

Benutzungsoberflächen

Prof. Dr.-Ing. Holger Vogelsang
holger.vogelsang@hs-karlsruhe.de



Inhaltsverzeichnis

- Änderungen zum Wintersemester 2015/2016 (10)
- Literatur (11)
- Lernziele (13)
- Roter Faden (14)
- Klassifikation von Anwendungen (15)
 - ◆ Nach Architektur (15)
 - ◆ Nach Zielplattform (19)
 - ◆ Vergleich (20)
- Architekturen von Fat-Client-Anwendungen (21)
 - ◆ Grundlagen (21)
- Die Eclipse Rich Client Platform (33)
 - ◆ Vorbereitung und Installation (39)
 - ◆ Vorbereitung und Installation (RCP-SDK) (40)
 - ◆ Vorbereitung (RCP-SDK) (41)
 - ◆ Aufbau einer RCP-Anwendung (45)



Inhaltsverzeichnis

- OSGi (54)
 - ◆ Problemstellung (55)
 - ◆ Einführung (58)
 - ◆ Schichten (62)
 - ◆ Lebenszyklus (65)
 - ◆ Verbindung von Diensten (66)
 - ◆ Fazit (96)
- Die Eclipse Rich Client Platform (97)
 - ◆ Plug-ins und Features (97)
- Modell der Oberfläche (112)
 - ◆ Einführung (112)
- Eclipse Modeling Framework (EMF) (129)
 - ◆ Einführung (129)
- Erweiterbarkeit (138)
 - ◆ Einführung (138)



Inhaltsverzeichnis

- ◆ [Erweiterungen und Erweiterungspunkte \(140\)](#)
- [Oberflächenlayout \(155\)](#)
 - ◆ [Übersicht \(155\)](#)
 - ◆ [Eine erste SWT-Anwendung \(157\)](#)
 - ◆ [Layoutmanagement \(161\)](#)
 - ◆ [FillLayout \(165\)](#)
 - ◆ [RowLayout \(168\)](#)
 - ◆ [GridLayout \(174\)](#)
 - ◆ [Verschachtelte Layouts \(182\)](#)
 - ◆ [FormLayout \(185\)](#)
 - ◆ [Klassenübersicht \(200\)](#)
 - ◆ [Interaktive Werkzeuge \(201\)](#)
- [Widgets \(203\)](#)
 - ◆ [Übersicht \(203\)](#)
 - ◆ [Spezielle \(212\)](#)



Inhaltsverzeichnis

- ◆ [Nebula-Projekt \(219\)](#)
- ◆ [RCPToolbox \(221\)](#)
- ◆ [Speicherverwaltung \(222\)](#)
- ◆ [Klassenhierarchie \(226\)](#)
- [Grundlagen der Ereignisbehandlung \(233\)](#)
 - ◆ [Beobachter-Muster \(233\)](#)
- [Model-View-Controller \(243\)](#)
 - ◆ [Einführung \(243\)](#)
 - ◆ [Beispiel JFace-Tabelle \(247\)](#)
- [Ereignisbehandlung durch Befehle \(276\)](#)
 - ◆ [Einführung \(276\)](#)
 - ◆ [Commands \(282\)](#)
 - ◆ [Handler \(285\)](#)
 - ◆ [Tastaturbindungen \(290\)](#)
 - ◆ [Menüs und Toolbars \(292\)](#)



Inhaltsverzeichnis

- Dependency Injection (297)
 - ◆ Einführung (298)
 - ◆ Annotationen (303)
 - ◆ Kontext und Gültigkeitsbereich (306)
 - ◆ Annotationen zur Steuerung des Verhaltens (311)
 - ◆ Anmerkungen und Tipps (320)
- Dynamische Änderung des Anwendungsmodells (323)
 - ◆ Einführung (323)
 - ◆ API (325)
- Ereignisbehandlung durch Befehle (335)
 - ◆ Bedingte Handler und Menüeinträge (335)
 - ◆ Beiträge durch Plug-ins (345)
- Dialoge (355)
 - ◆ Einführung (355)
 - ◆ MessageBox (359)



