

JFS 2005: Java SWT und JFace in der Praxis



Dr. Hartmut Kocher

Cortex Brainware
Consulting & Training GmbH
Kirchplatz 5
82049 Pullach im Isartal

Tel.: 089 - 7448500
<http://www.cortex-brainware.de>
hwk@cortex-brainware.de

Übersicht

Einführung

SWT

JFace

Rich Client Platform

Zusammenfassung

Einführung

Java bietet mehrere APIs zur Erstellung graphischer Oberflächen:

- AWT (Advanced Windowing Toolkit)
- Swing
- SWT (Standard Widget Toolkit) / JFace

Warum?

- Historie
- Unterschiedliche Zielsetzungen

AWT

Ansatz:

- GUI basierend auf den nativen Widgets der jeweiligen Plattform
- Es werden nur solche Widgets unterstützt, die auf allen Plattformen vorhanden sind.
- Für jedes Java-Widget gibt es eine Peer-Klasse für die jeweilige Plattform.

Vorteile:

- Schnell mit Look and Feel (L&F) der jeweiligen Plattform

Nachteile:

- Es werden nur wenige Widgets unterstützt (Kleinster gemeinsamer Nenner).
- Für moderne Anwendungen unzureichend

Swing

Ansatz:

- Plattformunabhängiges GUI, da Swing alle Widgets selbst zeichnet.
- Pluggable L&F, da Swing alles selbst macht
- Unterstützt MVC-Ansatz durch getrennte Widget- und Modellklassen.

Vorteile:

- Mächtiges GUI, das überall verfügbar ist.
- Viele Widgets, da kein „Kleinster-gemeinsamer Nenner“-Ansatz

Nachteile:

- Langsam, da alles selbst gezeichnet wird
 - Trägt viel zum (schlechten) Ruf der Java-Performance bei
 - Hat sich aber schon stark geändert
- Das L&F ist nie identisch zu den Plattform-Widgets

SWT

Ansatz:

- GUI basierend auf den nativen Widgets der jeweiligen Plattform
- Nicht verfügbare Widgets werden in Java nachgebaut
 - z.B. Trees in Motif
 - Best of Both Worlds

Vorteile:

- Sehr gute Performance und L&F der Zielplattform
- Unterstützt moderne Konzepte wie Drag and Drop.

Nachteile:

- Nur für einige Plattformen verfügbar
- Relativ „dünne“ Abstraktion der nativen Widgets
- Keine MVC-Unterstützung

JFace

Ansatz:

- Ergänzt SWT um MVC-Ansatz
 - Viewer mit eigenem Modell für ausgewählte Widgets
- Ergänzt SWT
 - Dinge, die in SWT vorhanden sind, werden genutzt

Vorteile:

- Man kann damit einfach komplexe Widgets wie Listen, Bäume etc. verwalten.
- Vereinfacht die Nutzung von SWT in realen Anwendungen.

Nachteile:

- Man benötigt zusätzliche Bibliotheken aus der Eclipse-Distribution
- SWT scheint an vielen Stellen durch
 - Man muss lernen, was man in SWT und was man in JFace realisiert

Rich Client Plattform

Ansatz:

- „Entkernte“ Eclipse-IDE für allgemeine Anwendungsentwicklung.
- Modulare Anwendungen durch Plugin-Konzept

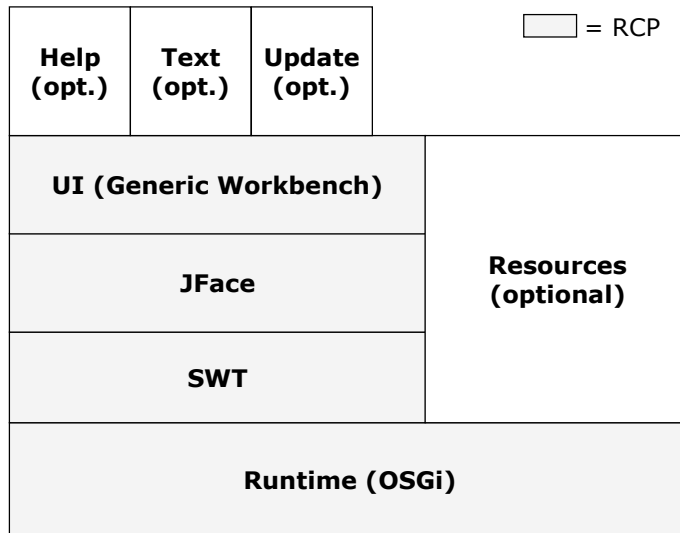
Vorteile:

- Moderne Anwendungsarchitektur für eigene Anwendungen verfügbar.
- Erlaubt modulare Anwendungen, die dynamisch erweitert werden können.
- Könnte Standard für Java-Client-Anwendungen werden

Nachteile:

- Mehrere Bibliotheken notwendig
- Steile Lernkurve, um Konzepte zu lernen.

Rich Client Platform



Speicherverbrauch

SWT

- **1.4 MB**

JFace

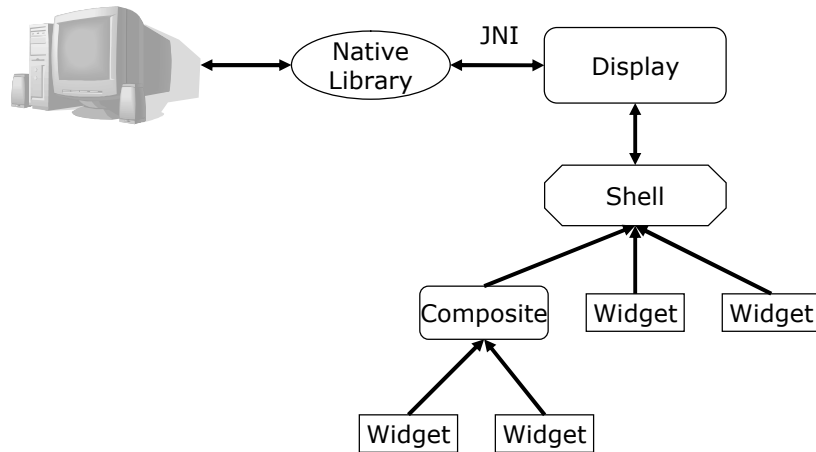
- **Runtime + SWT + JFace**
- **0,9 MB + 1,4 MB + 0,5 MB = 2,8 MB**

RCP

- **Runtime + SWT + JFace + Generische Workbench**
- **0,9 MB + 1,4 MB + 0,5 MB + 2,0 MB = 4,8 MB**

Daten basieren auf Eclipse 3.0

SWT Architektur



SWT

Für jede Plattform gibt es zwei Bibliotheken:

- swt.jar
 - Enthält SWT in der plattformspezifischen Ausprägung
- swt-win32-xxx.dll
 - Enthält den Interface-Code in einer plattformspezifischen Bibliothek

Beide Bibliotheken müssen zur Laufzeit verfügbar sein.

Hallo JFS

```
public static void main(String args[]) {
    Display display = Display.getDefault();
    Shell shell = new Shell(display);
    shell.setText("SWT Demo 1");
    shell.setSize(300, 175);
    Label hello = new Label(shell, SWT.CENTER);
    hello.setText("Hallo JFS!");
    hello.setBounds(shell.getClientArea());
    shell.open();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
            display.sleep();
    }
    display.dispose();
}
```



Grundlagen SWT

Beim Erzeugen eines Widgets muss immer das übergeordnete Widget angegeben werden.

- Es gibt keine add()-Methode.

```
Label hello = new Label(shell, SWT.CENTER);
```

Parent

Stil über Konstanten in
Klasse SWT

Eigenschaften werden über set-Methoden gesetzt.

```
hello.setText("Hallo JFS!");
```

Betriebssystem-Ereignisse werden einzeln bearbeitet.

