.EDIT, IconLib_16x16.PENCIL);
3     registry.registerImageConstant(IconsSmall.REFRESH, IconLib_16x16.RECYCLE);
4     registry.registerImageConstant(IconsSmall.UNDO, IconLib_16x16.ARROW_UNDO);
5     registry.registerImageConstant(IconsSmall.CANCEL, IconLib_16x16.CANCEL);
6     registry.registerImageConstant(IconsSmall.ERROR, IconLib_16x16.ERROR);
```

Für einen Web Client mit *luftigem* Layout wie folgt:

```
1     registry.registerImageConstant(IconsSmall.OK, IconLib_24x24.TICK);
2     registry.registerImageConstant(IconsSmall.EDIT, IconLib_24x24.PENCIL);
3     registry.registerImageConstant(IconsSmall.REFRESH, IconLib_24x24.RECYCLE);
4     registry.registerImageConstant(IconsSmall.UNDO, IconLib_24x24.ARROW_UNDO);
5     registry.registerImageConstant(IconsSmall.CANCEL, IconLib_24x24.CANCEL);
6     registry.registerImageConstant(IconsSmall.ERROR, IconLib_24x24.ERROR);
```

[18] Odz steht hier für ein (fast:-) frei erfundenes Kürzel der Firma, welches die Icon Bibliothek definiert hat.

3.14.7 Eigene Icon Bibliotheken mit Hilfe von IImageUrlProvider Enums

Das folgende Beispiel zeigt, wie man sich eine eigenen Icon Bibliothek mit Hilfe einer Image Provider Enum erstellen kann:

```
import java.net.URL;
import org.jowidgets.common.image.IImageUrlProvider;

public enum SilkIcons implements IImageUrlProvider {

    ACCEPT(getIconsPath() + "accept.png"),
    ADD(getIconsPath() + "add.png"),
    ANCHOR(getIconsPath() + "anchor.png"),
    APPLICATION(getIconsPath() + "application.png"),
    APPLICATION_ADD(getIconsPath() + "application_add.png"),

    //...removed most of the icon constants in this code example

    XHTML_GO(getIconsPath() + "xhtml_go.png"),
    XHTML_VALID(getIconsPath() + "xhtml_valid.png"),
    ZOOM(getIconsPath() + "zoom.png"),
    ZOOM_IN(getIconsPath() + "zoom_in.png"),
    ZOOM_OUT(getIconsPath() + "zoom_out.png");

    private static final String ICONS_PATH
        = "org/jowidgets/addons/icons/silkicons/icons/silkicons/";

    private final URL url;

    private SilkIcons(final String defaultPath) {
        this.url = getClass().getClassLoader().getResource(defaultPath);
    }

    private static String getIconsPath() {
        return ICONS_PATH;
    }

    @Override
    public URL getImageUrl() {
        return url;
    }

}
```

Die Enum implementiert die Schnittstelle `IImageUrlProvider`, wodurch die Konstanten vor deren Verwendung nicht explizit registriert werden müssen.

Die zugehörigen Icon Dateien sind in diesem Beispiel im `resources` Ordner des Moduls unter dem Pfad: `org/jowidgets/addons/icons/silkicons/icons/silkicons/` abgelegt, wie es die folgende Abbildung verdeutlicht:

Das folgende Beispiel zeigt, wie die oben definierten Icons verwendet werden können:

```
    final IButton zoomInButton = container.add(BPF.button().setIcon(SilkIcons.ZOOM_IN));
    final IButton zoomOutButton = container.add(BPF.button().setIcon(SilkIcons.ZOOM_OUT));
```

Dieser Ansatz eignet sich sowohl für die Erstellung logischer- als auch für die Erstellung konkreter Icon Bibliotheken.

Anmerkung: Beinhaltet eine konkreten Icon Bibliothek alle Icons einer unter Umständen größeren Icon Sammlung, kann es hilfreich sein, die Enum Klasse mit Hilfe eines Skriptes zu generieren. Für die Silk Icons Bibliothek wurde das so gemacht.

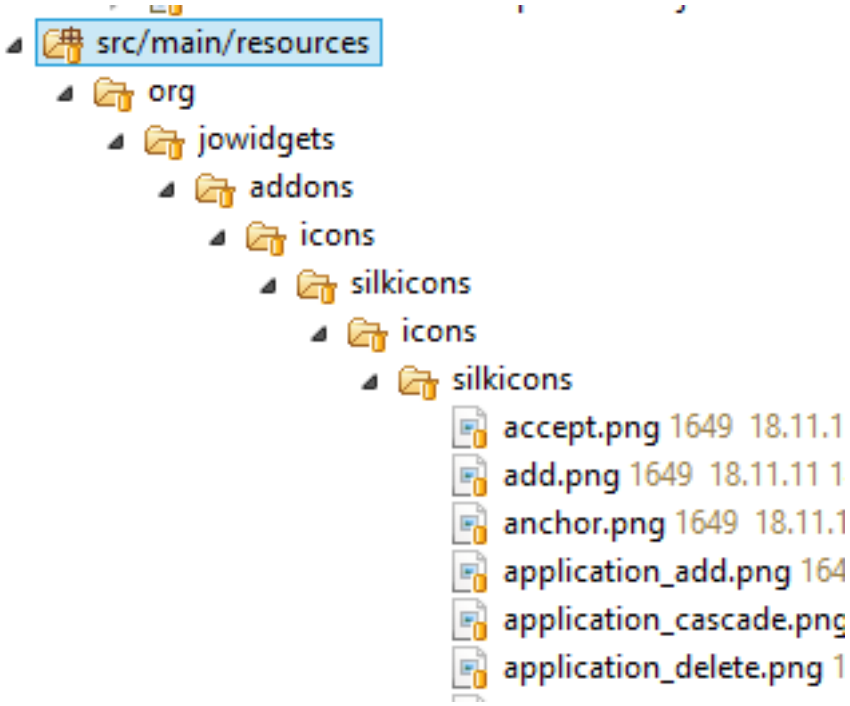


Abbildung 3.29: Icon Resources

3.14.8 Eigene Icon Bibliotheken - Trennung von API und Implementierung

Im folgenden soll beschrieben werden, wie man eine Icon Bibliothek erstellt, bei der die Definition der Image Konstanten (API) und die Bereitstellung der konkreten Icons (Implementierung) getrennt ist.

Dieser Ansatz eignet sich sowohl für die Erstellung logischer-, als auch für die Erstellung konkreter Icon Bibliotheken. Die Trennung könnte wie folgt motiviert sein:

- **Aufteilung API und Implementierung bei der Erstellung einer Widget Bibliothek:** Man möchte die API und die Implementierung in unterschiedliche Module packen, eventuell weil es mehrere Implementierungen gibt. Dann gehört die Icon Konstante zur API, das konkrete Icon zur Implementierung.
- **Einsparung von Resourcen im ausgelieferten Produkt:** Eine Icon API beinhaltet sehr viele Icons einer großer Icon Sammlung. Mehrere Produkte verwenden diese Icon API, welche nur Konstanten enthält, um einen schnellen Zugriff auf die möglichen Icons zu haben. Für ein konkretes Produkt wird dann ein konkretes Plugin erstellt, welches nur die Icons registriert, welche in dem Produkt tatsächlich verwendet werden. Die Menge der verwendeten Icons und somit auch das Plugin könnte man mit einem Tool erstellen (Source Code Generierung).

3.14.8.1 Definition der Icon Konstanten

Das folgende Beispiel zeigt die Definition der Icons für eine Audio Player Widget Bibliothek.

```
1 public enum AudioIcons implements IImageConstant {
2
```

```
    PLAY,
    STOP,
    PAUSE,
    FORWARD,
    REWIND,

    NEXT,
    PREVIOUS,
    FIRST,
    LAST,

    MUTED,
    NOT_MUTED,

    LOUDSPEAKER,
    EDIT_AUDIO_SETTINGS

}
```

Die Icon Konstanten können dann für das Default Setup der Widgets und / oder innerhalb der Implementierung verwendet werden.

3.14.8.2 Registrierung der konkreten Icons

Die folgende Klasse verwendet die Klasse `AbstractResourceImageInitializer` für die Registrierung der konkreten Icons:

```
package org.mycompany.audio.impl;

import org.jowidgets.common.image.IImageRegistry;
import org.jowidgets.tools.image.AbstractResourceImageInitializer;

public final class AudioIconsInitializer extends AbstractResourceImageInitializer {

    public AudioIconsInitializer(final IImageRegistry registry) {
        super(AudioIconsInitializer.class, registry, "org/mycompany/audio/impl/icons/");
    }

    @Override
    public void doRegistration() {
        registerResourceImage(AudioIcons.PLAY, "play.png");
        registerResourceImage(AudioIcons.STOP, "stop.png");
        registerResourceImage(AudioIcons.PAUSE, "pause.png");
        registerResourceImage(AudioIcons.FORWARD, "forward.png");
        registerResourceImage(AudioIcons.REWIND, "rewind.png");

        registerResourceImage(AudioIcons.NEXT, "next.png");
        registerResourceImage(AudioIcons.PREVIOUS, "previous.png");
        registerResourceImage(AudioIcons.FIRST, "first.png");
        registerResourceImage(AudioIcons.LAST, "last.png");

        registerResourceImage(AudioIcons.MUTED, "muted.png");
        registerResourceImage(AudioIcons.NOT_MUTED, "not_muted.png");

        registerResourceImage(AudioIcons.LOUDSPEAKER, "loudspeaker.png");
        registerResourceImage(AudioIcons.EDIT_AUDIO_SETTINGS, "edit_audio_settings.png");

        //do this, if your are assuming that this initializer should initialize all icons
        checkEnumAvailability(AudioIcons.class);
    }
```

```
34
35 }
```

Der Test in Zeile 32 ist optional und würde für den Fall, dass man z.B. vergessen hätte, ein Icon zu registrieren, eine `IllegalStateException` werfen. Wenn man zum Beispiel die Konstanten der Bibliothek erweitert, und die konkrete Registrierung vergisst, oder z.B. aus Versehen zwei mal den gleichen `key` für unterschiedliche Icons verwendet, tritt der Fehler bereits beim Starten der Applikation auf, was z.B. für einen *Smoketest* hilfreich ist, und nicht erst später, wenn das fehlende Icon verwendet wird.

Die folgende Abbildung zeigt die zugehörigen Icon Dateien innerhalb der Resourcen der Implementierung:

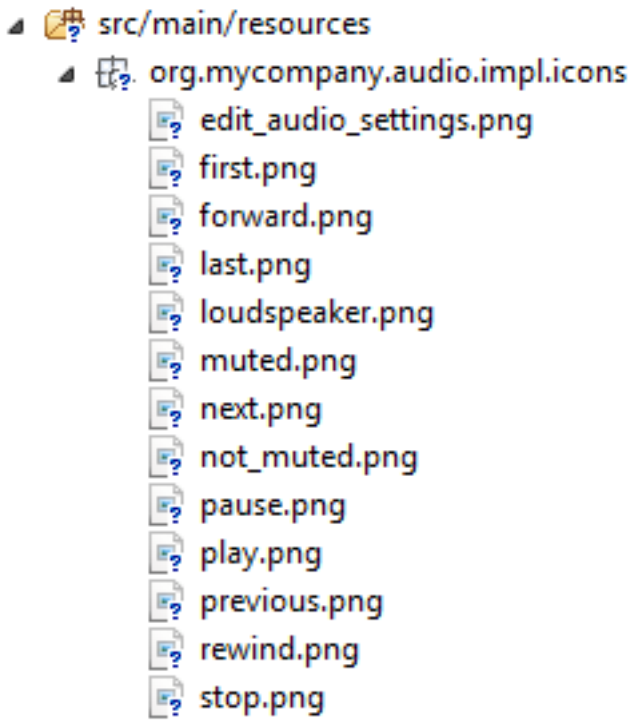


Abbildung 3.30: Audio Icon Resourcen

3.14.8.3 Registrierung mittels Toolkit Interceptor

Um sicher zu Stellen, dass die Icons vor der ersten Verwendung registriert sind, wird empfohlen, die Registrierung innerhalb eines Toolkit Interceptors durchzuführen. Das folgende Beispiel nutzt dazu die abstrakte Klasse `AbstractToolkitInterceptorHolder`:

```
1 package org.mycompany.audio.impl;
2
3 import org.jowidgets.api.toolkit.IToolkit;
4 import org.jowidgets.api.toolkit.IToolkitInterceptor;
5 import org.jowidgets.tools.toolkit.AbstractToolkitInterceptorHolder;
6
7 public final class AudioIconsInitializerToolkitInterceptor
```

```
8       extends AbstractToolkitInterceptorHolder {
9
10      @Override
11      protected IToolkitInterceptor createToolkitInterceptor() {
12          return new IToolkitInterceptor() {
13              @Override
14              public void onToolkitCreate(final IToolkit toolkit) {
15                  final AudioIconsInitializer initializer
16                      = new AudioIconsInitializer(toolkit.getImageRegistry());
17                  initializer.doRegistration();
18              }
19          };
20      }
21
22  }
```

Um diesen mit Hilfe des Java ServiceLoader Mechanismus zu registrieren, kann man unter `META-INF/services` eine Datei mit dem Namen `org.jowidgets.api.toolkit.IToolkitInterceptorHolder` und dem Inhalt:

```
org.jowidgets.helloworld.common.AudioIconsInitializerToolkitInterceptor
```

ablegen. Die folgende Abbildung soll das verdeutlichen:

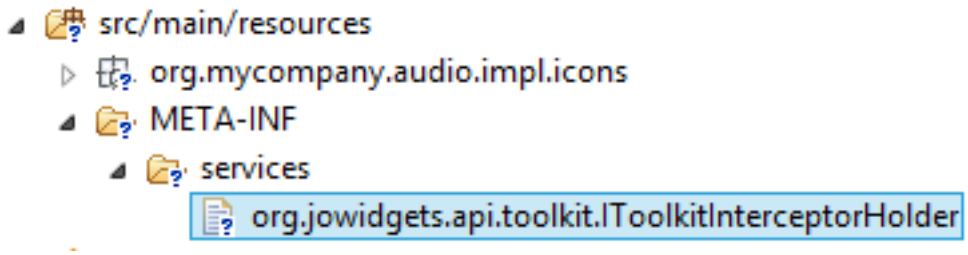


Abbildung 3.31: Icon Toolkit Interceptor

Dadurch werden die Icons automatisch registriert, wenn man das Modul `org.mycompany.audio.impl` im Classpath hat, und zwar bevor das jowidgets Toolkit verwendet wird, da der Interceptor bei der Erzeugung des Toolkit ausgeführt wird.

3.14.9 Überblick über vorhandene Icon Bibliotheken

Jowidgets liefert bereits vorgefertigte Icon Bibliotheken.

Die Enums `Icons` und `IconsSmall` liefern Konstanten, welche von jowidgets und der jo-client-platform verwendet werden. Das Modul `org.jowidget.impl` liefert konkrete Icons für diese, welcher unter der BSD Lizenz stehen, und somit frei verwendbar sind.

Das Addon Modul `org.jowidgets.addons.icons.silkicons` liefert eine IImageUrlProvider Enum für die Silk Icons von FamFamFam.

3.14.10 Betriebssystem Message Icons

Die Enum `org.jowidgets.api.image.Icons` enthält Image Konstanten für Betriebssystem Message Icons. Die folgende Tabelle zeigt die Icons mit dem zugehörigen Namen der Konstanten in der Enum.

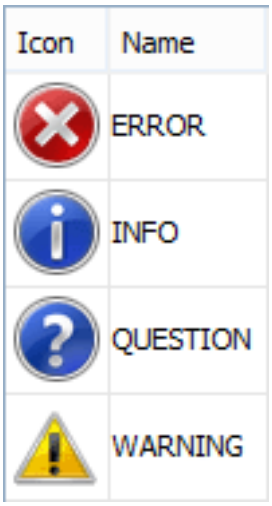


Abbildung 3.32: Icons

Das Aussehen dieser Icons hängt vom jeweiligen Betriebssystem ab. Die obere Tabelle wurde unter Windows 8 erstellt.

3.14.11 Icons Small

Die Enum `org.jowidgets.api.image.IconsSmall` enthält Icons, welche von jowidgets und der jo-client-platform verwendet werden. In der API finden sich nur die Konstanten ohne Default Icons, das Modul `org.jowidgets.impl` registriert für die Konstanten konkrete Icons in einer Auflösung von 16x16 Pixeln. Diese können wie in Austauschen von Icons beschrieben durch beliebige andere ersetzt (substituiert) werden. Die `IconsSmall` werden dort verwendet, wo in aktuellen (nicht Web) UI Frameworks eine 16x16 Auflösung üblich ist. Dazu zählen insbesondere Menüs, Labels, die Header von Fenstern, Tabellen, TabFoldern, etc..

Die folgende Tabelle zeigt die Default Icons der Enum `IconsSmall` mit dem zugehörigen Namen der Konstanten:

Diese obige Enum beinhalten auch die Betriebsystem Message Icons in einer kleiner Variante, welche in der folgenden Tabelle noch einmal gesondert dargestellt sind:

3.14.12 Silk Icons

Das Addon Modul `org.jowidgets.addons.icons.silkicons` liefert Icon Konstanten für die Silk Icons von FamFamFam. Diese stehen unter der Creative Commons Attribution 2.5 oder wahlweise Creative Commons Attribution 3.0 Lizenz. Weitere Informationen zur Lizenz finden sich auf der Seite von Silk Icons.

Um die SilkIcons zu verwenden, muss das folgende Modul hinzugefügt werden:

```
1     <dependency>
2         <groupId>org.jowidgets</groupId>
3         <artifactId>org.jowidgets.addons.icons.silkicons</artifactId>
4         <version>${jowidgets.version}</version>
5     </dependency>
```

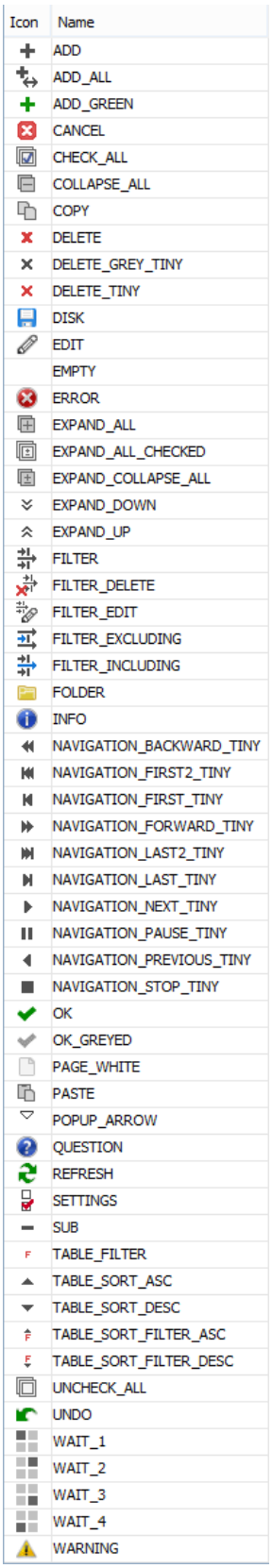


Abbildung 3.33: Icons Small

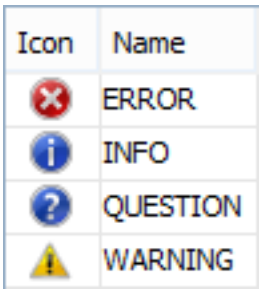


Abbildung 3.34: System Icons Small

Mit Hilfe des IconTableSnipped werden alle SilkIcons in einer Tabelle angezeigt:

Eine Auflistung aller Icons findet sich auch hier.

Hinweis: Jowidgets verwendet bewusst keine Silk Icon Konstanten für die eigenen Widgets, unter anderem weil unklar ist, inwieweit die Lizenzbedingungen von Silk Icons mit der BSD Lizenz kompatibel sind.[19] Jowidgets verwendet allerdings in einigen Beispielapplikationen die Silk Icons. Diese Beispiele gehören jedoch nicht zum Kern und sind somit auch nicht Teil einer mit jowidgets erstellten Applikation.

Bei der Nutzung von jowidgets als UI Framework kann man somit frei entscheiden, ob man SilkIcons verwenden möchte oder nicht. Im ersten Fall fügt man das Silk Icons Addon zu seinem Projekt hinzu, im zweiten Fall muss man nichts zusätzliches unternehmen.

3.14.13 Die Image Factory

Die Image Factory ermöglicht die Erzeugung von Images oder Buffered Images.

Die Schnittstelle 'IImageFactory' hat die folgenden Methoden:

```
IImage createImage(File file);

IImage createImage(URL url);

IImage createImage(IFactory<InputStream> inputStream);

IBufferedImage createBufferedImage(int width, int height);
```

3.14.13.1 Image Factory Instanz

Eine Instanz erhält man vom Toolkit mit Hilfe der folgenden Methode:

```
IImageFactory getImageFactory();
```

3.14.13.2 Die Schnittstelle IImage

Die Schnittstelle `IImage` hat die folgenden Methoden:

[19] Das bedeutet jedoch nicht, dass sie nicht kompatibel sind, sondern nur, dass dies bisher nicht geprüft wurde. Es soll insbesondere vermieden werden, dass man, wenn man jowidgets verwendet, automatisch an die Lizenzbedingungen von Silk Icons gebunden ist, sogar dann, wenn man diese Icon Konstanten durch eigene Images substiuiert.

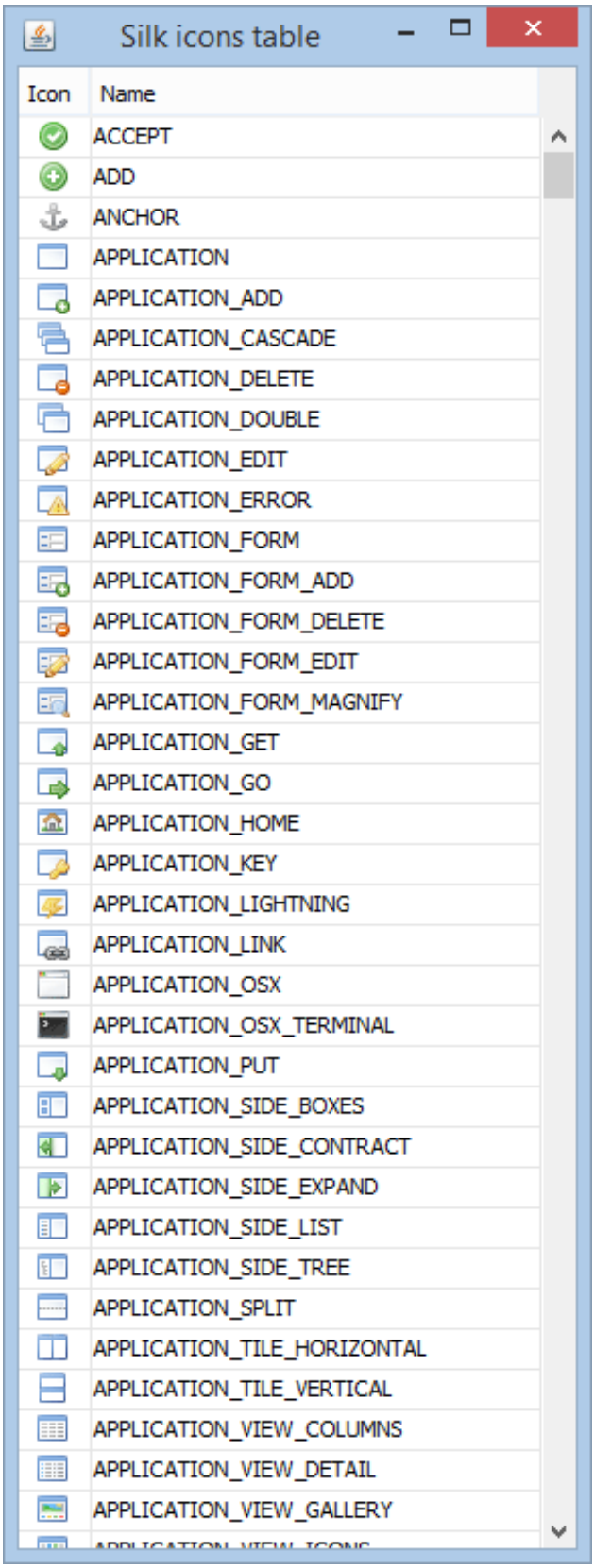


Abbildung 3.35: IconTableSnipped mit SilkIcons

```
    Dimension getSize();

    void initialize();

    void dispose();

    boolean isDisposed();

    void addDisposeListener(IDisposeListener listener);

    void removeDisposeListener(IDisposeListener listener);
```

Ein `IImage` ist von `IImageConstant` abgeleitet, und kann somit überall verwendet werden, wo Image Konstanten verwendet werden können.[20]

Wird ein Image (oder Buffered Image) von der Image Factory erzeugt, wird es automatisch in der Image Registry registriert, und ist somit für die Verwendung in Widgets unmittelbar verfügbar.

Das native Image (z.B. `org.eclipse.swt.graphics.Image`) wird erst erzeugt, wenn es einem Widget zugewiesen, oder die Methode `getSize()` oder `initialize()` aufgerufen wurde. Wird das Image mehrfach (für unterschiedliche Widgets) verwendet, wird trotzdem nur eine native Image Instanz erzeugt.

Wird die `dispose()` Methode auf dem Image aufgerufen, wird das Image automatisch aus der Image Registry entfernt und das zugehörige native Image disposed, falls es bereits initialisiert wurde. Das gleiche passiert, wenn auf der Image Registry die Methode `unRegisterImage(...)` für das Image aufgerufen wird. In beiden Fällen führen anschließend alle Operation auf dem Image (außer `isDisposed()`) zu einem Fehler.

Images können in jedem beliebigen Thread erzeugt werden. Für Images, welche explizit außerhalb des Ui Thread geladen werden sollen, sollte man darauf achten, dass *ohne* expliziten Aufruf der Methode `initialize()` nur ein Image Handle erzeugt wird, und das eigentliche Laden erst bei der ersten Verwendung im UI Thread stattfindet.

3.14.13.3 Die Schnittstelle IBufferedImage

Die Schnittstelle `IBufferedImage` ist von `IImage` abgeleitet. Zusätzlich zu den geerbten Methoden hat ein Bufferd Image die folgende weitere Methode:

```
    IGraphicContext getGraphicContext();
```

Ein Buffered Image liefert einen Graphic Context zum Zeichnen des Bildes.

3.14.13.4 Das ImageIconSnipped

Das `ImageIconSnipped` zeigt die Verwendung eines Images in Verbindung mit einem Icon Widget:

```
1 public final class ImageIconSnipped implements IApplication {
2
3     @Override
4     public void start(final IApplicationLifecycle lifecycle) {
5
```

[20] Hier macht es sich besonders unangenehm bemerkbar, dass der Name `IImageConstant` und nicht `IImageKey` gewählt wurde, da ein `IImage` nicht zwingend eine Konstante sein muss und das `dispose()` explizit zum Vertrag gehört.

```
6           //create the root frame
7           final IFrame frame = Toolkit.createRootFrame(
8               BPF.frame("Image Icon Snipped"),
9               lifecycle);
10          frame.setLayout(FillLayout.get());
11
12          //create a scroll composite
13          final IScrollComposite container = frame.add(BPF.scrollComposite());
14          container.setLayout(FillLayout.get());
15
16          //create a image from url
17          String url = "http://www.jowidgets.org/docu/images/widgets_hierarchy_1.gif";
18          final IImage image = ImageFactory.createImage(UrlFactory.create(url));
19
20          //use the icon widget to display the image
21          final IIcon imageIcon = container.add(BPF.icon(image));
22
23          //remove the icon on double click from its container to test dispose
24          imageIcon.addMouseListener(new MouseAdapter() {
25              @Override
26              public void mouseDoubleClicked(final IMouseButtonEvent mouseEvent) {
27                  imageIcon.dispose();
28                  container.layoutLater();
29              }
30          });
31
32          //dispose the image if it was removed from its container
33          imageIcon.addDisposeListener(new IDisposeListener() {
34              @Override
35              public void onDispose() {
36                  image.dispose();
37                  //CHECKSTYLE:OFF
38                  System.out.println("DISPOSED IMAGE");
39                  //CHECKSTYLE:ON
40              }
41          });
42
43          //set the root frame visible
44          frame.setVisible(true);
45      }
46  }
```

Das `image` wird auf einem Icon Widget gesetzt (Zeile 21). Auf dem Icon Widget wird ein Listener registriert, welcher dieses bei einem Doppeklick aus seinem Container entfernt, wodurch es disposed wird.

In Zeile 16 bis 18 könnte man zum Beispiel auch das Folgende schreiben, um das Image aus einem File einzulesen:

```
1       //create a image from file
2       final String path = "C:/projects/jo-widgets/trunk/docu/images/widgets_hierarchy_1.gif";
3       final IImage image = ImageFactory.createImage(new File(path));
```

3.14.13.5 Das ImageCanvasSnipped

Das ImageCanvasSnipped zeigt die Verwendung eines Images in Verbindung mit einem Canvas:

```
1   public final class ImageCanvasSnipped implements IApplication {
2
3       @Override
```

```
4       public void start(final IApplicationLifecycle lifecycle) {
5
6           //create the root frame
7           final IFrame frame = Toolkit.createRootFrame(BPF.frame("Image Canvas Snipped"), lifecycle);
8           frame.setLayout(FillLayout.get());
9
10          //create a scroll composite
11          final IScrollComposite container = frame.add(BPF.scrollComposite());
12          container.setLayout(FillLayout.get());
13
14          //create a image from url
15          final String url = "http://www.jowidgets.org/docu/images/widgets_hierarchy_1.gif";
16          final IImage image = ImageFactory.createImage(UrlFactory.create(url));
17
18          //use a canvas to display the image
19          final ICanvas canvas = container.add(BPF.canvas());
20
21          //set the preferred size of the canvas to the image size
22          canvas.setPreferredSize(image.getSize());
23
24          //add a paint listener to draw the image and an arrow
25          canvas.addPaintListener(new IPaintListener() {
26              @Override
27              public void paint(final IPaintEvent event) {
28                  final IGraphicContext gc = event.getGraphicContext();
29
30                  //draw the image
31                  gc.drawImage(image);
32
33                  //draw with green color
34                  gc.setForegroundColor(Colors.GREEN);
35
36                  //define a polygon that shapes an arrow
37                  final Point p1 = new Point(438, 205);
38                  final Point p2 = new Point(464, 205);
39                  final Point p3 = new Point(464, 199);
40                  final Point p4 = new Point(486, 211);
41                  final Point p5 = new Point(464, 223);
42                  final Point p6 = new Point(464, 217);
43                  final Point p7 = new Point(438, 217);
44                  final Point[] polygon = new Point[] {p1, p2, p3, p4, p5, p6, p7, p1};
45
46                  //fill the polygon
47                  gc.fillPolygon(polygon);
48              }
49          });
50
51          //set the root frame visible
52          frame.setVisible(true);
53      }
54  }
```

Das Image wird mit Hilfe eine Canvas gezeichnet (Zeile 31). Anschließend wird auf der Grafik ein grüner Pfeil, welcher durch ein Polygon definiert wird, gezeichnet (Zeile 47). Die folgende Abbildung zeigt das Ergebnis:

3.14.13.6 Das BufferedImageSnipped

Das `BufferedImageSnipped` zeigt die Verwendung eines Buffered Image:

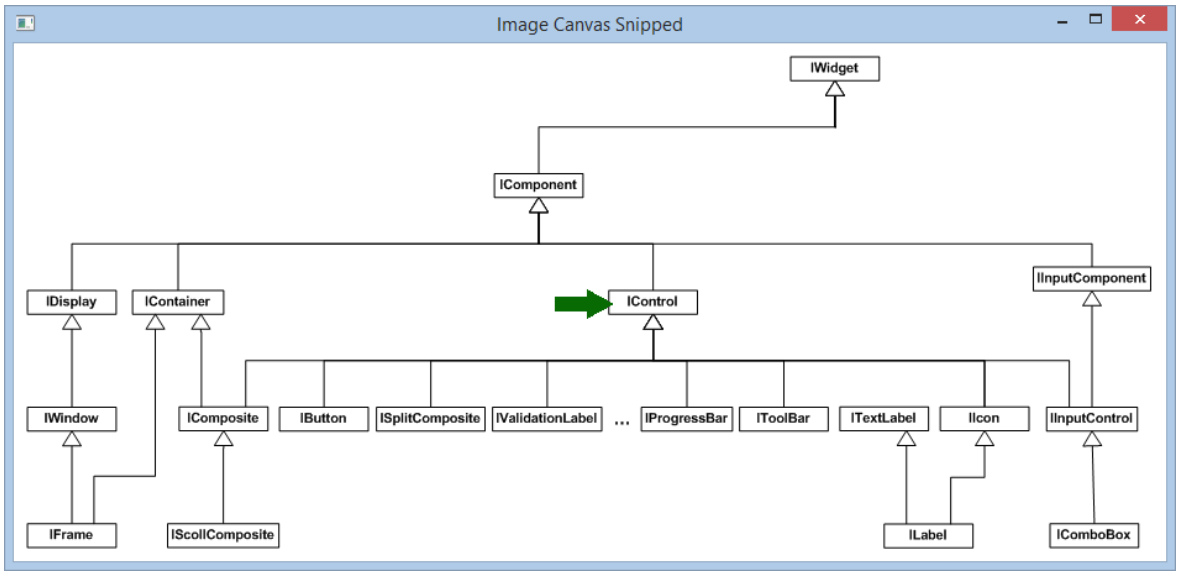


Abbildung 3.36: ImageCanvasSnipped