Inhaltsverzeichnis

- ◆ [TitleAreaDialog \(360\)](#)
- [Internationalisierung \(365\)](#)
 - ◆ [Einführung \(365\)](#)
 - ◆ [Ressource-Dateien \(369\)](#)
 - ◆ [Ressource-Dateien – Variante 1 \(373\)](#)
 - ◆ [Ressource-Dateien – Variante 2 \(374\)](#)
 - ◆ [Ressource-Dateien – Variante 2 \(RCP-spezifisch\) \(382\)](#)
 - ◆ [Ressource-Dateien – Variante 3 \(386\)](#)
 - ◆ [Formatumwandlungen \(388\)](#)
 - ◆ [Stringbehandlung \(393\)](#)
- [Multithreading \(395\)](#)
 - ◆ [Einführung \(395\)](#)
 - ◆ [Befehlsmuster \(400\)](#)
 - ◆ [Job \(413\)](#)



Inhaltsverzeichnis

- Anbindung an die Geschäftslogik (424)
 - ◆ Übersicht (424)
 - ◆ Ein- und Ausgabe (428)
 - ◆ Ein- und Ausgabe mit Databinding (433)
 - ◆ Dialogsteuerung (468)
 - ◆ Dialogsteuerung durch einen Zustandsautomaten (470)
 - ◆ Dialogsteuerung durch einen Ereignis-Vermittler (472)
 - ◆ Dialogsteuerung durch Nachrichten auf einem Bus (476)
 - ◆ Dialogsteuerung durch externe Skripte (485)
- Deklarative Beschreibungen (491)
 - ◆ Einführung (491)
 - ◆ XWT (501)
 - ◆ Fazit (511)
- Styling und Export (512)
 - ◆ Splash-Screen und Styling mit CSS (512)



Inhaltsverzeichnis

- ◆ [Export als ein lauffähiges Programm \(525\)](#)
- [Lose Enden \(530\)](#)
 - ◆ [Offene Punkte \(530\)](#)
 - ◆ [RAP \(532\)](#)
- [Ende \(533\)](#)

Änderungen zum Wintersemester 2015/2016



- keine



Literatur

Grundlagen und allgemeine Dokumentationen

- M. Marinilli, „Professional Java User Interfaces“, Wiley & Sons, 2006
- Java-Quelltextbeispiele (alles Mögliche): <http://www.java2s.com/>

SWT/JFace

- R. Warner, R. Harris, „The Definite Guide to SWT and JFace“, Apress, 2007
- M. Scarpino et.al., „SWT/JFace in Action“, Manning Publications Co., 2004
- <http://www.eclipse.org/swt/>
- <http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html>
- <http://www.eclipse.org/swt/snippets/>
- <http://wiki.eclipse.org/index.php/JFaceSnippets>



Literatur

Eclipse Rich Client Platform

- Bernhard Steppan, „Eclipse Rich Clients und Plug-ins“, Carl Hanser Verlag, Juli 2015
- Marc Teufel, „Eclipse 4“, entwickler.press, Oktober 2012
- Lars Vogel, „Eclipse 4 Application Development“, Mai 2015 (Sammlung von Tutorials aus <http://www.vogella.de/eclipse.html>)