- Die Nachrichtenschleife ist nicht versteckt.

```
while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
        display.sleep();
}
```

Grundlagen SWT

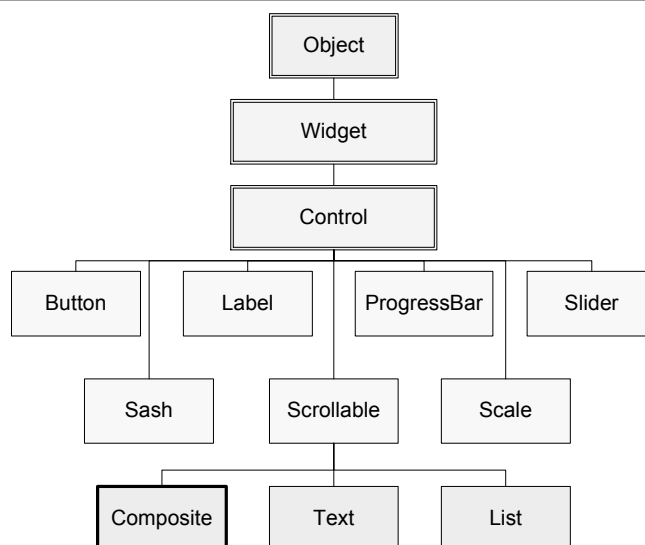
Alle Widgets sind möglichst dünn gekapselt.

- Daten werden im plattformspezifischen Widget gespeichert und nicht doppelt gehalten.
- Jedes Widget belegt Ressourcen auf dem Zielrechner.
 - Diese müssen verwaltet werden.

Alle benutzen Ressourcen müssen explizit verwaltet werden:

- Wer Ressourcen erzeugt, muss diese freigeben.
 - Widgets, Fonts, Colors, etc.
 - Aufruf von `dispose()`
- Übergeordnete Ressourcen geben Kinder rekursiv frei.
- Nach der Freigabe können Ressourcen nicht mehr benutzt werden.
 - z.B. ist das Label nicht mehr lesbar, da die native Datenstruktur gelöscht wurde (vgl. o.)

SWT Widgets



Advanced Widgets

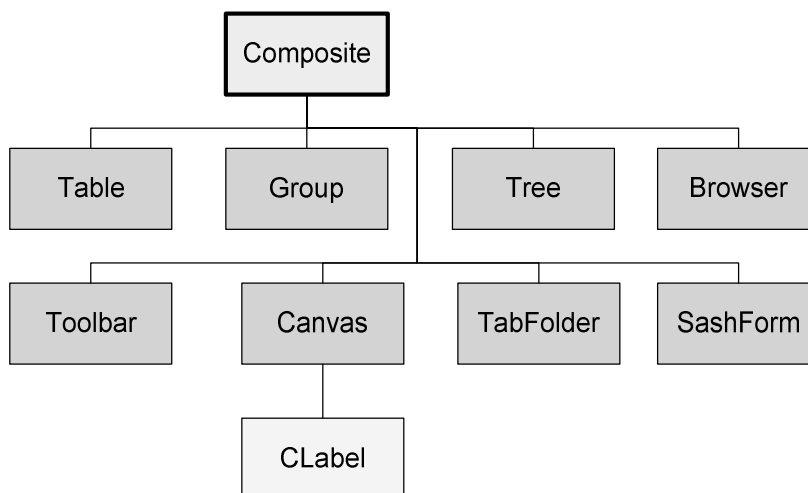
Alle komplexen Widgets sind von *Composite* abgeleitet.

- Dies ist die Basis für eigene Widgets oder für Widgets, die aus anderen Widgets zusammengefügt sind.
- Die Unterklasse *Canvas* dient als Basis für Widgets, die selbst zeichnen.
 - z.B. *CLabel*, mit dem man Farbverläufe, Icons und Text kombinieren kann.
 - Eigene Controls, die Graphik benötigen, sollten ebenfalls von *Canvas* erben.
 - Gezeichnet wird über ein *PaintListener*-Ereignis, das einen Graphik-Kontext enthält.

Eigene Controls kann man sich als spezielle Composites selbst bauen.