```
1  public final class BufferedImageSnipped implements IApplication {
2
3      @Override
4      public void start(final IApplicationLifecycle lifecycle) {
5
6          //create the root frame
7          final IFrameBluePrint frameBp = BPF.frame("Buffered Image Snipped");
8          final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);
9          frame.setSize(300, 200);
10         frame.setBackgroundColor(Colors.WHITE);
11         frame.setLayout(FillLayout.builder().margin(10).build());
12
13         //create a arrow buffered image
14         final IBufferedImage image = createArrowImage();
15
16         //create a label using the buffered image as icon
17         frame.add(BPF.label().setIcon(image).setText("Hello world"));
18
19         //set the root frame visible
20         frame.setVisible(true);
21     }
22
23     private static IBufferedImage createArrowImage() {
24         //create a buffered image
25         final IBufferedImage image = ImageFactory.createBufferedImage(52, 26);
26         final IGraphicContext gc = image.getGraphicContext();
27
28         //use anti aliasing
29         gc.setAntiAliasing(AntiAliasing.ON);
30
31         //define a polygon that shapes an arrow
32         final Point p1 = new Point(0, 6);
33         final Point p2 = new Point(26, 6);
34         final Point p3 = new Point(26, 0);
35         final Point p4 = new Point(48, 12);
36         final Point p5 = new Point(26, 24);
37         final Point p6 = new Point(26, 18);
```

```
38          final Point p7 = new Point(0, 18);
39          final Point[] polygon = new Point[] {p1, p2, p3, p4, p5, p6, p7, p1};
40
41          //use white background for the image
42          gc.setBackgroundColor(Colors.WHITE);
43          gc.clear();
44
45          //draw with green color
46          gc.setForegroundColor(Colors.GREEN);
47
48          //fill the polygon
49          gc.fillPolygon(polygon);
50
51          return image;
52      }
53  }
```

Die folgende Abbildung zeigt das Ergebnis:

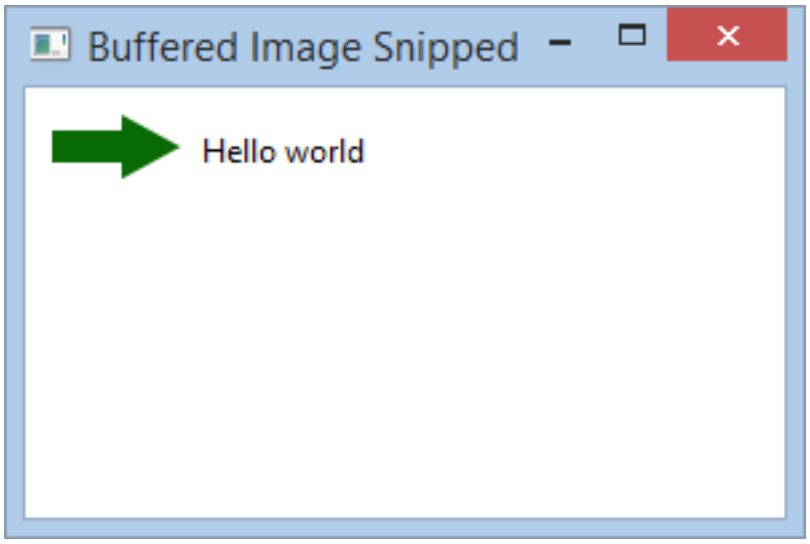


Abbildung 3.37: BufferedImageSnipped

3.15 Jowidgets Converter

3.15.1 Maskierte Texteingaben

3.16 Observable Values - Übersicht

Ein Observable Value ist ein Container für einen Wert, der sich zur Laufzeit ändern kann. Mit Hilfe eines `IObservableValueListener` kann man sich über Änderungen des Wertes informieren lassen. Observable Values können an andere Observable Values gebunden werden. Ein Observable Value Viewer ist ein Widget, welches einen Observable Value visualisiert und (optional) eine Modifikation des Wertes ermöglicht.

3.16.1 Observable Value Schnittstelle und Implementierungen

Im Folgenden wird die Schnittstelle `IObservableValue` vorgestellt und es werden existierende Default Implementierungen gezeigt.

3.16.1.1 Die Schnittstelle IObservableValue

Die Schnittstelle `IObservableValue` sieht wie folgt aus:

```
1  public interface IObservableValue<VALUE_TYPE> {
2
3      void setValue(VALUE_TYPE value);
4
5      VALUE_TYPE getValue();
6
7      void addValueListener(IObservableValueListener<?> listener);
8
9      void removeValueListener(IObservableValueListener<?> listener);
10 }
```

Ein `IObservableValueListener` hat die folgende Methode:

```
    void changed(IObservableValue<VALUE_TYPE> observableValue, VALUE_TYPE value);
```

Man bekommt sowohl den Observable Value der sich geändert hat, als auch den neuen Wert (`value`) übergeben. Der Listener feuert nur dann, wenn sich der Wert tatsächlich geändert hat. Wird zum Beispiel ein zweites mal der gleiche Wert gesetzt, wird kein ChangedEvent geworfen.

3.16.1.2 ObservableValue Default Implementierung

Die Klasse `ObservableValue` bietet eine Default Implementierung der Schnittstelle `IObservableValue`.

Das folgende Beispiel demonstriert die Verwendung der Klasse `ObservableValue`:

```
1      final IObservableValue<IPerson> forkliftDriver = new ObservableValue<IPerson>(dieter);
2
3      forkliftDriver.addValueListener(new IObservableValueListener<IPerson>() {
4          @Override
5          public void changed(final IObservableValue<IPerson> observableValue, final IPerson value) {
6              diaphone.setActive(value == klaus);
7          }
8      });
9
10     forkliftDriver.setValue(klaus);
```

Die Default Implementierung implementiert **nicht** equals() und hashCode(). Zwei `ObservableValue` Objekte sind insbesondere **nicht** `equal`, wenn ihre values gleich sind. Der folgende UnitTest soll verdeutlichen, warum:

```
1      final ObservableValue<String> value = new ObservableValue<String>();
2      value.setValue(STRING_1);
3
4      final Set<ObservableValue<String>> set = new HashSet<ObservableValue<String>>();
5      set.add(value);
```

```
6
7       value.setValue(STRING_2);
8
9       Assert.assertTrue(set.remove(value));
```

Durch das Ändern des Wertes in Zeile 7 würde sich der hasCode() ändern, wodurch der `value` nicht mehr aus der Liste entfernt werden könnte.

Die Klasse `ObservableValue` wurde so entworfen, dass davon abgeleitet werden kann. Das folgenden Beispiel zeigt eine `ObservablePerson`, welche `equals()` und `hashCode()` mit Hilfe einer id implementiert:

```
1  import org.jowidgets.util.Assert;
2  import org.jowidgets.util.ObservableValue;
3
4  public final class ObservablePerson extends ObservableValue<IPerson> {
5
6      private final Object id;
7
8      public ObservablePerson(final Object id) {
9          Assert.paramNotNull(id, "id");
10         this.id = id;
11     }
12
13     @Override
14     public int hashCode() {
15         final int prime = 31;
16         int result = 1;
17         result = prime * result + ((id == null) ? 0 : id.hashCode());
18         return result;
19     }
20
21     @Override
22     public boolean equals(final Object obj) {
23         if (this == obj) {
24             return true;
25         }
26         if (obj == null) {
27             return false;
28         }
29         if (!(obj instanceof ObservablePerson)) {
30             return false;
31         }
32         final ObservablePerson other = (ObservablePerson) obj;
33         if (id == null) {
34             if (other.id != null) {
35                 return false;
36             }
37         }
38         else if (!id.equals(other.id)) {
39             return false;
40         }
41         return true;
42     }
43
44 }
```

3.16.1.3 ObservableValueWrapper

Um ein `IObservableValue` mit Hilfe des Wrapper Patters zu *wrappen*, zum Beispiel um den Wert zu dekorieren, kann die Klasse `ObservableValueWrapper` verwendet werden. Das folgende Beispiel soll das verdeutlichen:

```
import org.jowidgets.util.IDecorator;
import org.jowidgets.util.IObservableValue;
import org.jowidgets.util.ObservableValueWrapper;

public final class ObservableValueDecorator<VALUE_TYPE>
    implements IDecorator<IObservableValue<VALUE_TYPE>> {

    //injected
    private IAuthorizationService authorizationService;

    @Override
    public IObservableValue<VALUE_TYPE> decorate(final IObservableValue<VALUE_TYPE> original) {
        if (authorizationService == null) {
            return original;
        }
        else {
            return new ObservableValueWrapper<VALUE_TYPE>(original) {
                @Override
                public void setValue(final VALUE_TYPE value) {
                    if (!authorizationService.hasAuthorization(Authorizations.UPDATE)) {
                        throw new SecurityException("No authorization for update");
                    }
                    else {
                        super.setValue(value);
                    }
                }
            };
        }
    }
}
```

3.16.1.4 MandatoryObservableValue

Die Klasse `MandatoryObservableValue` liefert eine Implementierung von `IObservableValue` welche nicht den Wert `null` annehmen kann. Die Implementierung sieht wie folgt aus:

```
public class MandatoryObservableValue<VALUE_TYPE> extends ObservableValue<VALUE_TYPE> {

    private final VALUE_TYPE defaultValue;

    public MandatoryObservableValue(final VALUE_TYPE defaultValue) {
        Assert.paramNotNull(defaultValue, "defaultValue");
        this.defaultValue = defaultValue;
    }

    @Override
    public void setValue(final VALUE_TYPE value) {
        if (value != null) {
            super.setValue(value);
        }
        else {
            super.setValue(defaultValue);
        }
    }

    @Override
    public VALUE_TYPE getValue() {
        final VALUE_TYPE superResult = super.getValue();
        if (superResult != null) {
            return superResult;
        }
        else {
```

```
27              return defaultValue;
28          }
29      }
30  }
```

Ein `MandatoryObservableValue` hat einen Default Value (Zeile 3), welcher verwendet wird, sobald `null` gesetzt wird.

3.16.1.5 ObservableBoolean

Ein `ObservableBoolean` liefert eine Implementierung von `IObservableValue` für ein `Boolean` bei dem nicht gewünscht ist, dass der Wert `null` angenommen werden kann, sondern nur `true` und `false`. Die Implementierung sieht wie folgt aus:

```
1   public final class ObservableBoolean extends ObservableValue<Boolean> {
2
3       public ObservableBoolean() {
4           setValue(false);
5       }
6
7       public ObservableBoolean(final boolean value) {
8           set(value);
9       }
10
11      public boolean get() {
12          return getValue().booleanValue();
13      }
14
15      public void set(final boolean value) {
16          super.setValue(Boolean.valueOf(value));
17      }
18
19      @Override
20      public void setValue(final Boolean value) {
21          Assert.paramNotNull(value, "value");
22          super.setValue(value);
23      }
24  }
```

Die Methoden `get()` und `set()` bieten einen komfortablen Zugriff auf den *kleinen* boolean ohne *Autoboxing*.[21] Der Ausdruck `boolean b = get()` kann (im Vergleich zu `boolean b = getValue()` auf einen herkömmlichen Observable Value) nie eine `NullPointerException` werfen, da man das Setzen von `null` explizit verhindert (Zeile 21). Man sollte einen solchen Wert nur an Observable Values binden, die ebenfalls Mandatory sind. Man könnte den Code ab Zeile 21 auch wie folgt ändern:

```
1       @Override
2       public void setValue(final Boolean value) {
3           if (value != null){
4               super.setValue(value);
5           }
6           else{
7               super.setValue(Boolean.FALSE);
8           }
9       }
```

[21] Warum Atoboxing **evil** ist, wird unter anderem auch hier diskutiert: [https://pboop.wordpress.com/2010/09/22/autoboxing-is-evil/]

Dies würde dem Verhalten des MandatoryObservableValue entsprechen, wobei man zusätzlich noch die sichere get() Methode hat.

3.16.2 Observable Value Binding

Die Klasse `org.jowidgets.util.binding.Bind` kann verwendet werden, um zwei Observable Values aneinander zu binden binden. Sie hat die folgenden statischen Methoden:

```
    public static <VALUE_TYPE> IBinding bind(
        final IObservableValue<VALUE_TYPE> source,
        final IObservableValue<VALUE_TYPE> destination) {...}

    public static <SOURCE_TYPE, DESTINATION_TYPE> IBinding bind(
        final IObservableValue<SOURCE_TYPE> source,
        final IObservableValue<DESTINATION_TYPE> destination,
        final IBindingConverter<SOURCE_TYPE, DESTINATION_TYPE> converter) {...}
```

Die erste Methode kann verwendet werden, um typgleiche Values aneinander zu binden, die zweite Methode erlaubt das Binden von unterschiedlichen Typen. Die Bindung ist **immer bidirektional**, das bedeutet, Änderungen auf `source` ändern die `destination` und Änderungen auf `destination` ändern die `source`. Haben `source` und `destination` initial einen unterschiedlichen Wert, nimmt durch das Binden `destination` den Wert von `source` an. Beide Methoden liefern eine `IBinding` Referenz zurück. Diese hat die folgenden Methoden:

```
    void setBindingState(boolean bind);

    void unbind();

    void bind();

    boolean isBound();

    void dispose();

    boolean isDisposed();
```

Die ersten drei Methoden dienen zum Setzen, die vierte zum Auslesen des `binding` State. Dadurch kann das Binding temporär gelöst und später wieder aktiviert werden. Initial ist der `binding` State `true`. Durch den Aufruf von `dispose()` wird das Binding dauerhaft gelöst, und die internen Referenzen auf `source` und `destination` verworfen. Nach einem Aufruf von `dispose()` kann nur noch die Methode `isDisposed()` aufgerufen werden. Alle anderen Methodenaufrufe führen dann zu einer `IllegalStateException`.

3.16.2.1 Binding Converter

Um Obervable Values unterschiedlichen Typs zu binden, kann ein `IBindingConverter` verwendet werden. Dieser ist wie folgt definiert:

```
1  public interface IBindingConverter<SOURCE_TYPE, DESTINATION_TYPE> {
2
3      DESTINATION_TYPE convertSource(SOURCE_TYPE sourceValue);
4
5      SOURCE_TYPE convertDestination(DESTINATION_TYPE destinationValue);
6  }
```

Das folgende Beispiel implementiert einen Binding Converter, welcher eine Liste in ein Array und zurück konvertiert:

```
public final class ListArrayBindingConverter implements IBindingConverter<List<String>, String[]> {

    @Override
    public String[] convertSource(final List<String> list) {
        if (list != null) {
            return list.toArray(new String[list.size()]);
        }
        else {
            return null;
        }
    }

    @Override
    public List<String> convertDestination(final String[] destinationValue) {
        if (destinationValue != null) {
            return new ArrayList<String>(Arrays.asList(destinationValue));
        }
        else {
            return null;
        }
    }
}
```

Dieser könnte wie folgt verwendet werden können:

```
    final ObservableValue<List<String>> source = new ObservableValue<List<String>>();
    final ObservableValue<String[]> destination = new ObservableValue<String[]>();

    Bind.bind(source, destination, new ListArrayBindingConverter());
```

3.16.2.2 Binding Test Beispiel

Der folgende JUnitTest demonstriert die Verwendung der Klasse `Bind`. Der Test wurde etwas verkürzt, der vollständige Test findet sich hier:

```
public class BindingTest {

    private static String STRING_1 = "STRING_1";
    private static String STRING_2 = "STRING_2";
    private static String STRING_3 = "STRING_3";
    private static String STRING_4 = "STRING_4";
    private static String STRING_5 = "STRING_5";
    private static String STRING_6 = "STRING_6";
    private static String STRING_7 = "STRING_7";
    private static String STRING_8 = "STRING_8";
    private static String STRING_9 = "STRING_9";
    private static String STRING_10 = "STRING_10";

    @Test
    public void testBinding() {
        //Create two observable values
        final ObservableValue<String> source = new ObservableValue<String>(STRING_1);
        final ObservableValue<String> destination = new ObservableValue<String>(STRING_2);

        //create a new binding
        final IBinding binding = Bind.bind(source, destination);

```

```
        //must be equal and must be STRING_1 (source before binding)
        testEquality(source, destination, STRING_1);

        //change source must change destination
        source.setValue(STRING_3);

        //must be equal and STRING_3
        testEquality(source, destination, STRING_3);

        //change destination must change source
        destination.setValue(STRING_4);

        //must be equal and STRING_4
        testEquality(source, destination, STRING_4);

        //unbind the values
        binding.unbind();

        //after unbind, change source, destination changes not
        source.setValue(STRING_5);
        Assert.assertEquals(STRING_5, source.getValue());
        Assert.assertEquals(STRING_4, destination.getValue());

        //after unbind, change destination, source changes not
        destination.setValue(STRING_6);
        Assert.assertEquals(STRING_6, destination.getValue());
        Assert.assertEquals(STRING_5, source.getValue());

        //bind the values again
        binding.bind();

        //must be equal and STRING_5 (last source value)
        testEquality(source, destination, STRING_5);
    }

    private void testEquality(
        final IObservableValue<String> source,
        final IObservableValue<String> destination,
        final String expectedValue) {

        //the values of the observable value must be equal
        Assert.assertEquals(source.getValue(), destination.getValue());

        //the source must be the expected value
        Assert.assertEquals(expectedValue, source.getValue());

        //the destination must be the expected value
        Assert.assertEquals(expectedValue, destination.getValue());
    }