OSGi

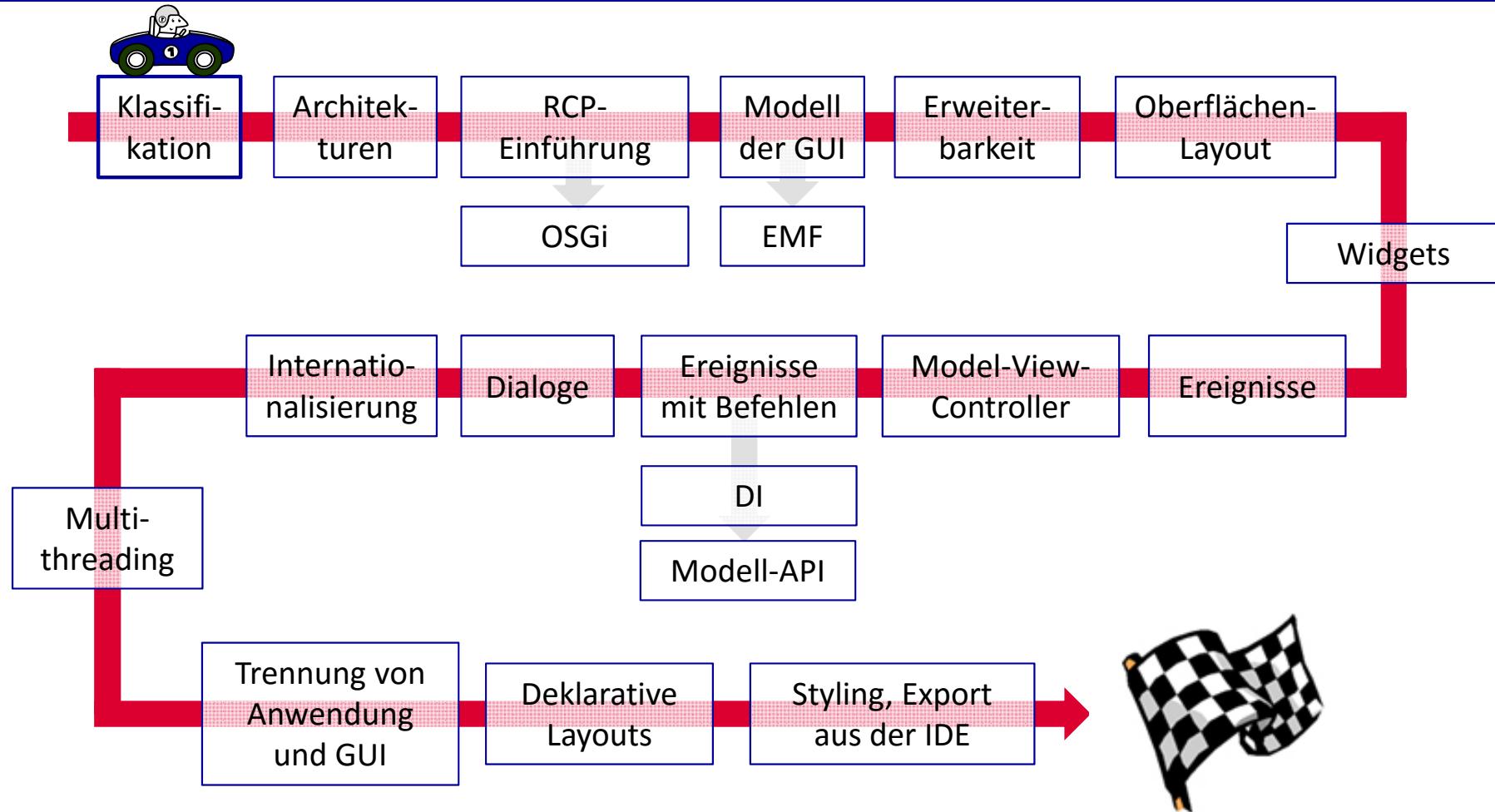
- G. Wütherich, N. Hartmann, B. Kolb, M. Lübken, „Die OSGi Service Platform“, dpunkt-Verlag, 2008



Lernziele

- Arbeit mit einem Framework
- Modularisierung mit „Dependency Injection“ und OSGi
- Struktur grafischer Oberflächen
- Ereignisbehandlung
- Trennung der Anwendungslogik von der Oberfläche
- Architekturen
- Erweiterbare Programme
- Multithreading
- Spaß

Roter Faden



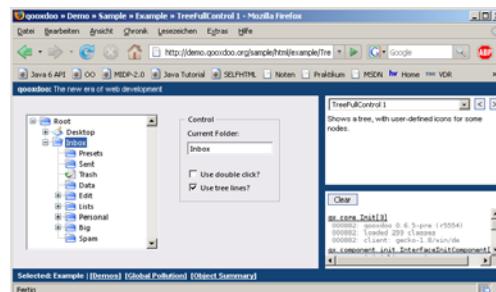
Klassifikation von Anwendungen



Nach Architektur

Die folgenden Begriffe sind in der Literatur sehr uneinheitlich definiert!

- Thin Client:
 - ◆ Beispielsweise Ajax-Anwendung im Browser
 - ◆ Start über URL, keine lokale Installation
 - ◆ Geschäftslogik überwiegend auf dem Server
 - ◆ Request/Response-Kommunikation



Browser
(HTML, JavaScript)



