Plus CD!

Mike Keith über OSGi, Gemini und JDK 8 » 86

1.11 Deutschland € 9,80 Österreich € 10,80, Schweiz sFr 19,20 S&S

eclipse MAGAZIN

www.eclipse-magazin.de

» EXKLUSIV AUF CD:

› **Gerrit Code Review:**
Das webbasierte Code-Review-System für Git Repositories

› Codan: Statische Analyse für C

› Eclipse Gemini

› Mercurial 1.7

› Graphiti 0.7.0

Verteilte Versionierung mit EGit

» Von Subversion nach Git

» Teamarbeit mit EGit und Gerrit Code Review

» Mercurial – die Alternative?

SWT küsst Qt » 80

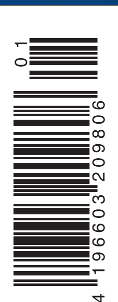
Die neue SWT-Plattform SWT/Qt

Graphiti » 67

Grafische Modelleditoren leicht gemacht

Scala IDE für Eclipse » 60

Der Eclipse-Auftritt in Scala



Datenträger enthält Info- und Lehrprogramme gemäß § 14 JuSchG

Angepackt: e4-Tutorial » 35

Dependency Injection und OSGi Declarative Services

Eclipse Client Platform » 74

Mit einem Klick zur eigenen EMF-Anwendung

SWT/Qt ist eine neue SWT-Plattform mit umfangreichen Stylingmöglichkeiten

SWT mit Stil

Die SWT-Plattform bietet seit mehreren Jahren eine solide Basis für die Erstellung von Desktopanwendungen. Mittlerweile gibt es viele erfolgreiche Anwendungen, die mit dieser Plattform erstellt wurden, z. B. die Eclipse IDE. Durch die Verwendung der native Controls der jeweiligen Betriebssysteme bietet SWT eine hohe Geschwindigkeit und Anwendungen, die weitestgehend über einen „native Look“ verfügen. Eine Umgestaltung der Anwendung ist jedoch nur begrenzt möglich. Hier setzt SWT/Qt an, um dem Entwickler umfangreiche Stylingmöglichkeiten zu bieten.

von Jürgen Becker

Um das Aussehen von SWT-Anwendungen zu ändern, z. B. für ein Branding, stößt der Entwickler bei den meisten SWT-Plattformen schnell an die Grenzen. So lassen sich oft nur die Hintergrund- bzw. Vordergrundfarben und der Schrifttyp ändern. Gradienten zu verwenden oder das Padding eines Widgets zu ändern, ist oft nicht möglich. Eine Lösung für dieses Problem sind *Custom Widgets* (Nebula [1], Lotus [2] etc.), die ein umfangreicheres Styling ermöglichen. Damit programmiert der Entwickler jedoch nicht mehr gegen das SWT API – er muss das zusätzliche API erlernen und ist auf dieses dann festgelegt. Das *Eclipse-e4-Projekt* [3] verfolgt unter anderem ein ähnliches Ziel und bietet für Standard-Widgets ein Styling per CSS an. Bisher sind die Möglichkeiten damit noch sehr begrenzt. So lassen sich z. B. Menüs, Buttons und Tabellenüberschriften nicht umgestalten. Die von der compeople AG [4] entwickelte *SWT-/Qt-Plattform* geht einen ähnlichen Weg wie e4, bietet jedoch weit umfangreichere Stylingmöglichkeiten. Als Basis dient das Qt-Framework [5] von Nokia. Es ist für alle verbreiteten Betriebssysteme (Windows, OS X, Linux) verfügbar und eine erprobte und stabile Basis, z. B. für den KDE-Desktop. Für Entwickler, die den KDE-Desktop verwenden, hat eine auf Qt basierende SWT-Plattform den Vorteil, dass sich die Eclipse IDE damit optisch besser integrieren lässt als die GTK-basierte Version.

SWT-/Qt-Aufbau

Qt ist in C++ geschrieben, daher bedarf es einer Zwischenschicht, die per *Java Native Interface* (JNI) den Zugriff ermöglicht. Diese kann per Hand geschrieben werden, was aufwändig und fehleranfällig ist. Nokia

bietet mit dem *Qt-Jambi-Projekt* eine fertige Library für den Zugriff von Java auf Qt an. Ein Großteil von Qt Jambi wird von einem Generator erzeugt, der für die benötigten C++-Klassen eine entsprechende Java-Klasse und den zugehörigen JNI-Code erzeugt. Der Entwickler muss sich dank Qt Jambi keine Gedanken über Speicherverwaltung oder die Freigabe von Ressourcen machen: Qt Jambi sorgt bei der Java Garbage Collection automatisch für die Freigabe der C++-Objekte. Dies vereinfacht die Benutzung von Qt erheblich. Es ist sogar möglich, in Java von Qt Widgets abgeleitete Klassen zu erstellen.