}
```

3.16.3 Observable Value Viewer

Ein Observable Value Viewer ist ein Widget, welches einen Observable Value visualisiert und (optional) eine Modifikation des Wertes ermöglicht.

3.16.3.1 Die Schnittstelle IObservableValueViewer

Die Schnittstelle `IObservableValueViewer` ist wie folgt definiert:

```
public interface IObservableValueViewer<VALUE_TYPE> {

    IObservableValue<VALUE_TYPE> getObservableValue();

}
```

3.16.3.2 Abgeleitete Widgets

Die Schnittstelle wird derzeit von den folgenden Widgets implementiert:

- CheckBox
- ToggleButton
- ComboBox
- Combobox Selection
- InputField
- Slider
- Slider Viewer

Die BluePrints dieser Widgets haben jeweils die Methode:

```
    BLUE_PRINT_TYPE setObservableValue(IObservableValue<VALUE_TYPE> value);
```

Dadurch wird der übergebene Obserservable Value an den neu erzeugten Viewer gebunden. Das Binding wird wieder gelöst, wenn das Widget disposed wird. Man kann dadurch anstatt:

```
1   final IObservableValue<Double> value = new ObservableValue<Double>(0.0d);
2
3   //add slider
4   final ISliderViewerBluePrint<Double> sliderBp = BPF.sliderViewerDouble(-1.0d, 1.0d);
5   final ISliderViewer<Double> sliderViewer = frame.add(sliderBp, "w 50::");
6
7   final IBinding binding = Bind.bind(value, sliderViewer.getObservableValue());
8   sliderViewer.addDisposeListener(new IDisposeListener() {
9       @Override
10      public void onDispose() {
11          binding.dispose();
12      }
13  });
```

einfach das folgende Schreiben:

```
1   final IObservableValue<Double> value = new ObservableValue<Double>(0.0d);
2
3   //add slider
4   final ISliderViewerBluePrint<Double> sliderBp = BPF.sliderViewerDouble(-1.0d, 1.0d);
5   sliderBp.setObservableValue(value);
6   frame.add(sliderBp, "w 50::");
```

3.16.3.3 Observable Value Viewer Snipped

Das ObservableValueSnipped verwendet einen Observable Value, um ein SliderViewer und ein InputField aneinander zu binden:

```java
public final class ObservableValueViewerSnipped implements IApplication {

    @Override
    public void start(final IApplicationLifecycle lifecycle) {

        //create the root frame
        final IFrameBluePrint frameBp = BPF.frame("Observable value viewer snipped");
        final IFrame frame = Toolkit.createRootFrame(frameBp, lifecycle);
        frame.setLayout(new MigLayoutDescriptor("wrap", "[][][]", "[]"));

        //create panorama observable value
        final IObservableValue<Double> panorama = new ObservableValue<Double>(0.0d);

        //add panorama label
        final ITextLabelBluePrint labelBp = BPF.textLabel("Panorama");
        labelBp.setForegroundColor(Colors.STRONG).setMarkup(Markup.STRONG);
        frame.add(labelBp);

        //add panorama slider
        final ISliderViewerBluePrint<Double> sliderBp = BPF.sliderViewerDouble(-1.0d, 1.0d);
        sliderBp.setObservableValue(panorama);
        frame.add(sliderBp, "growx, w 150!");

        //add panorama input field
        final IInputFieldBluePrint<Double> inputFieldBp = BPF.inputFieldDoubleNumber();
        inputFieldBp.setObservableValue(panorama);
        frame.add(inputFieldBp, "w 50!");

        //set the root frame visible
        frame.setVisible(true);
    }
}
```

Änderungen am Slider modifizieren das Eingabefeld und umgekehrt. Die folgende Abbildung zeigt das Ergebnis:

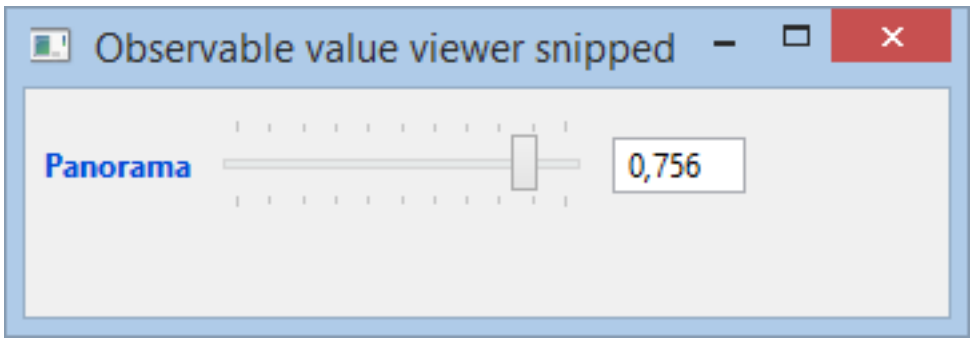


Abbildung 3.38: Observable Value Viewer Snipped

3.16.3.4 Binding Snipped

Das BindingSnipped verwendet mehrere Observable Values, um jeweils ein SliderViewer und ein InputField aneinander zu binden. Zudem kann man über eine Checkbox das Binding aller Observable Values untereinander aktivieren oder deaktivieren, wodurch sich die Slider und Eingabefelder synchron oder unabhängig voneinander sind:

```java
public final class BindingSnipped implements IApplication {

    private static final int COLUMNS = 10;
    private static final double MIN_VALUE = -1.0d;
    private static final double MAX_VALUE = 1.0d;
    private static final double DEFAULT_VALUE = 0.0d;

    @Override
    public void start(final IApplicationLifecycle lifecycle) {

        //create the root frame
        final IFrame frame = Toolkit.createRootFrame(BPF.frame("Binding snipped"), lifecycle);
        frame.setLayout(new MigLayoutDescriptor(
            "wrap",
            StringUtils.loop("[]10", COLUMNS - 1) + "[]",
            "[]0[]20[]"));

        //create observable values and bindings
        final ArrayList<IObservableValue<Double>> observableValues
            = new ArrayList<IObservableValue<Double>>(COLUMNS);
        final ArrayList<IBinding> bindings = new ArrayList<IBinding>(COLUMNS - 1);
        for (int i = 0; i < COLUMNS; i++) {
            final IObservableValue<Double> observableValue = new ObservableValue<Double>();
            observableValues.add(observableValue);
            if (i > 0) {
                //bind next value to the previous
                final IBinding binding = Bind.bind(
                    observableValues.get(i - 1),
                    observableValue);
                bindings.add(binding);
            }
        }

        //add sliders
        for (int i = 0; i < COLUMNS; i++) {
            final ISliderViewerBluePrint<Double> sliderBp
                = BPF.sliderViewerDouble(MIN_VALUE, MAX_VALUE);
            sliderBp.setVertical();
            sliderBp.setDefaultValue(DEFAULT_VALUE);
            sliderBp.setObservableValue(observableValues.get(i));
            frame.add(sliderBp, "w 50::");
        }

        //add input fields
        for (int i = 0; i < COLUMNS; i++) {
            final IInputFieldBluePrint<Double> inputFieldBp
                = BPF.inputFieldDoubleNumber();
            inputFieldBp.setObservableValue(observableValues.get(i));
            frame.add(inputFieldBp, "w 50::");
        }

        //add binding checkbox
        final ICheckBox bindingCb = frame.add(BPF.checkBox().setText("Bind"));
        bindingCb.setSelected(true);
        bindingCb.addInputListener(new IInputListener() {
            @Override
            public void inputChanged() {
                for (final IBinding binding : bindings) {
                    binding.setBindingState(bindingCb.isSelected());
                }
            }
        });

        //set the root frame visible
```

```
65          frame.setVisible(true);
66      }
67  }
```

Die folgende Abbildung zeigt das Ergebnis:

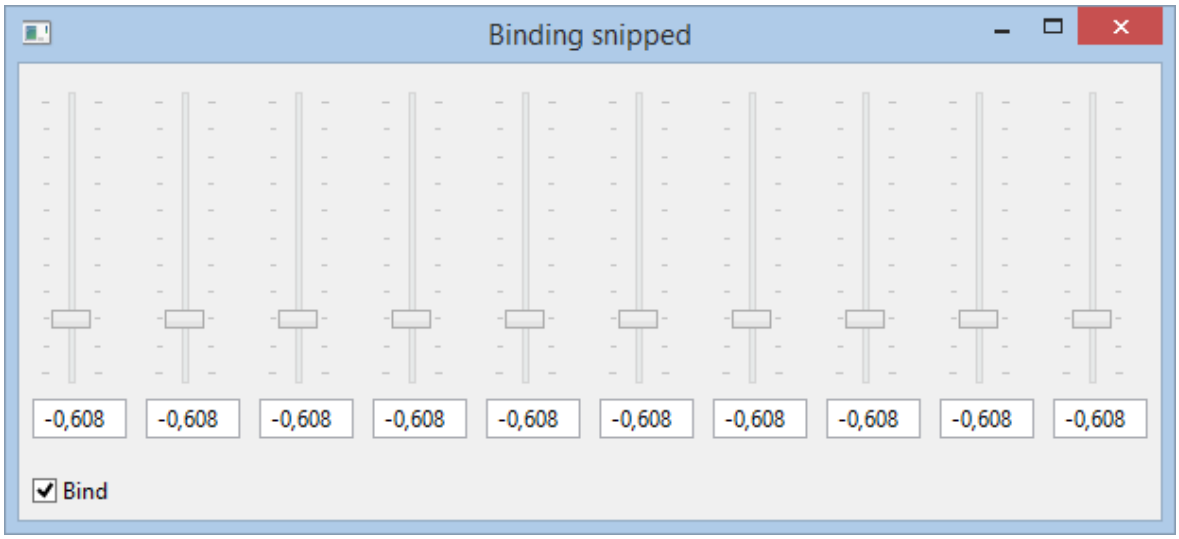


Abbildung 3.39: Binding Snipped

Kapitel 4

Core Widgets - Übersicht

Bei den Core Widgets handelt es sich (im Abgrenzung zu den Addon Widgets) um die Widgets, welche für alle SPI Implementierungen und Betriebssysteme verfügbar sind.[1]

Bei den Core Widgets kann man zwischen Basis Widgets und Composite Widgets unterscheiden. Basis Widgets erweitern dabei direkt ein SPI Widget und fügen zum Beispiel Convenience Methoden hinzu, während Composite Widgets aus anderen (u.U. nur einem) Basis Widgets und / oder Composite Widgets zusammengesetzt sind, und ganz neue Funktionen anbieten.

Zu den Basis Widgets zählen zum Beispiel das TextField, die Combobox, der Slider, der Tree, die Tabelle, usw., zu den Composite Widgets zählen zum Beispiel das InputField, das UnitValueField, das CollectionInputField, das ExpandComposite, der PasswordChangeDialog, der SliderViewer, und weitere.

Für die Verwendung der Widgets ist diese Unterscheidung allerdings nicht relevant, weshalb die Widgets in diesem Kapitel auch nicht nach diesem Kriterium gruppiert sind.

Hinweis:

Zum aktuellen Zeitpunkt existiert **noch nicht** für jedes Core Widget eine ausführliche Beschreibung in dieser Dokumentation. Allerdings gibt es für **alle Widgets** Beispielapplikation, welche die Verwendung demonstrieren. Möchte man ein Widget nutzen, welches nicht in diesem Dokument detailliert beschrieben ist, wird empfohlen per `References Workspace` in Eclipse für das zugehörige **BluePrint**[2] nach den Beispielen zu suchen. Dazu wird empfohlen, jowidgets komplett auszuchecken und die Module unter `trunk\modules` (**und NICHT `trunk\bundles`!!!**) in Eclipse zu importieren. Die Module (`trunk\modules`) enthalten auch die Beispiele. Um die BluePrints für unterschiedlichen Widgets zur identifizieren, kann man sich zum Beispiel die Accessor Klasse `org.jowidgets.tools.widgets.blueprint.BPF` anschauen. Mit dieser lassen sich die BluePrints für alle Widgets erzeugen.

[1] Oder deren Verfügbarkeit über die API abfragbar ist, was jedoch ausschließlichen den FileChooser und DirectoryChooser betrifft, welche von Ajax Web Clients (RWT) nicht unterstützt werden. Dies läßt sich zum Beispiel mittels `Toolkit.getSupportedWidgets().hasFileChooser()` abfragen.

[2] Für jedes Widget existiert genau eine eigene BluePrint Schnittstelle, während unterschiedliche Widgets sich die gleiche Widget Schnittstelle Teilen können (z.B. `IFrame` für das Frame Widget (`IFrameBluePrint`) und das Dialog Widget (`IDialogBluePrint`)).

Kapitel 5

Addon Widgets - Übersicht

Addon Widgets gehören nicht zum Kern von jowidgets. Diese sind unter Umständen nicht mit allen SPI Implementierungen oder Betriebssystemen einsetzbar und sind daher in separaten Modulen untergebracht, welche mit `org.jowidgets.addons.widgets...` beginnen. Die API und die Implementierung befindet sich dabei in getrennten Modulen. Es folgt eine kurze Übersicht der Addon Widgets:

- Die **Swt-Awt Bridge Widgets** bieten einen komfortablen Zugang zur `org.eclipse.swt.awt_SWT_AWT` Bridge und lösen dabei bekannte Probleme der nativen Bridge durch (zum Teil aus dem Web bekannte) Workarounds. Falls ein UI Event Dispatching erforderlich ist, wird die BridgetSwtEventLoop benötigt. Diese sorgt dafür, dass alle Events im *selben* UI Thread stattfinden. Dazu wird u.A. das Swt Display im Swing Event Dispatcher Thread erzeugt.
- Die **OleControl, OfficeControl** und **Mediaplayer** Widgets sind nur für Windows relevant und benötigen unter Swing die BridgetSwtEventLoop. Die Widgets ermöglichen die Einbettung von allgemeinen Windows OLE Applikationen, sowie von MS-Word und MS-Excel in jowidgets.
- Der **Browser** ist ohne Bridge derzeit nur für Swt verfügbar, für Swing wird die BridgetSwtEventLoop benötigt. [1]
- Für den **PdfReader** existiert derzeit genau eine Implementierung auf Basis des Browser Widget.
- Der **DownloadButton** bietet die Möglichkeit, eine Datei auf dem Client zu Speichern. Es gibt eine Implementierung, welche das Browser Widget (z.B. für RWT), und eine, welche den FileChooser verwendet. Die zweite kann für alle SPI Implementierungen verwendet werden, die einen FileChooser unterstützen (was der RWT Web Ajax Client explizit nicht tut). Dies ist derzeit für Swt und Swing der Fall. Für SWT kann man dabei die Option auswählen, die einem mehr zusagt (Browser oder FileChooser). Mit Hilfe der Download Button API kann man somit Code zum Speichern einer Datei schreiben, der sowohl für Web Clients als auch für Rich Client funktioniert. Reicht die Funktionalität des Download Buttons aus, also zum Beispiel das Speichern einer einzelnen Datei anstatt mehrerer, sollte der Download Button dem File Chooser vorgezogen werden, da so das Feature auch für Web Clients funktioniert.

Hinweis:

[1] Mit einer JavaFx Implementierung könnte man die BridgetSwtEventLoop für Swing vermutlich überflüssig machen, da JavaFX besser in Swing integriert werden kann, als Swt.

Zum aktuellen Zeitpunkt existiert **noch nicht** für jedes der oben aufgeführten Widgets eine ausführliche Beschreibung in dieser Dokumentation. Allerdings gibt es für **alle Widgets** Beispielapplikation, welche die Verwendung demonstrieren. Möchte man ein Widget nutzen, welches nicht in diesem Dokument detailliert beschrieben ist, wird empfohlen per `References Workspace` in Eclipse für das zugehörige **BluePrint**[2] nach den Beispielen zu suchen. Dazu wird empfohlen, jowidgets komplett auszuchecken und die Module unter `trunk\modules` (**und NICHT `trunk\bundles`!!!**) in Eclipse zu importieren. Die Module (`trunk\modules`) enthalten auch die Beispiele.

Für die Widgets der Addon Module `org.jowidgets.addons.widgets.abc...xyz.api` findet man die BluePrints zum Beispiel, indem man in den API's nach der BluePrint Accessor Klasse sucht, welche normalerweise mit ...BPF endet, also zum Beispiel `BrowserBPF`, `MediaPlayerBPF`, usw..

[2] Für jedes Widget existiert genau eine eigene BluePrint Schnittstelle, während unterschiedliche Widgets sich die gleiche Widget Schnittstelle Teilen können (z.B. `IFrame` für das Frame Widget (`IFrameBluePrint`) und das Dialog Widget (`IDialogBluePrint`)).

Kapitel 6

Weiterführende Themen

Das folgende Kapitel enthält weiterführende Themen zu jowidgets.

6.1 Jowidgets Code in native Projekte integrieren

Möchte man jowidgets in einem bereits existierenden Projekt verwenden, in welchem die Applikation nicht durch einen Application Runner erzeugt wird, bzw. das Root Fenster kein `IFrame` ist, benötigt man entweder an der Stelle, wo jowidgets eingebunden werden soll, einen nativen Wrapper, oder man erzeugt ein komplettes jowidgets Root-, oder Kind Fenster innerhalb der nativen Applikation.

In beiden Fällen wird vorausgesetzt, dass für die UI Technologie, welche im nativen Projekt verwendet wird, eine SPI Implementierung von jowidgets existiert, was derzeit für Swing, Swt und Rwt der Fall ist. [1] Eine Abhängigkeit auf die SPI Implementierung muss spätestens zur Laufzeit vorhanden sein. Verwendet man die Addon Module `org.jowidgets.spi.impl.swt.addons` oder `org.jowidgets.spi.impl.swing.addons` hat man diese Abhängigkeit bereits transitiv zur Compilezeit.

Der Aspekt des *Mischens* von unterschiedlichen nativen UI Technologien, wie zum Beispiel Swing Widgets in einer Swt Applikation, wird nicht hier, sondern wird in Swt-Awt Bridge Widgets behandelt.

6.1.1 Erzeugen von Fenstern

In bestimmten Anwendungsfällen mochte man eventuell ein komplettes jowidgets Root- oder Kind Fenster erzeugen.

6.1.1.1 Erzeugen eines Root Fensters

Ein Root Fenster kann mit Hilfe des Toolkit wie folgt erzeugt werden:

```
final IFrame frame = Toolkit.createRootFrame(BPF.frame("Jowidgets Root Frame"));
```

[1] Für JavaFx existiert eine prototypische Implementierung.

6.1.1.2 Erzeugen eines Kind Fensters

Ein Kind Fenster kann mit Hilfe der GenericWidgetFactory wie folgt erzeugt werden:

```
final IDialogBluePrint dialogBp = BPF.dialog("Jowidgets dialog");
final IFrame dialog = Toolkit.getWidgetFactory().create(nativeUiReference, dialogBp);
```

Die Ui Referenz ist dabei die Referenz des nativen Vaterfensters, also zum Beispiel eine Shell oder ein JFrame.

6.1.2 Jowidgets Wrapper Factory

Eine SPI Implementierung kann für ein IFrame und ein IComposite Wrapper zur Verfügung stellen. Diese können mit Hilfe der `IWidgetWrapperFactory` erzeugt werden, welche man mit Hilfe der folgenden Methode vom Toolkit erhält:

```
IWidgetWrapperFactory getWidgetWrapperFactory();
```

Die Schnittstelle `IWidgetWrapperFactory` hat die folgenden Methoden:

```
boolean isConvertibleToFrame(final Object uiReference);

IFrame createFrame(final Object uiReference);

boolean isConvertibleToComposite(final Object uiReference);

IComposite createComposite(final Object uiReference);
```

Mit Hilfe der Methoden `isConvertibleTo...()` kann geprüft werden, ob für ein natives Widget ein Wrapper für ein IFrame oder ein IComposite erzeugt werden kann. Die create() Methoden erzeugen einen konkreten Wrapper.

Achtung: Die folgenden Punkte sollten dabei unbedingt beachtet werden:

- Wenn man ein IFrame oder ein IComposite aus einem nativen Widget wie zum Beispiel einem `JFrame` (Swing), `Shell` (swt), `JPanel` (Swing), `Composite` (Swt) erzeugt, dann wird auf dem Wrapper **nicht** automatisch dispose aufgerufen, wenn das native Widget disposed wird. Das liegt daran, dass es nicht für jedes UI Framework ein Dispose Konzept gibt. Während ein Swt Composite disposed wird, sobald man es aus seinem Container entfernt, kann ein JPanel beliebig wiederverwendet werden, nachdem man es aus seinem Container entfernt hat. Es liegt daher in der Verantwortung des Entwicklers zu entscheiden, ob und wann der Wrapper disposed werden soll. Die SPI Implementierung für SWT liefert jedoch im Addon Modul `org.jowidgets.spi.impl.swt.addons` die Utility Klasse `SwtToJoWrapper`, welche genau dieses Problem für den Standardfall löst.
- Ein Jowidgets Wrapper Widget hat keinen Parent, wenngleich einer gesetzt werden kann.
- Man sollte auf der UI Referenz, die man *wrapped* , keine zusätzlichen modifizierenden nativen Operation wie zu Beispiels dem Hinzufügen oder Entfernen von Kind Controls durchführen. Stattdessen sollte man eine **eigene** UI Referenz, welche ausschließlich dem Wrappen dient, erzeugen, eventuell durch das Verschachteln von JPanel's oder Composite's.

6.1.3 Jowidgets Code in Swt / RCP Projekte integrieren

Es folgen spezielle Informationen und Beispiele für die Integration von jowidgets Code in Projekte, welche auf dem Standard Widget Toolkit (Swt) basieren.

6.1.3.1 SwtToJoFramesSnipped

Das SwtToJoFramesSnipped demonstriert die Erzeugung von Fenstern innerhalb einer nativen SWT Applikation:

```
1  package org.jowidgets.examples.swt.snipped;
2
3  import net.miginfocom.swt.MigLayout;
4
5  import org.eclipse.swt.SWT;
6  import org.eclipse.swt.events.SelectionAdapter;
7  import org.eclipse.swt.events.SelectionEvent;
8  import org.eclipse.swt.widgets.Button;
9  import org.eclipse.swt.widgets.Display;
10 import org.eclipse.swt.widgets.Shell;
11 import org.jowidgets.api.toolkit.Toolkit;
12 import org.jowidgets.api.widgets.IFrame;
13 import org.jowidgets.api.widgets.blueprint.IDialogBluePrint;
14 import org.jowidgets.api.widgets.blueprint.IFrameBluePrint;
15 import org.jowidgets.common.types.Dimension;
16 import org.jowidgets.tools.widgets.blueprint.BPF;
17
18 public final class SwtToJoFramesSnipped {
19
20     private SwtToJoFramesSnipped() {}
21
22     public static void main(final String[] args) throws Exception {
23         final Display display = new Display();
24
25         final Shell shell = new Shell(display);
26         shell.setSize(1024, 768);
27         shell.setLayout(new MigLayout("", "[]", "[][]"));
28
29         //add button to open a jowidgets root frame
30         final Button frameButton = new Button(shell, SWT.NONE);
31         frameButton.setText("Create root frame");
32         frameButton.addSelectionListener(new SelectionAdapter() {
33
34             @Override
35             public void widgetSelected(final SelectionEvent e) {
36                 final IFrameBluePrint frameBp = BPF.frame("Jowidgets Root Frame");
37                 frameBp.setSize(new Dimension(400, 300));
38                 frameBp.setAutoDispose(true);
39                 final IFrame frame = Toolkit.createRootFrame(frameBp);
40                 frame.add(BPF.textLabel("This is a jowidgets root frame"));
41                 frame.setVisible(true);
42             }
43
44         });
45
46         //add button to open a jowidgets modal dialog
47         final Button dialogButton = new Button(shell, SWT.NONE);
48         dialogButton.setText("Create dialog");
49         dialogButton.addSelectionListener(new SelectionAdapter() {
50
51             @Override
```

```
52              public void widgetSelected(final SelectionEvent e) {
53                  final IDialogBluePrint dialogBp = BPF.dialog("Jowidgets dialog");
54                  dialogBp.setSize(new Dimension(400, 300));
55                  final IFrame dialog = Toolkit.getWidgetFactory().create(shell, dialogBp);
56                  dialog.add(BPF.textLabel("This is a jowidgets modal dialog"));
57                  dialog.setVisible(true);
58              }
59
60          });
61
62          shell.open();
63          while (!shell.isDisposed()) {
64              if (!display.readAndDispatch()) {
65                  display.sleep();
66              }
67          }
68          display.dispose();
69      }
70
71  }
```

6.1.3.2 SwtToJoWrapper

Um für ein Swt `Composite` ein `IComposite` Wrapper zu erzeugen, kann man die Utility Klasse `SwtToJoWrapper` verwenden, welche sich im Modul `org.jowidgets.spi.impl.swt.addons` befindet. Der folgende Code zeigt einen Teil der Implementierung:

```
1   public final class SwtToJoWrapper {
2
3       private SwtToJoWrapper() {}
4
5       public static IComposite create(final Composite composite) {
6           Assert.paramNotNull(composite, "composite");
7           final IComposite result = Toolkit.getWidgetWrapperFactory().createComposite(composite);
8           composite.addDisposeListener(new DisposeListener() {
9               @Override
10              public void widgetDisposed(final DisposeEvent e) {
11                  result.dispose();
12              }
13          });
14          return result;
15      }
16
17  }
```

In Zeile 8 wird auf dem SWT Composite ein `DisposeListener` hinzugefügt, der das`IComposite` Wrapper disposed, sobald das SWT Composite disposed wird. Dies ist im Normalfall genau da gewünschte Verhalten.

Das folgende Beispiel demonstriert die Verwendung:

```
1       final IComposite joComposite = SwtToJoWrapper.create(swtComposite);
```

6.1.3.3 SwtToJoCompositeSnipped

Das SwtToJoCompositeSnipped demonstriert die Erzeugung eines `IComposite` Wrapper innerhalb einer nativen SWT Applikation:

```java
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.jowidgets.api.widgets.IComposite;
import org.jowidgets.api.widgets.IInputField;
import org.jowidgets.common.widgets.controller.IInputListener;
import org.jowidgets.common.widgets.layout.MigLayoutDescriptor;
import org.jowidgets.spi.impl.swt.addons.SwtToJoWrapper;
import org.jowidgets.tools.widgets.blueprint.BPF;

public final class SwtToJoCompositeSnipped {

    private SwtToJoCompositeSnipped() {}

    public static void main(final String[] args) throws Exception {
        //create a swt display
        final Display display = new Display();

        //create a swt shell
        final Shell shell = new Shell(display);
        shell.setSize(400, 300);
        shell.setText("SwtToJo composite snipped");
        shell.setLayout(new FillLayout());

        //create a swt composite
        final Composite swtComposite = new Composite(shell, SWT.NONE);

        //create a jowidgets composite wrapper and do some jowidgets stuff
        final IComposite joComposite = SwtToJoWrapper.create(swtComposite);
        joComposite.setLayout(new MigLayoutDescriptor("[][grow]", "[]"));

        //add a label
        joComposite.add(BPF.textLabel("Name"));

        //add a input field for double values
        final IInputField<String> nameField = joComposite.add(BPF.inputFieldString(), "grow x");
        nameField.addInputListener(new IInputListener() {
            @Override
            public void inputChanged() {
                //CHECKSTYLE:OFF
                System.out.println("Hello " + nameField.getValue());
                //CHECKSTYLE:ON
            }
        });

        //open the swt shell and start event dispatching
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        display.dispose();
    }
}
```

6.1.4 Jowidgets Code in Swing Projekte integrieren

Die Integration in Swing funktioniert analog zu SWT. Es folgen die gleichen Beispiele wie für SWT jetzt für Swing.

6.1.4.1 SwingToJoFramesSnipped

Das SwingToJoFramesSnipped demonstriert die Erzeugung von Fenstern innerhalb einer nativen Swing Applikation:

```
1  package org.jowidgets.examples.swing.snipped;
2
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5
6  import javax.swing.JButton;
7  import javax.swing.JFrame;
8  import javax.swing.SwingUtilities;
9  import javax.swing.UIManager;
10
11 import net.miginfocom.swing.MigLayout;
12
13 import org.jowidgets.api.toolkit.Toolkit;
14 import org.jowidgets.api.widgets.IFrame;
15 import org.jowidgets.api.widgets.blueprint.IDialogBluePrint;
16 import org.jowidgets.api.widgets.blueprint.IFrameBluePrint;
17 import org.jowidgets.common.types.Dimension;
18 import org.jowidgets.tools.widgets.blueprint.BPF;
19
20 public final class SwingToJoFramesSnipped {
21
22     private SwingToJoFramesSnipped() {}
23
24     public static void main(final String[] args) throws Exception {
25         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
26         SwingUtilities.invokeLater(new Runnable() {
27             @Override
28             public void run() {
29                 createAndShowJFrame();
30             }
31         });
32     }
33
34     private static void createAndShowJFrame() {
35         //create the root frame with swing
36         final JFrame frame = new JFrame();
37         frame.setSize(1024, 768);
38         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39         frame.setLayout(new MigLayout("", "[]", "[][]"));
40
41         //add button to open a jowidgets root frame
42         final JButton frameButton = new JButton("Create root frame");
43         frame.add(frameButton);
44         frameButton.addActionListener(new ActionListener() {
45             @Override
46             public void actionPerformed(final ActionEvent e) {
47                 final IFrameBluePrint frameBp = BPF.frame("Jowidgets Root Frame");
48                 frameBp.setSize(new Dimension(400, 300));
49                 frameBp.setAutoDispose(true);
50                 final IFrame frame = Toolkit.createRootFrame(frameBp);
51                 frame.add(BPF.textLabel("This is a jowidgets root frame"));
52                 frame.setVisible(true);
53             }
54         });
55
56         //add button to open a jowidgets modal dialog
57         final JButton dialogButton = new JButton("Create dialog");
58         frame.add(dialogButton);
59         dialogButton.addActionListener(new ActionListener() {
```

```
60              @Override
61              public void actionPerformed(final ActionEvent e) {
62                  final IDialogBluePrint dialogBp = BPF.dialog("Jowidgets dialog");
63                  dialogBp.setSize(new Dimension(400, 300));
64                  final IFrame dialog = Toolkit.getWidgetFactory().create(frame, dialogBp);
65                  dialog.add(BPF.textLabel("This is a jowidgets modal dialog"));
66                  dialog.setVisible(true);
67              }
68          });
69
70          //show the frame
71          frame.setVisible(true);
72      }
73
74  }
```

6.1.4.2 SwingToJoCompositeSnipped

Das SwingToJoCompositeSnipped demonstriert die Erzeugung eines `IComposite` Wrapper innerhalb einer nativen Swing Applikation:

```
1   package org.jowidgets.examples.swing.snipped;
2
3   import javax.swing.JFrame;
4   import javax.swing.JPanel;
5   import javax.swing.SwingUtilities;
6   import javax.swing.UIManager;
7
8   import net.miginfocom.swing.MigLayout;
9
10  import org.jowidgets.api.widgets.IComposite;
11  import org.jowidgets.api.widgets.IInputField;
12  import org.jowidgets.common.widgets.controller.IInputListener;
13  import org.jowidgets.common.widgets.layout.MigLayoutDescriptor;
14  import org.jowidgets.spi.impl.swing.addons.SwingToJoWrapper;
15  import org.jowidgets.tools.widgets.blueprint.BPF;
16
17  public final class SwingToJoCompositeSnipped {
18
19      private SwingToJoCompositeSnipped() {}
20
21      public static void main(final String[] args) throws Exception {
22          UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
23          SwingUtilities.invokeLater(new Runnable() {
24              @Override
25              public void run() {
26                  createAndShowJFrame();
27              }
28          });
29      }
30
31      private static void createAndShowJFrame() {
32          //create the root frame with swing
33          final JFrame frame = new JFrame("SwingToJo composite snipped");
34          frame.setSize(400, 300);
35          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36          frame.setLayout(new MigLayout("", "0[grow]0", "0[grow]0"));
37
38          //create a jpanel
39          final JPanel panel = new JPanel();
40          frame.add(panel, "growx, growy");
41
```

```
42          //create a jowidgets composite wrapper and do some jowidgets stuff
43          final IComposite joComposite = SwingToJoWrapper.create(panel);
44          joComposite.setLayout(new MigLayoutDescriptor("[][grow]", "[]"));
45
46          //add a label
47          joComposite.add(BPF.textLabel("Name"));
48
49          //add a input field for double values
50          final IInputField<String> nameField = joComposite.add(BPF.inputFieldString(), "grow x");
51          nameField.addInputListener(new IInputListener() {
52              @Override
53              public void inputChanged() {
54                  //CHECKSTYLE:OFF
55                  System.out.println("Hello " + nameField.getValue());
56                  //CHECKSTYLE:ON
57              }
58          });
59
60          //show the frame
61          frame.setVisible(true);
62      }
63
64  }
```

6.2 Nativen Code in jowidgets Code integrieren

Unter gewissen Umständen kann es wünschenswert sein, nativen UI Code innerhalb von jowidgets Code zu integrieren. Dabei wird vorausgesetzt, dass die native UI Technologie der der verwendeten SPI Implementierung entspricht.

Mögliche Gründe für die Einbettung von nativem Code könnten sein:

- Es existieren komplexe native Widgets, welche man verwenden möchte und für die in jowidgets kein Pendant existiert.
- Ein Jo Widget bietet eine bestimmte Funktion nicht an, obwohl es das native Widget tut.

Hinweis:

Man sollte sich bewusst machen, dass man durch die direkte Einbettung von nativem Code später nicht mehr **einfach** in der Lage ist, das Modul oder Widget in Kombination mit einer anderen UI Technologie zu verwenden.

Anstatt nativen Code direkt einzubetten, könnte man für die oben genannten Fälle auch wie folgt vorgehen:

- Kapseln eines nativen Widgets durch eine Jo Widget Schnittstelle
- Erweitern einer existierenden Jo Widget Schnittstelle um nativ verfügbare Funktionen

Falls dieses Vorgehen nicht in Frage kommt, kann man den nativen Code direkt Einbetten, indem man die native UI Referenz verwendet.

6.2.1 Verwendung der nativen UI Referenz

Jedes Jo Widget liefert mit Hilfe der folgenden Methode die native UI Referenz:

```
    Object getUiReference();
```

Der Typ hängt dabei von der verwendeten SPI Implementierung ab. Für Basis Widgets ist dies in der Regel das direkte native Pendant, also zum Beispiel unter Swing ein `JPanel` für ein `IComposite`, ein `JButton` für ein `IButton` und so weiter. Es gibt jedoch Ausnahmen, so liefert eine `ITable` unter Swing ein `JScrollPane` welches eine `JTable` enthält und unter Swt eine `org.eclipse.swt.widget.Table`. Im Zweifelsfall sollte man einfach in der aktuellen SPI Implementierung nachschauen, oder es mittels `getClass().getName()` ausprobieren. Composite Widgets liefern meist ein `JPanel` für Swing und ein `org.eclipse.swt.widget.Composite` für Swt, aber auch hier gibt es Ausnahmen.

Man erhält als UI Referenz immer das native Root Widget, welches von der Spi für die Erzeugung des Widget angelegt wurde. Dadurch ist immer eindeutig, wo die native Widget Hierarchie für das Jo Widget beginnt. Für die Swing SPI Implementierung wäre es also nicht zulässig, die `JTable` als UI Referenz zurückzugeben. Will man eine Referenz auf die zugehörige `JTable` haben, kann man diese aus dem `JScrollPane` herausholen.

6.2.2 Verwendung der nativen UI Referenz unter Swt

Das JoToSwtSnipped demonstriert die Verwendung der nativen Swt UI Referenz in einer jowidgets Applikation:

```
1  package org.jowidgets.examples.swt.snipped;
2
3  import org.eclipse.swt.SWT;
4  import org.eclipse.swt.browser.Browser;
5  import org.eclipse.swt.widgets.Composite;
6  import org.jowidgets.api.layout.FillLayout;
7  import org.jowidgets.api.toolkit.Toolkit;
8  import org.jowidgets.api.widgets.IComposite;
9  import org.jowidgets.api.widgets.IFrame;
10 import org.jowidgets.common.application.IApplication;
11 import org.jowidgets.common.application.IApplicationLifecycle;
12 import org.jowidgets.tools.widgets.blueprint.BPF;
13
14 public final class JoToSwtSnipped implements IApplication {
15
16     @Override
17     public void start(final IApplicationLifecycle lifecycle) {
18         //create the root frame
19         final IFrame frame = Toolkit.createRootFrame(BPF.frame("JoToSwt Snipped"), lifecycle);
20         frame.setSize(1024, 768);
21         frame.setLayout(FillLayout.get());
22
23         //create a regular jo composite
24         final IComposite joComposite = frame.add(BPF.composite());
25
26         //get the native ui reference which must be a swt composite
27         //because swt SPI impl is used
28         final Composite swtComposite = (Composite) joComposite.getUiReference();
29         swtComposite.setLayout(new org.eclipse.swt.layout.FillLayout());
30
31         //create a swt browser
32         final Browser browser = new Browser(swtComposite, SWT.NONE);
33         browser.setUrl("http://www.jowidgets.org/");
34
35         //set the root frame visible
36         frame.setVisible(true);
37     }
```

```
38
39      public static void main(final String[] args) throws Exception {
40          Toolkit.getApplicationRunner().run(new JoToSwtSnipped());
41          System.exit(0);
42      }
43  }
```

6.2.3 Verwendung der nativen UI Referenz unter Swing

Das JoToSwingSnipped demonstriert die Verwendung der nativen Swing UI Referenz in einer jowidgets Applikation:

```
1   package org.jowidgets.examples.swing.snipped;
2
3   import java.awt.BorderLayout;
4
5   import javax.swing.JPanel;
6   import javax.swing.JTextPane;
7   import javax.swing.UIManager;
8
9   import org.jowidgets.api.layout.FillLayout;
10  import org.jowidgets.api.toolkit.Toolkit;
11  import org.jowidgets.api.widgets.IComposite;
12  import org.jowidgets.api.widgets.IFrame;
13  import org.jowidgets.common.application.IApplication;
14  import org.jowidgets.common.application.IApplicationLifecycle;
15  import org.jowidgets.tools.widgets.blueprint.BPF;
16
17  public final class JoToSwingSnipped implements IApplication {
18
19      @Override
20      public void start(final IApplicationLifecycle lifecycle) {
21          //create the root frame
22          final IFrame frame = Toolkit.createRootFrame(BPF.frame("JoToSwing Snipped"), lifecycle);
23          frame.setSize(800, 600);
24          frame.setLayout(FillLayout.get());
25
26          //create a regular jo composite
27          final IComposite joComposite = frame.add(BPF.composite());
28
29          //get the native ui reference which must be a JPanel
30          //because swing SPI impl is used
31          final JPanel panel = (JPanel) joComposite.getUiReference();
32          panel.setLayout(new BorderLayout());
33
34          //create a JTextPane and add it to the panel
35          final JTextPane textPane = new JTextPane();
36          panel.add(textPane);
37
38          //set the root frame visible
39          frame.setVisible(true);
40      }
41
42      public static void main(final String[] args) throws Exception {
43          UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
44          System.setProperty("apple.laf.useScreenMenuBar", "true");
45          Toolkit.getApplicationRunner().run(new JoToSwingSnipped());
46          System.exit(0);
47      }
48  }
```

6.2.4 Kapseln eines nativen Widgets durch eine Jo Widget Schnittstelle

Man definiert zum Beispiel für das komplexe native Widget W_SWT, welches man in sein Modul X einbetten möchte, eine Widgets Schnittstelle (API_Y) inklusive Setup, bzw. BluePrint wie in Erstellung eigener Widget Bibliotheken gezeigt.

Gegen diese API_Y kann man dann im Modul X implementieren, ohne Abhängigkeiten auf die native UI Technologie SWT haben zu müssen.

Man erstellt zusätzlich für die API_Y ein Impl_Y_SWT Modul, welches die nativen UI Abhängigkeiten beinhalten, und die Widget Schnittstelle mit Hilfe von Widget W_SWT adaptiert.

Will man das Modul X später mit anderen UI Technologien verwenden, muss man nur eine weitere Implementierung z.B. Impl_Y_SWING für API_Y anbieten, ohne dass man Modul X nochmal anfassen muss.

Je öfter man das native Widget W_SWT einbetten möchte, desto mehr rechtfertigt dies das im ersten Schritte aufwändigere Vorgehen.

Will man beispielsweise für das native Widget W_SWT von vornherein für Swing und Swt eine Implementierung haben haben, ohne Wigdet W_SWT neu mit Swing zu Implementieren, kann man dazu auch die Swt-Awt Bridge Widgets verwenden.

6.2.4.1 Das Addon Browser Widget

Das Browser Addon Widget `IBrowser` kann dabei als Beispiel dienen. Es besteht aus vier Modulen:

- **org.jowidgets.addons.widgets.browser.api** beinhaltet die API für ein Browser Widget
- **org.jowidgets.addons.widgets.browser.impl.swt.common** beinhalten eine Implementierung, welche die Browser Widget Schnittstellen der API mit Hilfe des nativen Swt Browser implementiert.
- **org.jowidgets.addons.widgets.browser.impl.swt** beinhaltet einen Toolkit Interceptor, welcher das Browser Widget aus dem `...impl.swt.common` Modul in der Generic Widget Factory registriert.
- **org.jowidgets.addons.widgets.browser.impl.swing** beinhaltet einen Toolkit Interceptor, welcher das Browser Widget aus dem `...impl.swt.common` Modul mit Hilfe des AwtSwtControl nach Swing *bridged* und es in der Generic Widget Factory registriert.

Der folgende Code zeigt die `BrowserFactory` aus `org.jowidgets.addons.widgets.browser.impl.swt`:

```
1  final class BrowserFactory implements IWidgetFactory<IBrowser, IBrowserBluePrint> {
2
3      @Override
4      public IBrowser create(final Object parentUiReference, final IBrowserBluePrint bluePrint) {
5          final IComposite composite = Toolkit.getWidgetFactory().create(
6              parentUiReference,
7              BPF.composite());
8          return SwtBrowserFactory.createBrowser(
9              composite,
10             (Composite) composite.getUiReference(),
11             bluePrint);
12     }
13 }
```

Der folgende Code zeigt die `BrowserFactory` aus `org.jowidgets.addons.widgets.browser.impl.swing`:

```
final class BrowserFactory implements IWidgetFactory<IBrowser, IBrowserBluePrint> {

    @Override
    public IBrowser create(final Object parentUiReference, final IBrowserBluePrint bluePrint) {
        final IAwtSwtControl awtSwtControl
            = AwtSwtControlFactory.getInstance().createAwtSwtControl(parentUiReference);
        return SwtBrowserFactory.createBrowser(
            awtSwtControl,
            awtSwtControl.getSwtComposite(),
            bluePrint);
    }

}
```

In beiden Fällen wird das Widget wie folgt mittels eines `IToolkitInterceptor` registriert:

```
final class BrowserToolkitInterceptor implements IToolkitInterceptor {

    @Override
    public void onToolkitCreate(final IToolkit toolkit) {
        final IGenericWidgetFactory widgetFactory = toolkit.getWidgetFactory();
        widgetFactory.register(IBrowserBluePrint.class, new BrowserFactory());
        toolkit.getBluePrintProxyFactory().addDefaultsInitializer(
            IBrowserSetupBuilder.class,
            new BrowserDefaults());
    }
}
```

6.2.4.2 Definition der Schnittstelle bei Verwendung der SWT - AWT Bridge

Wenn man ein natives Widget mit Hilfe der Swt-Awt Bridge Widgets integrieren möchte, sollte man bei der Definition der Schnittstelle darauf achten, den Anteil, der *gebridget* wird, möglichst **minmal** zu halten. Dies soll an einem Beispiel verdeutlicht werden:

In einem SWT Projekt soll der legacy Text Editor `MyEditorPanel`, der in Swing Implementiert ist, verwendet werden. Das `MyEditorPanel` hat eine Toolbar, mit der bestimmte Aktion ausgeführt werden können, wie zum Beispiel dem Ändern der Textfarbe oder Schriftart. Der *komplexe* Anteil der Implementierung steckt jedoch `MyEditorTextPane`, welches das `JTextPane` erweitert, und komplexe Textdekorierungen umsetzt. Dann wäre es eventuell sinnvoll, nur für das `MyEditorTextPane` die Jo Widget Schnittstelle `IMyEditorTextPane` zu definieren und für die Implementierung die SwtAwtBridge zu verwenden. Anschließend könnte man die Schnittstelle `IMyEditorPanel` definieren, wodurch der Toolbar Aspekt und eventuell weitere hinzukommen. Dieses Widget müsste man dann nur ein Mal implementieren und könnte dafür die Schnittstelle `IMyEditorTextPane` verwenden, wodurch man dann insbesondere keine Abhängigkeiten zu Swing hat. Dieses Vorgehen hat zwei Vorteile:

1. Die Toolbar wir nicht *gebridged*. Das bedeutet, die Toolbar Buttons sind unter Swing JButtons und unter Swt SwtButtons. Ein `JTextPane` sieht unter Swt weniger wie ein Fremdkörper aus, als zum Beispiel ein JButton oder eine JComboBox.

2. Will man später auf die Bridge verzichten, und die Funktion nativ für Swt portieren, muss man nur das `MyEditorTextPane` portieren, und nicht das `MyEditorPanel` mit der Toolbar und evetuell noch weiteren, UI unabhängigen Funktionen. Zieht man sowieso eine spätere Portierung in Betracht, kann man auch gleich *richtig* schneiden.

6.2.5 Erweitern einer existierenden Jo Widget Schnittstelle um nativ verfügbare Funktionen

Wenn man eine vorhandene Widget Schnittstelle erweitern möchte, gibt es folgende Möglichkeiten:

- Eventuell ist die Erweiterung auch für alle anderen SPI Implementierungen möglich und sinnvoll
  - Man führt selbst die Erweiterung durch und stellt den resultierenden *Patch* zur Verfügung, damit er integriert werden kann.[2]
  - Man wünscht sich die Erweiterung, indem man einen Issue erstellt[3]
- Eventuell ist die Erweiterung nur für bestimmte SPI Implementierungen oder Betriebssysteme möglich und sinnvoll
  - Man verfährt wie beim vorigen Punkt, nur dass das Widget als Addon Widgets entworfen wird, und somit nicht zum Kern gehört. Zudem muss man prüfen, ob man von der vorhandenen Schnittstelle ableitet, oder ob man eine neue Schnittstelle definiert.
- Eventuell kann (darf) oder will man die Erweiterung nicht veröffentlichen.
  - Man verfährt wie oben und fügt das Widget einer passenden oder neuen firmeninternen Widget Bibliothek hinzu.

6.3 Der Toolkit Interceptor - Übersicht

Ein Toolkit Interceptor wird meist für die folgenden Aufgaben verwendet:

- Registrierung oder Substitution von Images in der Image Registry
- Überschreiben der Widget Defaults in der BluePrint Proxy Factory
- Registrierung von Widgets in der Generic Widget Factory
- Registrierung von Convertern im IConverterProvider
- Zum Setzen von Toolkit Properties
- Registrierung von Convenience Methoden Implementierungen für Setup Builder in der BluePrint Proxy Factory
- Registrierung von Widget Decorators zum Dekorieren von Widgets in der Generic Widget Factory
- Registrierung von WidgetFactory Decorators zum Dekorieren der Factories von Widgets in der Generic Widget Factory

6.3.1 Der Toolkit Interceptor

Es folgt eine Beschreibung der relevanten Schnittstellen sowie konkrete Beispiele:

6.3.1.1 Die Schnittstelle IToolkitInterceptor

Die Schnittstelle `IToolkitInterceptor` hat die folgende Methode:

[2] Vorab abzustimmen was man vor hat, kann dabei nichts schaden
[3] Oder man beauftragt eine Erweiterung, was unter Umständen die Bearbeitungszeit verkürzt.

```
    void onToolkitCreate(IToolkit toolkit);
```

Diese Methode wird aufgerufen, nachdem das Toolkit erzeugt wurde, jedoch bevor die Toolkit Instanz über die Klasse Toolkit verfügbar ist. Aus diesem Grund wird die neu erzeugte Instanz als Parameter übergeben. Dadurch ist sicher gestellt, dass alle Toolkit Interceptoren vor der ersten Benutzung des Toolkit ausgeführt werden. Wenn `onToolkitCreate()` aufgerufen wird, hat jowidgets bereits seine Core Widgets, Icons und Defaults registriert.

Achtung:

Der Aufruf von `Toolkit.getInstance()` sowie allen anderen indirekten Aufrufen davon, wie zum Beispiel `Toolkit.getBluePrintProxyFactory()` führen zu einem Fehler. Stattdessen muss man zum Beispiel `toolkit.getBluePrintProxyFactory()` schreiben. Zudem sollten in einem Toolkit Interceptor **keine Widgets erzeugt werden!!!**

6.3.1.2 Toolkit Interceptor Beispiel

Das folgende Beispiel zeigt eine gekürzte Version des Toolkit Interceptor für die jo-client-platform. Der vollständige Code findet sich hier.