- z.B. eine Outlook-Listbar

Advanced Widgets (Ausschnitt)

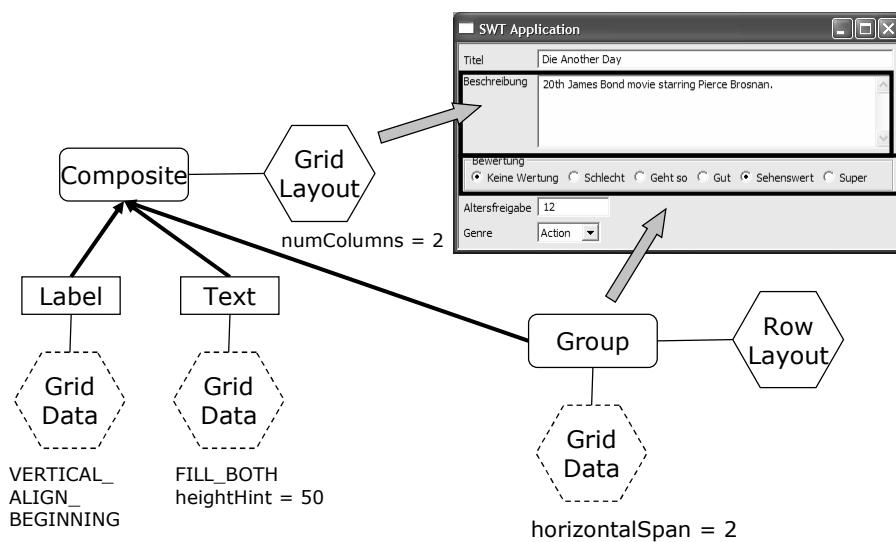


Layouts

Man kann Controls innerhalb von Composites auf verschiedene Art platzieren:

- Absolut, indem man kein Layout spezifiziert.
 - Dann muss man Größe und Position jedes Controls mit `setBounds()` explizit angeben.
- Mit Hilfe eines Layouts kann man die Controls nach bestimmten Regeln platzieren lassen.
 - Es gibt fertige Layouts wie `GridLayout`, `FormLayout` oder `StackLayout`.
 - Man kann leicht eigene Layouts definieren.
 - Jedes `Composite` kann ein eigenes `Layout` besitzen.
 - Für jedes Layout kann es zusätzliche Layoutdaten in jedem `Control` geben.

Beispiel: GridLayout



Event Handling

Ereignisse werden auf zwei Arten behandelt:

- Untypisierte Ereignisse und typisierte Ereignisse

Untypisierte Ereignisse

- Interface *Listener* mit einer Methode *void handleEvent(Event e)*

- Ereignis wird als Typ übergeben

- *void addListener(int eventType, Listener listener);*
- Ereignistypen sind in der Klasse SWT definiert.
- Man kann den gleichen *Listener* für verschiedene Ereignisse registrieren und innerhalb einer Abfrage bearbeiten:

```
void handleEvent(Event e) {
    switch (e.type) {
        SWT.Selection:
            System.out.println("Button pressed");
            break;
    }
}
```

Event Handling

Typisierte Ereignisse

- Spezielle Schnittstellen für jeden Ereignistyp
- Jedes Ereignis benötigt speziellen Handler
- Besseres Design, da die Aufgabe des Handlers sofort klar ist
- Intern werden typisierte Ereignisse über untypisierte Ereignisse realisiert.

Beispiel: Button-Ereignis

- Schnittstelle *SelectionListener* mit Methode *void widgetSelected(SelectionEvent e)*
- Button mit Methode *void addSelectionListener(SelectionListener listener)*

Im Normalfall sollten typisierte Ereignisse verwendet werden.

Menüs

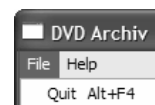
Menüs sind aus wenigen Klassen aufgebaut:

- Menu
 - Menüleiste
 - Untermenü
 - Popup-Menü
- MenuItem
 - Normale Auswahl
 - Checkbox
 - Radio-Button
 - Seperator
- Die Untertypen werden über die Stil-Konstante im Konstruktor übergeben:
 - z.B. `new Menu(shell, SWT.BAR)` erzeugt eine Menüleiste im Fenster

Menüs

Menüaufbau durch direktes Erzeugen der hierarchischen Menüstruktur

- Icons, Shortcuts etc. können gesetzt werden
- Auswahl über *SelectionListener*



```
final Menu menuBar = new Menu(shell, SWT.BAR);
shell.setMenuBar(menuBar);

final MenuItem fileMenuBar = new MenuItem(menuBar, SWT.CASCADE);
fileMenuBar.setText("File");

final Menu fileMenu = new Menu(fileMenuBar);
fileMenuBar.setMenu(fileMenu);

final MenuItem quitMenuItem = new MenuItem(fileMenu, SWT.NONE);
quitMenuItem.setText("Quit\tAlt+F4");
quitMenuItem.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
    }
});
```

Popup-Menüs

Popup-Menüs können sehr leicht mit jedem *Control* assoziiert werden.