Für den SWT-Entwickler bleibt dies verborgen – SWT/Qt hält sich vollständig an das SWT API. Durch ein einfaches Austauschen der SWT Jars erhält der Entwickler die wie folgt beschriebenen, neuen Funktionalitäten. Das heißt, dass auch bestehende Anwendungen, wie anhand der Eclipse IDE gezeigt wird, ohne Änderungen zu einem neuen Styling kommen (**Abb. 1**).

Qt-Styling

Qt bietet eine Reihe von Möglichkeiten, um Anwendungen zu stylen. Da Qt die nativen Controls der einzelnen Betriebssysteme nicht nutzt, sondern die Controls selbst rendert, werden Styles mitgeliefert, die dem native Look and Feel nachempfunden sind. Im KDE-Umfeld gibt es eine ganze Reihe weiterer solcher in C++ geschriebener Styles. Seit der Version 4.2 kann eine Qt-Anwendung auch per *Qt Style Sheets* umgestaltet werden. Die Qt Stylesheets sind an *HTML Cascading Style Sheets* (*CSS*) angelehnt. Die meisten Controls beherrschen das bekannte *Box Model* [6], mit Margins, Border, Padding und dem Content in der Mitte. Style Sheets können einzeln pro Widget oder global für die ganze Anwendung gesetzt werden. Bei SWT/Qt wird dies über die *setData(...)*-Methode der Widgets bzw. des Displays angeboten. Durch die Verwendung der *setData*-Methode wird keine API-Erweiterung benötigt und die Anwendung ist nicht an die SWT-/Qt-Plattform gebunden. Sie funktioniert weiterhin mit allen SWT-Plattformen. Um zum Beispiel für alle Widgets global ein Padding von 5 Pixeln zu setzen, kann dies über das Display erfolgen. Für einzelne Instanzen wie die eines Buttons sieht es fast genauso aus:

```
display.setData("stylesheet", "* { padding: 5px; }" );

Button button = new Button(parent, SWT.PUSH);
button.setData("stylesheet", "QPushButton { padding: 5px; }" );
```

An diesem Beispiel ist ersichtlich, dass zum jetzigen Zeitpunkt noch die Kenntnis der verwendeten Qt-Klassen nötig ist, um SWT-Anwendungen zu stylen. Es gibt verschiedene Lösungsansätze, um dies zu vereinfachen. Diese sind jedoch noch nicht umgesetzt.

Style-Sheet-Beispiele

Anhand einer Reihe kleiner Beispiele soll gezeigt werden, welche Änderungen mit Style Sheets möglich sind. Eine vollständige Dokumentation findet sich in der Qt-Dokumentation [7]. Ein Button mit einem 5 Pixel Padding, mit vertikalem Gradienten, abgerundeten Ecken und „bold"-Font kann mit dem in Listing 1 dargestellten Style Sheet erzeugt werden.

Für den *hover*-Zustand kann, wenn die Maus über den Button bewegt wird, der Hintergrund auf Orange geändert werden (**Abb. 2**):

```
QPushButton:hover {
  background-color: #fedb74;
}
```

Neben *hover* gibt es noch weitere so genannte *Pseudo-States*, zum Beispiel *selected*, *on* etc. Jeder einzelne Zustand kann getrennt gestylt werden. Für alle Labels wird eine schwarze fette Schrift und ein Border am unteren Rand gesetzt (**Abb. 3**):

```
QLabel {
  color: #000;
  font: bold large "Arial";
  border-width: 0px 0px 1px 0px;
  border-color: #000;
  border-style: solid;
}
```

Anhand des Text Widgets soll gezeigt werden, wie man auf dynamische Properties, die mittels *Widget.setData()* gesetzt wurden, im Style Sheet Bezug nimmt. Als Beispiel dienen die in Formularen oft verwendeten Markierungen für Pflichtfelder (*mandatory*) und Falscheingaben (*error*). Pflichtfelder sollen einen gelben Hintergrund, Textfelder mit falschem Inhalt einen roten Rahmen erhalten. Bei den Instanzen des Text Widgets wird dies wie folgt gesetzt. Die Proper-