```
1  final class CapToolkitInterceptor implements IToolkitInterceptor {
2
3      @Override
4      public void onToolkitCreate(final IToolkit toolkit) {
5          registerWidgets(toolkit);
6          registerIcons(toolkit);
7          addDefaultsInitializer(toolkit);
8          setBuilderConvenience(toolkit);
9          registerConverter(toolkit);
10     }
11
12     @SuppressWarnings("unchecked")
13     private void registerWidgets(final IToolkit toolkit) {
14         final IGenericWidgetFactory factory = toolkit.getWidgetFactory();
15
16         factory.register(
17             IBeanTableBluePrint.class,
18             new BeanTableFactory());
19
20         //... removed some widgets in this example
21
22         factory.register(
23             IBeanLinkDialogBluePrint.class,
24             new BeanLinkDialogFactory());
25     }
26
27     private void registerIcons(final IToolkit toolkit) {
28         final IImageRegistry registry = toolkit.getImageRegistry();
29
30         registry.registerImageConstant(
31             CapIcons.TABLE_HIDE_COLUMN,
32             IconsSmall.SUB);
33
34         //... removed some icons in this example
35
36         registerImage(registry, CapIcons.NODE_CONTRACTED, "node_contracted.png");
37     }
38
39     private void registerImage(
```

```
40          final IImageRegistry registry,
41          final IImageConstant imageConstant,
42          final String relPath) {
43
44          String path = "org/jowidgets/cap/ui/icons/" + relPath;
45          final URL url = getClass().getClassLoader().getResource(path);
46
47          registry.registerImageConstant(imageConstant, url);
48      }
49
50      private void registerConverter(final IToolkit toolkit) {
51          final IConverterProvider converterProvider = toolkit.getConverterProvider();
52          converterProvider.register(IDocument.class, new DocumentConverter());
53      }
54
55      private void addDefaultsInitializer(final IToolkit toolkit) {
56          final IBluePrintProxyFactory bppf = toolkit.getBluePrintProxyFactory();
57
58          bppf.addDefaultsInitializer(
59              IBeanTableSetupBuilder.class,
60              new BeanTableDefaults());
61
62          //... removed some defaults in this example
63
64          bppf.addDefaultsInitializer(
65              ILookUpComboBoxSelectionBluePrint.class,
66              new LookUpComboBoxSelectionDefaults());
67      }
68
69      private void setBuilderConvenience(final IToolkit toolkit) {
70          final IBluePrintProxyFactory bppf = toolkit.getBluePrintProxyFactory();
71
72          bppf.setSetupBuilderConvenience(
73              IBeanTableSetupBuilder.class,
74              new BeanTableSetupConvenience());
75
76          //... removed some convenience methods in this example
77
78          bppf.setSetupBuilderConvenience(
79              IBeanRelationTreeDetailSetupBuilder.class,
80              new BeanRelationTreeDetailSetupConvenience());
81      }
82
83  }
```

6.3.1.3 Die Schnittstelle IToolkitInterceptorHolder

Um einen Toolkit Interceptor zu registrieren, benötigt man einen `IToolkitInterceptorHolder`. Die Schnittstelle sieht wie folgt aus:

```
1  public interface IToolkitInterceptorHolder {
2
3      int DEFAULT_ORDER = 2;
4
5      int getOrder();
6
7      IToolkitInterceptor getToolkitInterceptor();
8  }
```

Mit Hilfe der Order kann die Reihenfolge beeinflusst werden, in der Toolkit Interceptoren aufgerufen werden. Ein Interceptor mit einer kleineren Order wird vor einem Interceptor mit einer größeren

Order ausgeführt. Die Methode `getToolkitInterceptor()` wird erst aufgerufen, bevor der Interceptor tatsächlich aufgerufen wird, und nicht bereits bei der Registrierung des Holders.

6.3.1.4 Die Klasse AbstractToolkitInterceptorHolder

Die Klasse `AbstractToolkitInterceptorHolder` liefert eine Basisimplementierung der Schnittstelle `IToolkitInterceptorHolder`. Das folgende Beispiel zeigt die Verwendung anhand des Toolkit Interceptor Holder's des UI Security Plugin der jo-client-platform. [4]:

```
1  public final class CapSecurityUiToolkitInterceptorHolder
2      extends AbstractToolkitInterceptorHolder {
3
4      public CapSecurityUiToolkitInterceptorHolder() {
5          super(Integer.MAX_VALUE);
6      }
7
8      @Override
9      protected IToolkitInterceptor createToolkitInterceptor() {
10         return new CapSecurityUiToolkitInterceptor();
11     }
12
13 }
```

6.3.1.5 Registrierung eines Toolkit Interceptor Holder mittels Java ServiveLoader

Eine Implementierung der Schnittstelle `IToolkitInterceptorHolder` kann mit Hilfe des Java ServiceLoader Mechanismus registriert werden. Dadurch wird sichergestellt, dass der Interceptor vor der ersten Verwendung des Toolkit aufgerufen wird.

Die folgende Abbildung verdeutlicht dabei das Vorgehen beispielhaft Anhand der Swt Implementierung des Addon Browser Widgets:

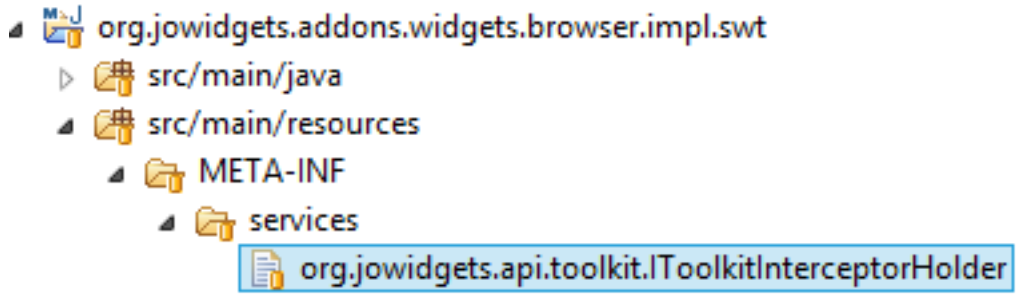


Abbildung 6.1: Toolkit Interceptor Holder mit Service Loader

Im Ordner `META-INF/services` muss eine Datei mit dem Namen `org.jowidgets.api.toolkit.IToolkitInterceptorHolder` abgelegt werden, welche den voll qualifizierten Klassennamen aller Implementierungen beinhaltet. Im Beispiel hat die Datei den folgenden Inhalt:

[4] Das UI Plugin liefert keine echten Security Aspekte, denn die liefert das Service Security Plugin. Das Plugin verhindert aber zum Beispiel die Ausführung von Diensten für die keine Rechte vorhanden sind, um die Usability zu erhöhen, oder dekoriert Widgets, so dass zum Beispiel eine BeanTable, falls man keine Leserechte hat, durch ein Composite mit Informationen darüber an den Nutzer, ersetzt wird.

```
org.jowidgets.addons.widgets.browser.impl.swt.BrowserToolkitInterceptorHolder
```

6.3.1.6 Explizite Registrierung eines Toolkit Interceptor Holder

Die Klasse `ToolkitInterceptor` liefert die folgende statische Methode zur Registrierung eines `IToolkitInterceptorHolder`:

```
public static void registerToolkitInterceptorHolder(final IToolkitInterceptorHolder holder) {...}
```

Achtung: Die explizite Registrierung muss **immer** vor der ersten Verwendung des Toolkit erfolgen, ansonsten hat sie keinen Effekt. Das explizite Registrieren sollte also nur verwendet werden, wenn man diese Bedingung garantieren kann. Man könnte zum Beispiel vermuten, dass in einem RCP Projekt ein IStartup (earlyStartup()) ein guter Zeitpunkt für die Registrierung ist. Allerdings ist unklar, ob der UI Thread erst gestartet wird, nachdem alle `earlyStartup()` Aufrufe stattgefunden haben (Die API Spezifikation macht dazu jedenfalls keine Aussage.) Falls dies nicht der Fall ist, könnte ein anderes Plugin, welches ebenfalls `IStartup` verwendet mittels `Display.asynExcec()` auf das Toolkit zugreifen, bevor der Inteceptor aufgerufen wurde. Aus diesem Grund wird davon dringend abgeraten.

6.4 Die Generic Widget Factory - Übersicht

Die Generic Widget Factory ist zum einen eine Registry für konkrete Widget Factories. Dabei wird für jeden Widget Typ (welcher durch sein BluePrint Typ definiert ist, und nicht durch die Widget Schnittstelle) genau eine IWidgetFactory registriert. Zum anderen dient die Generic Widget Factory zur Erzeugung aller Widgets. Dies wird mit Hilfe der registrierten `IWidgetFatory` Instanzen implementiert.

Die Generic Widget Factory wird durch Client Code in der Regel nur direkt bei der Erstellung eigener Widget Bibliotheken verwendet.

6.4.1 Die Generic Widget Factory Methoden

Es folgt eine Beschreibung der Methoden der Schnittstelle `IGenericWidgetFactory`:

6.4.1.1 Methoden zur Erzeugung von Widgets

Die folgenden Methoden dienen der Erzeugung von Widgets:

```
<WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
    WIDGET_TYPE create(DESCRIPTOR_TYPE descriptor);

<WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
    WIDGET_TYPE create(Object parentUiReference, DESCRIPTOR_TYPE descriptor);
```

Die erste Methode erzeugt Root Widgets, welche keinen Parent hat, die zweite Methode erzeugt ein Kind Widget, wobei die native UI Refrenz des Vaters übergeben wird. Client Code ruft die zweite Methode im Normalfall nicht direkt auf, außer bei der Erstellung eigener Widget Bibliotheken in einer `IWidgetFactory`.

Die erste Methode kann zum Beispiel für die Erzeugung eines Root Frame verwendet werden:

```
    final IFrame rootFrame = Toolkit.getWidgetFactory().create(BPF.frame());
```

Das gleiche kann man aber auch wie folgt schreiben:

```
    final IFrame rootFrame = Toolkit.createRootFrame(BPF.frame());
```

6.4.1.2 Methoden zur Registrierung und Deregistrierung von Widgets

```
    <WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
        void register(
            Class<? extends DESCRIPTOR_TYPE> descriptorClass,
            IWidgetFactory<WIDGET_TYPE, ? extends DESCRIPTOR_TYPE> widgetFactory);

    <WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
        void unRegister(Class<? extends DESCRIPTOR_TYPE> descriptorClass);
```

Die erste Methode registriert eine `IWidgetFactory` für einen definierten BluePrint Typ. Dabei darf für diesen Typ noch kein Widget registriert sein. Möchte man explizit eine Widget Factory ersetzen, um zum Beispiel eine vorhandene Widget Implementierung auszutauschen, muss man den BluePrint Typ vorab per `unRegister()` deregistrieren.

Die Schnittstelle `IWidgetFactory` ist wie folgt definiert:

```
1  public interface IWidgetFactory
2      <WIDGET_TYPE extends IWidgetCommon,
3      DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>> {
4
5      WIDGET_TYPE create(Object parentUiReference, DESCRIPTOR_TYPE descriptor);
6  }
```

Das folgende Beispiel zeigt eine typische Implementierung:

```
1  final class BeanFormFactory<BEAN_TYPE> implements
2      IWidgetFactory<IBeanForm<BEAN_TYPE>, IBeanFormBluePrint<BEAN_TYPE>> {
3
4      @Override
5      public IBeanForm<BEAN_TYPE> create(
6          final Object parentUiReference,
7          final IBeanFormBluePrint<BEAN_TYPE> bluePrint) {
8
9          final IComposite composite = Toolkit.getWidgetFactory().create(
10             parentUiReference,
11             BPF.composite());
12
13         return new BeanFormImpl<BEAN_TYPE>(composite, bluePrint);
14     }
15 }
```

In Zeile 9 wird mit Hilfe der Generic Widget Factory ein Composite erzeugt. Dieses wird in Zeile 13 der `BeanFormImpl` übergeben. Diese verwendet das `IComposite` um das Formular darin darzustellen.

6.4.1.3 Abfragen einer Widget Factory

Die folgende Methode liefert eine `IWidgetFactory` für einen BluePrint Typ:

```
<WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
  IWidgetFactory<WIDGET_TYPE, DESCRIPTOR_TYPE> getFactory(
      Class<? extends DESCRIPTOR_TYPE> descriptorClass);
```

Falls für den BluePrint Typ keine Factory registriert ist, wird `null` zurückgegeben.

6.4.1.4 Dekorieren von Widgets

Die folgenden Methoden können verwendet werden, um alle Widgets oder Widget Factories für einen bestimmten Typ zu dekorieren:

```
    <WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
        void addWidgetDecorator(
            Class<? extends DESCRIPTOR_TYPE> descriptorClass,
            IDecorator<WIDGET_TYPE> decorator);

    <WIDGET_TYPE extends IWidgetCommon, DESCRIPTOR_TYPE extends IWidgetDescriptor<WIDGET_TYPE>>
        void addWidgetFactoryDecorator(
            Class<? extends DESCRIPTOR_TYPE> descriptorClass,
            IDecorator<IWidgetFactory<WIDGET_TYPE, ? extends DESCRIPTOR_TYPE>> decorator);
```

Weitere Information finden sich im Abschnitt Austauschen und Dekorieren von Widgets.

6.4.1.5 Widget Factory Listener

Die folgenden Methoden können verwendet werden, um einen Widget Factory Listener hinzuzufügen und zu entfernen:

```
    void addWidgetFactoryListener(IWidgetFactoryListener listener);

    void removeWidgetFactoryListener(IWidgetFactoryListener listener);
```

Ein `IWidgetFactoryListener` hat die folgende Methode:

```
    void widgetCreated(IWidgetCommon widget);
```

Diese Methode wird immer aufgerufen, wenn ein Widget erzeugt wird. Dies kann zum Beispiel für Debugging Zwecke oder für JUnit Tests interessant sein. Der folgende Listener wurde einer Applikation vor dem Start hinzugefügt:

```
1  public final class WidgetFactoryListener implements IWidgetFactoryListener {
2
3      private final Set<IWidget> allWidgets;
4      private final Set<IWidget> undisposedWidgets;
5
6      public WidgetFactoryListener() {
7          this.allWidgets = new HashSet<IWidget>();
8          this.undisposedWidgets = new HashSet<IWidget>();
9      }
```

```
10
11      @Override
12      public void widgetCreated(final IWidgetCommon widgetCommon) {
13          if (widgetCommon instanceof IWidget) {
14              final IWidget widget = (IWidget) widgetCommon;
15              allWidgets.add(widget);
16              undisposedWidgets.add(widget);
17              widget.addDisposeListener(new IDisposeListener() {
18                  @Override
19                  public void onDispose() {
20                      undisposedWidgets.remove(widget);
21                  }
22              });
23          }
24      }
25
26      @Override
27      public String toString() {
28          return "allWidgetsCount="
29                  + allWidgets.size()
30                  + ", undisposedWidgetsCount="
31                  + undisposedWidgets.size();
32      }
33
34  }
```

Nach dem Beenden wurde die `toString()` Methode aufgerufen. Das führte zu folgender Ausgabe:

```
    allWidgetsCount=48, undisposedWidgetsCount=0
```

Anschließend wurde in der Implementierung von `IContainer` ein Fehler eingebaut, so dass die Kinder eines Containers nicht mehr disposed wurden. Nach erneutem Start und Beenden ergab sich die folgende Ausgabe:

```
    allWidgetsCount=48, undisposedWidgetsCount=45
```

6.5 Widget Defaults

TODO

6.5.1 Widget Defaults überschreiben

TODO

6.6 Austauschen und Dekorieren von Widgets - Übersicht

Neben dem globalen Anpassen der Widget Defaults ist es auch möglich, Widget Implementierungen komplett auszutauschen oder zu dekorieren. Das globale Ändern eines Widgets kann durch viele Aspekte motiviert sein:

Beispiel 1:

Man möchte in einer Applikation Widgets, für welche man keine Leserecht besitzt, durch *Fake* Widgets ersetzen, welche die gleiche Schnittstelle implementieren, jedoch anstatt dem original Widget ein Label anzeigen, das Nutzter darauf hinweist, dass ein Recht fehlt. Zudem sollen keine Daten von dem zugehörigen Dienst eingelesen werden, weil dies sowieso zu einer Security Exception führen würde. Die jo-client-platform nutzt diese Möglichkeit zur Dekoration der BeanTable, des BeanRelationTree und weiteren Widgets. Der zugehörige `IToolkitInterceptor` findet sich hier

Beispiel 2:

Angenommen man möchte ein Testtool entwickeln, welches die Möglichkeit bieten soll, bestimmte Methodenaufrufe auf bestimmten Widgets eines bestimmten Typs zu protokollieren, während man eine Applikation bedient, um daraus Eingabedaten für einen automatisierten JUnitTest zu erstellen. Dann könnte man diese Widgets einfach durch einen Wrapper ersetzen, welcher die gewünschte Protokollierung durchführt, nachdem er die Methode auf dem Original Widget aufgerufen hat. Im Rahmen einer Bachelorarbeit wurde ein solches Testtool entwickelt.

Beispiel 3:

Ein weiterer Anwendungsfall könnte sein, dass man für das Debugging beim Layouten von verschachtelten Composites einen farbigen Border um alle Composites zeichnen möchte, um zu sehen, welcher Container in welche Schachtelungsebene welche Größe hat.

Im nächsten Abschnitt folgt ein weiteres, zusammenhängendes Beispiel.

6.6.1 Austauschen und Dekorieren von Widgets mit Hilfe der Generic Widget Factory

Die Generic Widget Factory bietet die folgenden Möglichkeiten zum Austauschen und Dekorieren von Widgets:

- Eine registrierte Widget Implementierung durch eine andere ersetzen
- Eine registrierte Widget Implementierung dekorieren
- Eine registrierte Widget Factory dekorieren

6.6.1.1 Beispiel

Diese Möglichkeiten sollen Anhand eines durchgängigen Beispiels erläutert werden:

Für ein existierendes Produkt wird bei einem Neukunden Akquise gemacht. Das Produkt bietet die Möglichkeit, Entitäten und deren Beziehungen individuell zu definieren, um diese in einer Datenbank zu verwalten und Workflows auf den Daten durchzuführen. Nachdem der Kunde eine Demoversion des Produktes getestet, und dabei einige für ihn spezifische Masken für Maschinenteile und Baugruppen modelliert hat, kommt der folgende Wunsch auf:

In seinem System werden sehr viele komplexe technische Parameter Bezeichnungen und Identifikationsnummern als Attributname für Teile und Baugruppen verwendet. Der Kunde würde diese Informationen gelegentlich gerne in einem Legacy System für Suchanfragen verwenden. Dies gilt sowohl für die Attributnamen als auch für den Inhalt. Da die Attributnamen derzeit mit Hilfe von Text Label Widgets angezeigt werden, kann man Sie nicht in die Zwischenablage kopieren, sondern muss diese abschreiben. Die folgende Abbildung zeigt eine für den Kunden typische Datenmaske für ein Maschinenteil[5]:

[5] Diese ist frei erfunden. Jede Ähnlichkeit mit realen Produkten ist rein zufällig.

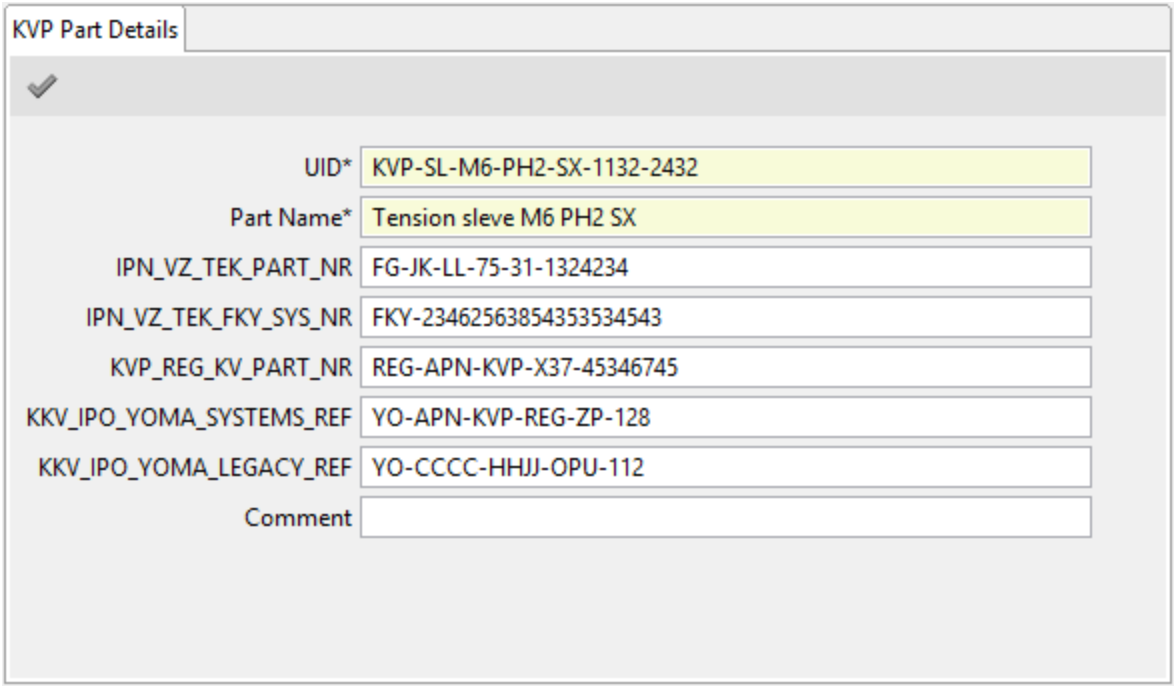


Abbildung 6.2: Beispiel Eingabemaske

Die Anpassung sollte so kostengünstig wie möglich sein (Kundenwunsch), und möglichst wenig Auswirkungen auf das zugrundeliegende Produkt und somit Systeme bei andere Kunden haben (Wunsch des Auftragnehmer). Dem Kunden werden die folgende Lösungsmöglichkeiten angeboten:

- Alle Text Label werden durch ein ITextField ersetzt, welche wie ein Text Label aussehen, weil sie keinen Border haben und die Hintergrundfarbe vom Parent Container erben. Diese sind readonly, können also nicht geändert werden, es ist aber möglich, den Text zu markieren und so in die Zwischenablage zu kopieren. Dies könnte umgesetzt werden, indem die registrierte Widget Implementierung für `ITextLabelBluePrint` durch eine andere Implementierung ersetzt wird.

- Alle Text Label erhalten ein Kontextmenü, welches eine *Copy to clipboard* Aktion enthält. Dies könnte umgesetzt werden, indem für `ITextLabelBluePrint` die registrierte Implementierung des Widget dekoriert wird.

Der Kunde stellt beide Lösungen einer Gruppe von Benutzern vor, wobei der eine Teil die Erste, der andere Teil die zweite Lösung bevorzugt. Der Kunde fragt, ob man auf einfache und kostengünstige Weise beide Lösungen anbieten können.

Es wird angeboten, das man in den Einstellungen einen Parameter hinzufügt, der das Verhalten festlegt. Um die Lösung möglichst kostengünstig zu halten, wird vom Kunden akzeptiert, das sich die Änderung nur auf neu erstellte Masken und nicht auf bereits erzeugte auswirken, und somit u.U. ein Neustart der Client Applikation notwendig ist. Da laut Nutzer dieser Parameter nach einmaliger Konfiguration in der Regel nicht mehr geändert wird, stellt dies für den Kunden kein Problem dar. Diese Lösung könnte umgesetzt werden, indem für `ITextLabelBluePrint` die Widget Factory dekoriert wird.

6.6.1.2 Widget Implementierungen austauschen

Die folgende Klasse implementiert die Schnittstelle `ITextLabel` mit Hilfe eines Textfeldes:

```
public final class TextFieldLabel extends ControlWrapper implements ITextLabel {

    private final ITextControl textField;

    public TextFieldLabel(final ITextControl textField) {
        super(textField);
        this.textField = textField;
    }

    @Override
    public void setFontSize(final int size) {
        textField.setFontSize(size);
    }

    @Override
    public void setFontName(final String fontName) {
        textField.setFontName(fontName);
    }

    @Override
    public void setMarkup(final Markup markup) {
        textField.setMarkup(markup);
    }

    @Override
    public void setText(final String text) {
        textField.setText(text);
    }

    @Override
    public String getText() {
        return textField.getText();
    }

}
```

Die meisten Methoden werden vom [Control Wrapper] geerbt, die anderen werden an das `textField` delegiert.

Die folgende Factory erzeugt Instanzen dieser TextFieldLabel's:

```
public final class TextFieldLabelFactory
    implements IWidgetFactory<ITextLabel, ITextLabelBluePrint> {

    @Override
    public ITextLabel create(
        final Object parentUiReference,
        final ITextLabelBluePrint textLabelBp) {

        final ITextFieldBluePrint textFieldBp = BPF.textField();

        textFieldBp.setSetup(textLabelBp);
        textFieldBp.setBorder(false).setEditable(false).setInheritBackground(true);

        final ITextControl textField = Toolkit.getWidgetFactory().create(
            parentUiReference,
            textFieldBp);

```

```
18          return new TextFieldLabel(textField);
19      }
20
21  }
```

In Zeile 9 wird ein neues BluePrint für ein Text Field erzeugt. In Zeile 11 wird das Setup des Textlabels auf dem BluePrint des Texfeldes gesetzt. Dabei werden alle Properties, welche das Textfeld und das Textlabel gemeinsam haben (wie zum Beispiel die Vordergrundfarbe, die Schriftart, etc,) auch für das Textfeld übernommen. In Zeile 14 wird dann ein Texfeld mit Hilfe des `textFieldBp` erzeugt. Dieses wird der Klasse `TextFieldLabel` übergeben, welche die Schnittstelle `ITextLabel` implementiert (sie weiter oben).

Der folgende Code tauscht in der Generic Widget Factory die aktuelle Implementierung für Text Labels durch die `TextFieldLabelFactory` aus. Dies passiert mit Hilfe eines Toolkit Interceptors:

```
1  @Override
2  public void onToolkitCreate(final IToolkit toolkit) {
3      toolkit.getWidgetFactory().unRegister(ITextLabelDescriptor.class);
4
5      toolkit.getWidgetFactory().register(
6          ITextLabelDescriptor.class,
7          new TextFieldLabelFactory());
8  }
```

Die folgende Abbildung zeigt den Effekt:

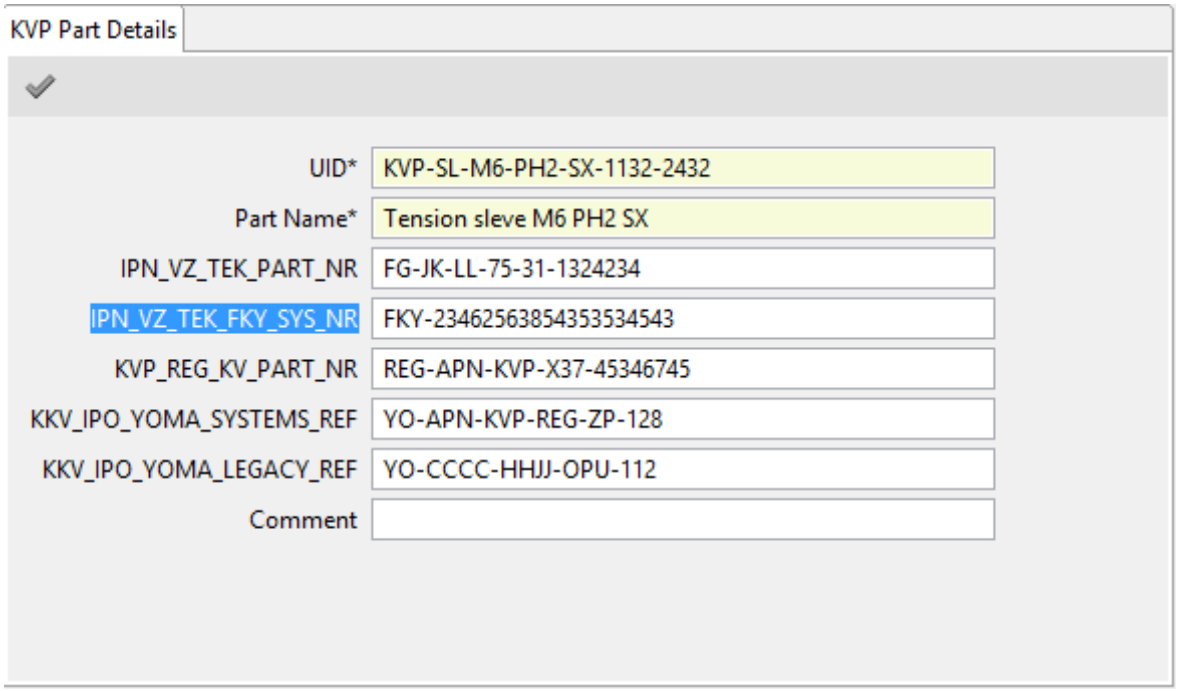


Abbildung 6.3: TextFieldLabel

Durch dass globale Austauschen der Label Implementierung wird nun für alle Labels ein Textfeld für die Darstellung verwendet. Insbesondere lassen sich die Attribute nun markieren und zum Beispiel per STRG-C in die Zwischenablage kopieren.

6.6.1.3 Widget Implementierungen dekorieren

Bei der zweiten Variante soll allen Labels ein Kontextmenü hinzufügen, welche eine Copy Action enthält, die den Text des Labels in die Zwischenablage kopiert. Um dies umzusetzen wird zuerst ein `IDecorator<ITextLabel>` wie folgt implementiert:

```
public final class LabelPopupDecorator implements IDecorator<ITextLabel> {

    @Override
    public ITextLabel decorate(final ITextLabel original) {
        final IMenuModel copyMenu = new MenuModel();

        final IActionItemModelBuilder copyActionBuilder = ActionItemModel.builder();
        copyActionBuilder
            .setText("Copy to Clipboard")
            .setToolTipText("Copies the label text to the system clipboard")
            .setIcon(IconsSmall.COPY);

        final IActionItemModel copyAction = copyMenu.addItem(copyActionBuilder);
        copyAction.addActionListener(new IActionListener() {
            @Override
            public void actionPerformed() {
                Clipboard.setContents(new StringTransfer(original.getText()));
            }
        });

        original.setPopupMenu(copyMenu);
        return original;
    }
}
```

Der Dekorierer wird dann mit Hilfe eines Toolkit Interceptors wie folgt hinzugefügt:

```
@Override
public void onToolkitCreate(final IToolkit toolkit) {
    toolkit.getWidgetFactory().addWidgetDecorator(
        ITextLabelDescriptor.class,
        new LabelPopupDecorator());
}
```

Die folgende Abbildung zeigt den Effekt:

Alle Labels haben jetzt ein Kontextmenü zum Kopieren des Labelinhalts.

6.6.1.4 Widget Factories dekorieren

Nun soll noch gezeigt werden, wie man mit Hilfe eines Widget Factory Decorator je nach Konfiguration die eine oder andere Variante wählen kann.

```
public final class LabelFactoryDecorator
    implements IDecorator<IWidgetFactory<ITextLabel, ITextLabelBluePrint>> {

    private final LabelPopupDecorator labelPopupDecorator;
    private final TextFieldLabelFactory textFieldLabelFactory;

    //will be injected
    private ILabelConfig labelConfig;

```

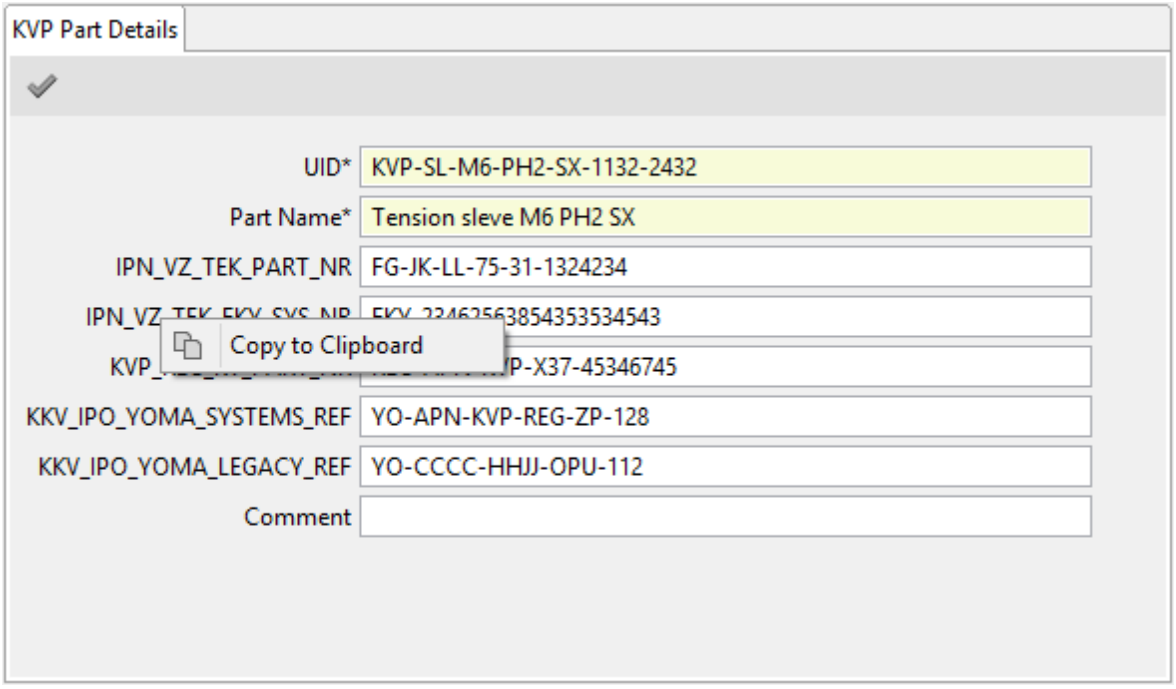


Abbildung 6.4: LabelPopupDecorator

```
10      public LabelFactoryDecorator() {
11          this.textFieldLabelFactory = new TextFieldLabelFactory();
12          this.labelPopupDecorator = new LabelPopupDecorator();
13      }
14
15      @Override
16      public IWidgetFactory<ITextLabel, ITextLabelBluePrint> decorate(
17          final IWidgetFactory<ITextLabel, ITextLabelBluePrint> originalFactory) {
18
19          //no config or default type so use original
20          if (labelConfig == null || LabelType.DEFAULT == labelConfig.getLabelType()) {
21              return originalFactory;
22          }
23          //use TextFieldLabel
24          else if (LabelType.TEXT_FIELD == labelConfig.getLabelType()) {
25              return textFieldLabelFactory;
26          }
27          //use copy action
28          else if (LabelType.COPY_ACTION == labelConfig.getLabelType()) {
29              return new IWidgetFactory<ITextLabel, ITextLabelBluePrint>() {
30                  @Override
31                  public ITextLabel create(
32                      final Object parentUiReference,
33                      final ITextLabelBluePrint descriptor) {
34
35                      //create the original widget with the original factory
36                      final ITextLabel originalWidget = originalFactory.create(
37                          parentUiReference,
38                          descriptor);
39
40                      //decorate the popup menus
41                      return labelPopupDecorator.decorate(originalWidget);
42                  }
```

```
            };
        }
        else {
            throw new IllegalStateException(
                "Lable type '" + labelConfig.getLabelType() + "' is not supported.");
        }
    }

}
```

Dieser Dekorierer kann wie folgt registriert werden:

```
@Override
public void onToolkitCreate(final IToolkit toolkit) {
    toolkit.getWidgetFactory().addWidgetFactoryDecorator(
        ITextLabelDescriptor.class,
        new LabelFactoryDecorator());
}
```

Das Überschreiben und Dekorieren aus den beiden vorigen Abschnitten wird dadurch obsolet.

6.7 Erstellung eigener Widget Bibliotheken

TODO

6.8 Jowidgets Classloading

TODO

6.9 Jowidgets und RCP

TODO