- Spezieller Konstruktor der Menu-Klasse.
- Menü muss dem *Control* zugeordnet werden.

Beispiel:

```
Menu m = new Menu(control);  
MenuItem i = new MenuItem(m, SWT.NONE);  
i.setText("New");  
  
control.setMenu(m);
```

Dialoge

Es gibt plattformspezifische Dialoge für Standardfälle:

- Message Box
- Farbauswahl
- Datei- oder Ordnerauswahl
- Drucken
- Font-Auswahl

Die Anwendung ist sehr einfach:

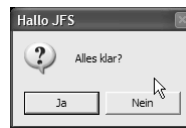
```
<DialogType> dlg = new <DialogType>(shell);  
dlg.setSomeData(value);  
<ReturnType> result = dlg.open();
```

- Über Konstanten werden Modalität, Buttons, Icons etc. ausgewählt.

Standarddialoge

Ausgabe einer Nachricht in einer *MessageBox*:

```
MessageBox mbox = new MessageBox(shell,  
    SWT.ICON_QUESTION | SWT.YES | SWT.NO);  
mbox.setMessage("Alles klar?");  
mbox.setText("Hallo JFS");  
if(mbox.open() == SWT.YES) {  
  
}  
}
```



Eigene Dialoge

Eigene Dialoge können von der Oberklasse *Dialog* abgeleitet werden.

- Eigene *open()*-Methode mit geeignetem Rückgabewert definieren.
- Set- und Get-Methoden für alle Parameter.
- Die *open()*-Methode muss alle Widgets definieren, Ereignisse bearbeiten und die Nachrichtenschleife implementieren.
- Als Ergebnis wird je nach gedrückter Taste *null* oder der gewünschte Rückgabewert zurückgegeben.

Die Oberklasse bietet wenig Hilfestellung.

- Man muss fast alles selbst machen.
- JFace bietet deutlich höherwertige Abstraktionen.

Weitere Eigenschaften

SWT unterstützt eine Reihe weiterer Eigenschaften:

- Zeichnen auf der Canvas über PaintListener.
- Drag & Drop
- Einbindung von OLE und ActiveX-Komponenten unter Windows.
 - Damit kann man z.B. Word in SWT-Anwendungen einbetten!!!
- Einbetten von Swing-Widgets
 - Damit lassen sich vorhandene Swing-Anwendungen mit SWT kombinieren.
 - Swing-Controls können genutzt werden.

JFace

JFace bietet eine Abstraktionsebene oberhalb von SWT.

- Dabei wird SWT nicht verdeckt, sondern nur um höherwertige Abstraktionen ergänzt.
- Details können weiter in SWT behandelt werden.
- JFace versteckt SWT nicht.

JFace benötigt eigene Bibliotheken aus Eclipse oder der RCP.

- jface.jar Basisbibliothek
- runtime.jar Laufzeitumgebung
- osgi.jar basierend auf dem OSGI-Framework
- jfacetext.jar Spezielle Textviewer-Klassen

JFace Überblick

JFace bietet Abstraktionen in folgenden Bereichen:

- Viewer mit MVC-Ansatz für komplexe Controls
 - TreeViewer, ListView, TableView
- Editoren für Zellen in Tabellen und Bäumen
- Action-Framework zur Verwaltung von Benutzeraktionen
 - Allgemeine Behandlung für Menüs und Toolbars
- Dialoge als Basis für eigene Dialoge
 - Ergänzend zu den SWT-Standarddialogen
- Framework für Benutzereinstellungen (Preferences)
- Textframework mit Dokumentenmodell
 - Damit lassen sich z.B. Texteditoren mit Syntaxhervorhebungen etc. realisieren