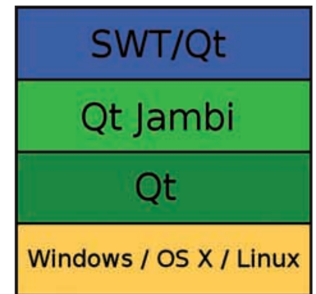


Abb. 1: SWT-/Qt-Architektur

Abb. 2: Umgestaltete Buttons

Abb. 3: Beispiellabel

Abb. 4: Dynamische Properties im CSS

Listing 1

```
QPushButton {
  padding: 5px;
  background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
    stop: 0.0 #f5f9ff,
    stop: 1.0 #c0dbff);
  color: #006aff;
  font-weight: bold;
  border: 1px solid #6593cf;
  border-radius: 4px;
}
```

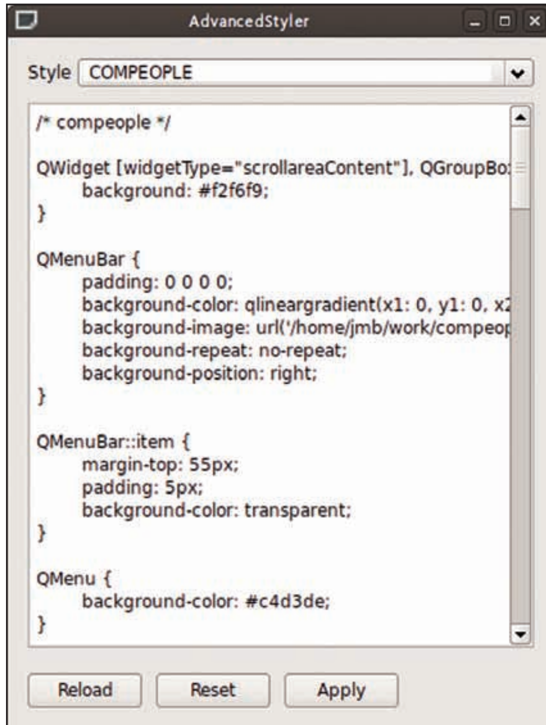


Abb. 5: Advanced-Styler-Plug-in

ties können jederzeit, zum Beispiel während einer Validierung, geändert werden (Listing 2).

Umgestaltung der Eclipse IDE

Nun soll ein umfangreicheres Style Sheet für die Eclipse-IDE erstellt werden und hierbei die IDE ein Branding der compeople AG erhalten. Zu diesem Zweck sind eine Reihe der Farben anzupassen und oben rechts ein Logo im Menü zu hinterlegen. Um das Ändern und Ausprobieren eines Style Sheets zu vereinfachen, wurde ein Eclipse-Plug-in entwickelt, das ein interaktives Editieren von Style Sheets direkt aus RCP-Anwendungen heraus ermöglicht (**Abb. 5**). Das einfache Ausprobieren von Änderungen beschleunigt die Arbeit deutlich, wenn Entwickler und Interaktionsdesigner zusammen am Bildschirm UI-Ideen durchspielen.

Als Erstes wird das Logo oben rechts im Menü platziert und das Menü mit einem Gradienten versehen (**Abb. 6**):

```
QMenuBar {
  padding-top: 70px;
  background-color: qlineargradient(x1: 0, y1: 0, x2: 1, y2: 0,
```

Listing 2

```
Text text1 = new Text(parent, SWT.SINGLE);
text1.setData("mandatory", "true");

Text text2 = new Text(parent, SWT.SINGLE);
text2.setData("error", "true");
```

Im Stylesheet wird auf die Properties Bezug genommen:

```
QLineEdit[mandatory='true'] {
background-color: lightyellow;
...
}

QLineEdit[error='true'] {
  border: 2px solid red;
}
```

Listing 3

```
QMenuBar {
  padding: 0px;
  ...
}
QMenuBar::item {
  margin-top: 55px;
  padding: 5px;
  background-color: transparent;
}
QMenu {
  background-color: #c4d3de;
}
QMenuBar::item:selected, QMenu::item:selected {
  background: #fff;
  color: #000;
}
QMenu {
  background-color: #c4d3de;
}
QMenuBar::item:selected, QMenu::item:selected {
  background: #fff;
  color: #000;
}
```

Listing 4

```
QPushButton {
  border: 1px solid #00416a;
  border-radius: 4px;
  padding: 4px;
  background-color: #c4d3de;
  color: #00416a;
  font-weight: bold;
}
QPushButton:hover {
  border: 1px solid #ff8c1a;
}
QPushButton:disabled {
  color: #aaa;
}
```

```
        stop : 0.0 #c4d3de, stop: 0.7 #fff);
        background-image: url('.../icons/compeople-logo.gif');
        background-repeat: no-repeat;
        background-position: right;
    }
```

Die freischwebenden Menüpunkte in **Abbildung 6** wirken etwas verloren. Sie sollen am unteren Rand platziert werden. Als weitere Anforderung ist zugleich auch die Farbe für selektierte Menüpunkte und Submenüs zu ändern (**Abb. 7**), (Listing 3).

Der Großteil der Flächen lässt sich auf einfache Art und Weise mit einer passenderen Hintergrundfarbe versehen:

```
    QWidget [widgetType="scrollareaContent"], QGroupBox {
        background: #f2f6f9;
    }
```

Für Buttons werden eine flache Optik, ein *bold*-Font und leicht abgerundete Ecken gewählt. Der gerade selektierte Button erhält einen orangen Border, ein deaktivierter Button eine graue Schrift (**Abb. 8**), (Listing 4).

Es lassen sich ebenso auch Bilder für den Hintergrund setzen. In Verbindung mit den oben genannten Properties können zum Beispiel. Standard-Buttons für

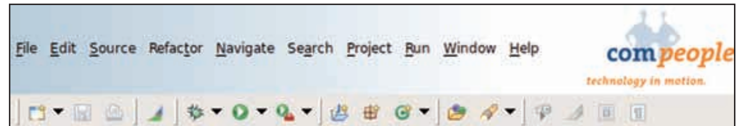


Abb. 6: Menüumgestaltung 1. Schritt

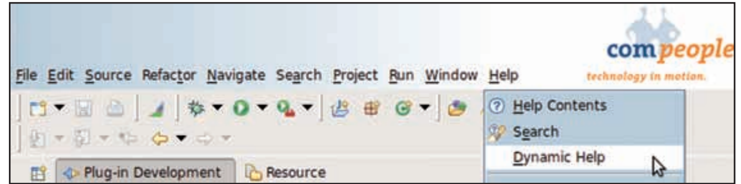


Abb. 7: Menüumgestaltung 2. Schritt

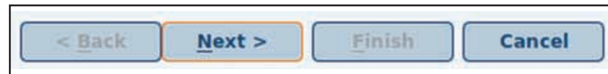


Abb. 8: Buttons im Dialog

Dialoge mit Icons versehen werden. Dies setzt voraus, dass mittels *setData(...)* die Properties gesetzt wurden:

```
    *[purpose='cancel'] {
        background: url('icons/cancel.gif');
    }
    *[purpose='ok'] {
        background: url('icons/ok.gif');
    }
```

Anzeige

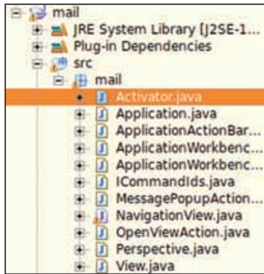


Abb. 9: Package Explorer

Nun soll ein komplizierteres Widget, der Tree, näher betrachtet werden. Der Tree setzt sich aus einer Reihe von Controls zusammen, von denen hier ein Teil umgestaltet werden soll. Zum einen gibt es den eigentlichen Baum, bei dem die Hintergrundfarbe geändert werden soll. Für Selektionen ist eine auffallendere Farbe zu setzen. Um Platz zu sparen, wird darüber hinaus das Padding reduziert (**Abb. 9**), was durch folgendes Listing erreicht wird:

```
QTreeWidget {
    background: #ffffec;
    selection-background-color: #ff8c1a;
    padding: 0px;
}
```

Da der Tree auch als Tree-Table Verwendung findet, ist es eine gute Idee, den Header umzufärben und für die Zeilen alternierende Farben zu verwenden. Die Änderungen können gut am Beispiel der Task View nachvollzogen werden (**Abb. 10**), (Listing 5).

Darüber hinaus gibt es noch viele weitere Optionen, um den Tree umzugestalten. So kann man beispielsweise auch die Symbole für die Verzweigungen und die Linien verändern.