Hallo JFS mit JFace

```
public class HelloJFSJF extends ApplicationWindow {
    public HelloJFSJF() {
        super(null);
    }

    protected Control createContents(Composite parent) {
        Composite container = new Composite(parent, SWT.NONE);
        container.setLayout(new FillLayout());

        Label label = new Label(container, SWT.CENTER);
        label.setText("Hallo JFS!");
        return container;
    }

    public static void main(String args[]) {
        HelloJFSJF window = new HelloJFSJF();
        window.setBlockOnOpen(true);
        window.open();
        Display.getCurrent().dispose();
    }
}
```



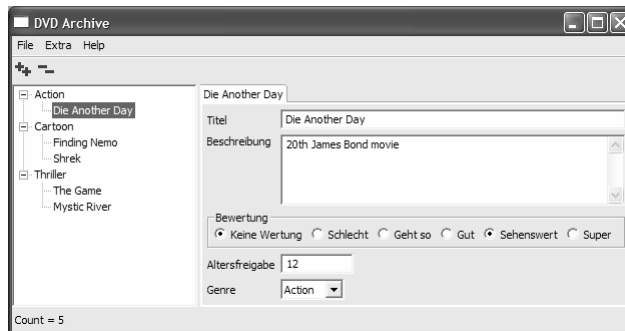
Fenster mit Inhalt füllen

Nachrichtenschleife eingebaut

Standardfenster

Ein Standardfenster für JFace unterstützt mehrere Bereiche, die durch Überschreiben von Methoden konfiguriert werden:

- Menüleiste
- Inhaltsbereich
- Werkzeugleiste
- Statusleiste



JFace Viewers

Viewer bieten eine MVC-Abstraktion oberhalb der SWT-Controls.

- *ContentProvider* liefern Zugriff auf das darunter liegende Modell.
- *LabelProvider* extrahieren Icons und Text für die Darstellung.
- Dadurch entsteht eine Entkopplung von Modell und Präsentation.

Zusätzliche Funktionen des Viewers unterstützen die Manipulation von Daten

- Filtern
- Sortieren

Beispiel TreeViewer

Ein ContentProvider für einen Baum muss die folgenden Funktionen bereits stellen:

```
public Object[] getElements(Object inputElement);
public Object[] getChildren(Object parentElement);
public Object getParent(Object);
public boolean hasChildren(Object element);
public void inputChanged(Viewer viewer, Object oldInput,
                        Object newInput);

public void dispose();
```

- *getElements* liefert die Wurzelemente des Baums
- *getChildren* liefert die Kinder des jeweiligen Elements
- *inputChanged* ist hilfreich, wenn man Listener für das Modell registriert hat
- Die Wurzel des Baums wird mit *setInput* beim TreeViewer gesetzt.

Beispiel TreeViewer

Der LabelProvider muss aus den Elementen den Text und ein optionales Icon extrahieren:

```
public Image getImage(Object element);
public String getText(Object element);
public boolean isLabelProperty(Object element, String property);
```

- *isLabelProperty* dient der Optimierung, um Elemente nur dann neu zu zeichnen, wenn eine Änderung auch Auswirkung auf die Anzeige hat.

Listener und andere Funktionen (z.B. *update*, *refresh*) unterstützen das Aktualisieren von Viewern, ohne alles neu aufbauen zu müssen.

Actions

Befehle werden in Aktionen verpackt (Command Pattern).

- Aktionen können gleichzeitig in Menüs und in Toolbars verwendet werden.
 - Das *Action*-Interface bzw. die Standardimplementierung *Action* hat eine *run()*-Methode, die in einer Unterklasse implementiert werden muss.
 - Zusätzlich kann man den Namen und Icons angeben, die in Menüs bzw. Toolbars verwendet werden sollen.
 - Aufbau von Menüs und Toolbars vereinfacht sich dadurch drastisch.
- Zusätzliche Funktionalität unterstützt das Einfügen von Befehlen in vorhandene Menüs und vereinfacht das gesamte Management von Benutzeraktionen.

Weitere JFace Bestandteile

Dialoge

- Es gibt Standarddialoge für Fehlermeldungen und zur Eingabe von Parametern.
- Eigene Dialoge können aus Vorlagen einfach gebaut werden.