Bis auf wenige Ausnahmen lassen sich alle Eigenschaften der Widgets verändern. Hier allerdings sollte veranschaulicht werden, wie mit wenig Aufwand eine gut aussehende und moderne Anwendung gestaltet werden kann. Das Endprodukt zeigt **Abbildung 11**.

Stand und Ausblick

Obwohl die Implementierung von SWT/Qt schon weit fortgeschritten ist, sind manche Bereiche noch unvollständig. Die wichtigsten Widgets wurden umgesetzt. Da sich die Eclipse IDE mit SWT/Qt starten lässt und auch im Wesentlichen nutzbar ist, funktionieren auch andere Anwendungen grundsätzlich mit SWT/Qt. Die Implementierung einzelner Funktionen ist noch unvollständig, z. B. Drag and Drop. Andere Bereiche wie Printing und OLE sind noch nicht umgesetzt. Wer diese Funktionen jedoch nicht benötigt, kann SWT/Qt auf dem aktuellen Stand ausprobieren und einsetzen. Das Projekt wird unter dem Namen *swtqt* bei EclipseLabs gehostet und weitergeführt [8]. Für Anregungen und Fragen steht dort ein Forum zur Verfügung.

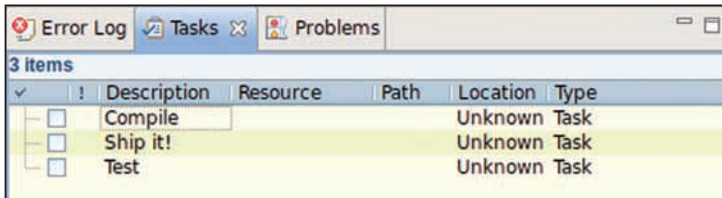


Abb. 10: Task View

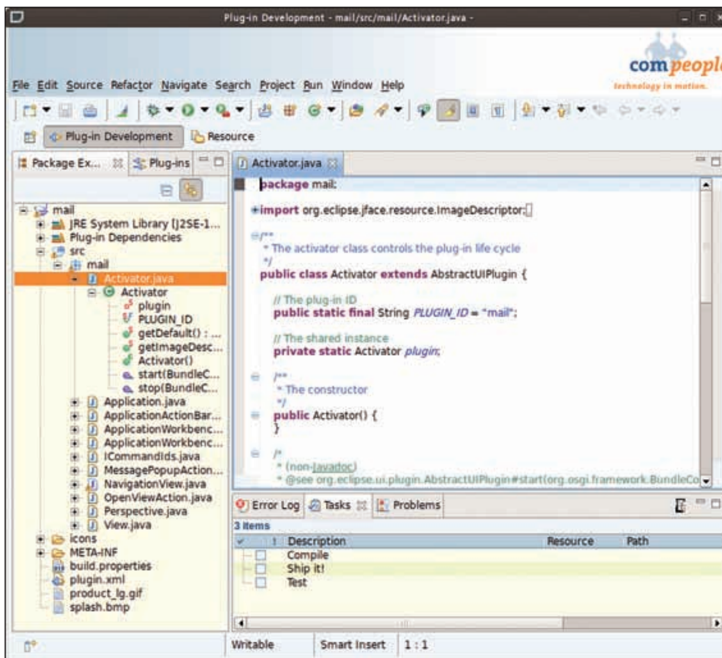


Abb. 11: Umgestaltete Eclipse IDE

Listing 5

```
QTreeWidget {
    ...
    alternate-background-color: #f5f5ce;
}
QHeaderView {
    border: 0px;
    height: 25px;
}
QHeaderView::section {
    background-color: #c4d3de;
    color: black;
    padding-left: 4px;
}
```

Jürgen Becker ist als Softwareentwickler für die compeople AG tätig. Er beschäftigt sich hauptsächlich mit der Entwicklung von UI-Frameworks und deren Anwendung. Des Weiteren interessiert er sich für die Architektur und Entwicklung von skalierbaren Webanwendungen.

Links & Literatur

[1] http://www.eclipse.org/nebula/

[2] http://www-10.lotus.com/ldd/lewiki.nsf/dx/dqx1cswidgets.htm

[3] http://wiki.eclipse.org/E4

[4] http://compeople.de/

[5] http://qt.nokia.com/

[6] http://www.w3.org/TR/CSS2/box.html

[7] http://doc.trolltech.com/4.5/stylesheet.html

[8] http://code.google.com/a/eclipselabs.org/p/swtqt/