Benutzereinstellungen

- Es gibt ein Preferences-Framework, was es erlaubt, Benutzereinstellungen hierarchisch in einzelnen Seiten anzuzeigen.
- Es gibt vorgefertigte „leere“ Seiten, die nur mit Inhalt gefüllt werden müssen.

Wizards

- Um komplexe Einstellungen in mehreren Schritten vorzunehmen, gibt es ein Wizard-Framework, mit dem man schnell eigene Wizards erstellen kann.

Rich Client Platform

Die Eclipse Rich Client Platform erlaubt es, modulare Anwendungen in Java zu entwickeln.

- Neben SWT und JFace umfasst die RCP die gesamte Infrastruktur, die benötigt wird, um Anwendungen aus Plugins aufzubauen.
- RCP ist eine leere Workbench ohne IDE-Funktionalität.
- Es gibt Mechanismen, um Plugins zu laden und diese auszuführen.
- Plugins können flexibel sein und können aufeinander aufbauen.
 - Jedes Plugin kann Erweiterungspunkte definieren, die andere Plugins benutzen können.
- Plugins werden in einem XML-Manifest definiert und nur bei Benutzung geladen.
- Für aufwendige Anwendungen wird ein Framework vorgegeben, welches das Erstellen von Anwendungen vereinfacht.

Windows-Anwendungen

Ein wichtiger Bereich dürften native Windows-Anwendungen darstellen.

- Der Einsatz von SWT und Java erlaubt höhere Produktivität im Vergleich zu herkömmlichen Windows/C++-Lösungen.
- Viele Anwender wollen dennoch keine Java-Laufzeitumgebung auf ihrem Rechner.
- Es gibt Lösungen, die es erlauben, SWT-Anwendungen in native exe-Anwendungen umzubauen, die keine JRE benötigen.
 - z.B. kann „JET“ SWT-Anwendungen übersetzen
 - <http://www.excelsior-usa.com/>
- Probleme gibt es, wenn wie bei RCP Klassen zur Laufzeit geladen werden (Reflection).
 - Dann benötigt man einen JIT-Compiler bzw. eine JRE.

Tools

Es gibt mehrere Werkzeuge, mit denen man SWT/JFace-Anwendungen erstellen kann.

- Eclipse Visual Editor Project
 - Open-Source GUI-Builder für Swing und SWT
 - <http://www.eclipse.org/vep/>
- Kommerzielle GUI-Builder für Swing/SWT
 - SWT Designer / WindowBuilder (<http://www.instantiations.com/> bzw. <http://www.swt-designer.com/>)
 - Jigloo (<http://www.cloudgarden.com/jigloo/index.html>)
- Graphik-Bibliotheken
 - Open Source / Kommerzielle Widgets verfügbar (z.B. Kalender) (<http://www.eclipseplugincentral.com/>)
 - Graphikbibliothek JGo unterstützt zukünftig SWT (<http://www.nwoods.com/go/jgo.htm>)

Zusammenfassung

SWT vereinigt die Vorteile von AWT und Swing.

- Native Widgets für originäres Look & Feel
- Hohe Geschwindigkeit mit wenig Overhead
- Komplexe Widgets für moderne GUIs
- Für alle wichtigen Plattformen verfügbar.

JFace bietet zusätzliche Abstraktionen, die auf SWT aufsetzen.

- Viewer mit MVC-Ansatz
- Command Pattern für Befehlsausführung

Rich Client Platform fügt Flexibilität durch Plugin-Konzepte hinzu.

Insgesamt eignen sich diese Technologien, um Anwendungen zu bauen, die auf den ersten Blick nicht von nativen Anwendungen zu unterscheiden sind.

Ressourcen

Bücher:

- Rob Warner: „The Definitive Guide to SWT and JFace“, APress, 2004.
- M. Scarpino et al.: „SWT/JFace in Action“, Manning, 2004 (auch als EBook).

Internet:

- <http://www.eclipse.org/platform/index.html> (unter SWT)

Artikel im Internet:

- Übersicht
 - <http://www.informit.com/guides/content.asp?q=java&SeqNum=82>
- Layouts
 - <http://www.informit.com/guides/content.asp?q=java&seqNum=84>