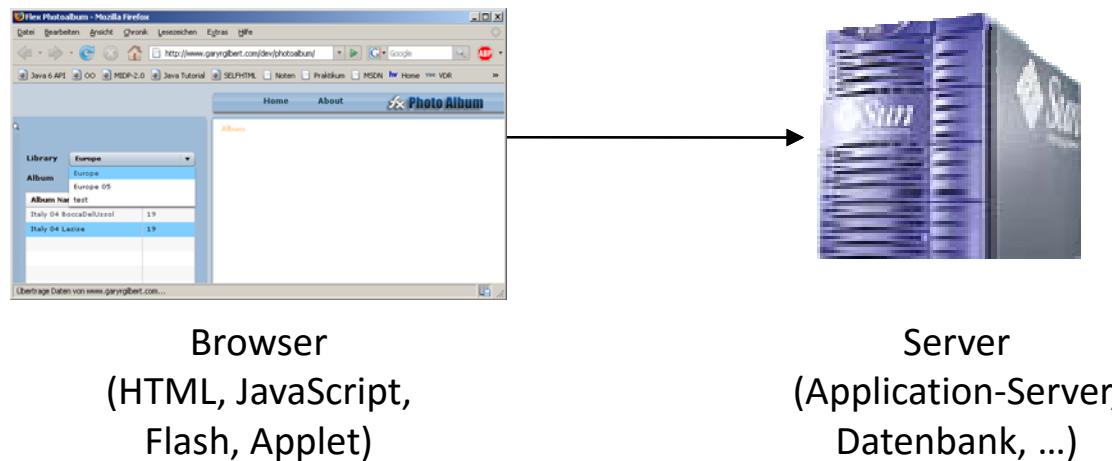
Server
(Application-Server,
Datenbank, ...)

Klassifikation von Anwendungen



Nach Architektur

- Rich Thin Client:
 - ◆ Anwendung läuft mit Hilfe eines Plug-ins im Browser (z.B. Flash, Silverlight, ...)
 - ◆ Start über URL, keine lokale Installation
 - ◆ Geschäftslogik überwiegend auf dem Server
 - ◆ Request/Response-Kommunikation

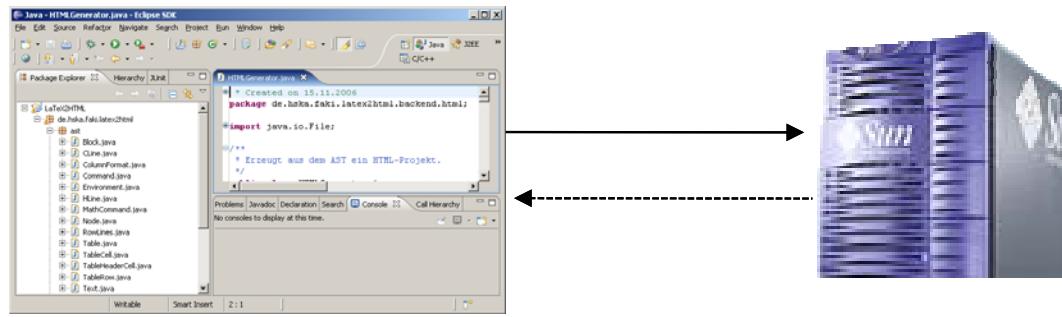


Klassifikation von Anwendungen



Nach Architektur

- Rich (Fat) Client:
 - ◆ Clientseitiges Framework (z.B. RCP)
 - ◆ Start auf dem Client, lokale Installation, benötigte Module werden vom Server nachgeladen
 - ◆ Geschäftslogik auf Client und Server
 - ◆ Kommunikation Request/Response oder bidirektional möglich



Client
(Eclipse RCP, ...)

Server
(Application-Server,
Datenbank, ...)



- Smart Client (häufig im Microsoft-Umfeld):
 - ◆ .NET, ASP .NET, XAML, ...
 - ◆ Start über URL, keine lokale Installation
 - ◆ Geschäftslogik auf Client und Server
- Fat Client (plattformunabhängig):
 - ◆ Beispielsweise in Java mit Swing, SWT/JFace, ... oder C++ mit QT
 - ◆ Start auf dem Client, lokale Installation
 - ◆ Geschäftslogik auf überwiegend auf dem Client, aber auch Server möglich
- Fat Client (plattformabhängig):
 - ◆ Beispielsweise in C++ für Windows, Linux
 - ◆ Start auf dem Client, lokale Installation
 - ◆ Geschäftslogik auf überwiegend auf dem Client, aber auch Server möglich

Klassifikation von Anwendungen



Nach Zielplattform

- Klassischer Desktop-PC → siehe vorherige Seiten
- Mobile Geräte:
 - ◆ Unterschiedliche Betriebssysteme
 - Android
 - iOS
 - Windows Phone
 - ...
 - ◆ Unterschiedliche Leistungen
 - Mobile Telefone
 - Smartphones , Tablet-PCs
 - ◆ Unterschiedliche Programmierumgebungen
 - Java/Java ME
 - C++/Objective-C/Swift
 - HTML/CSS/...
 - ...

Klassifikation von Anwendungen

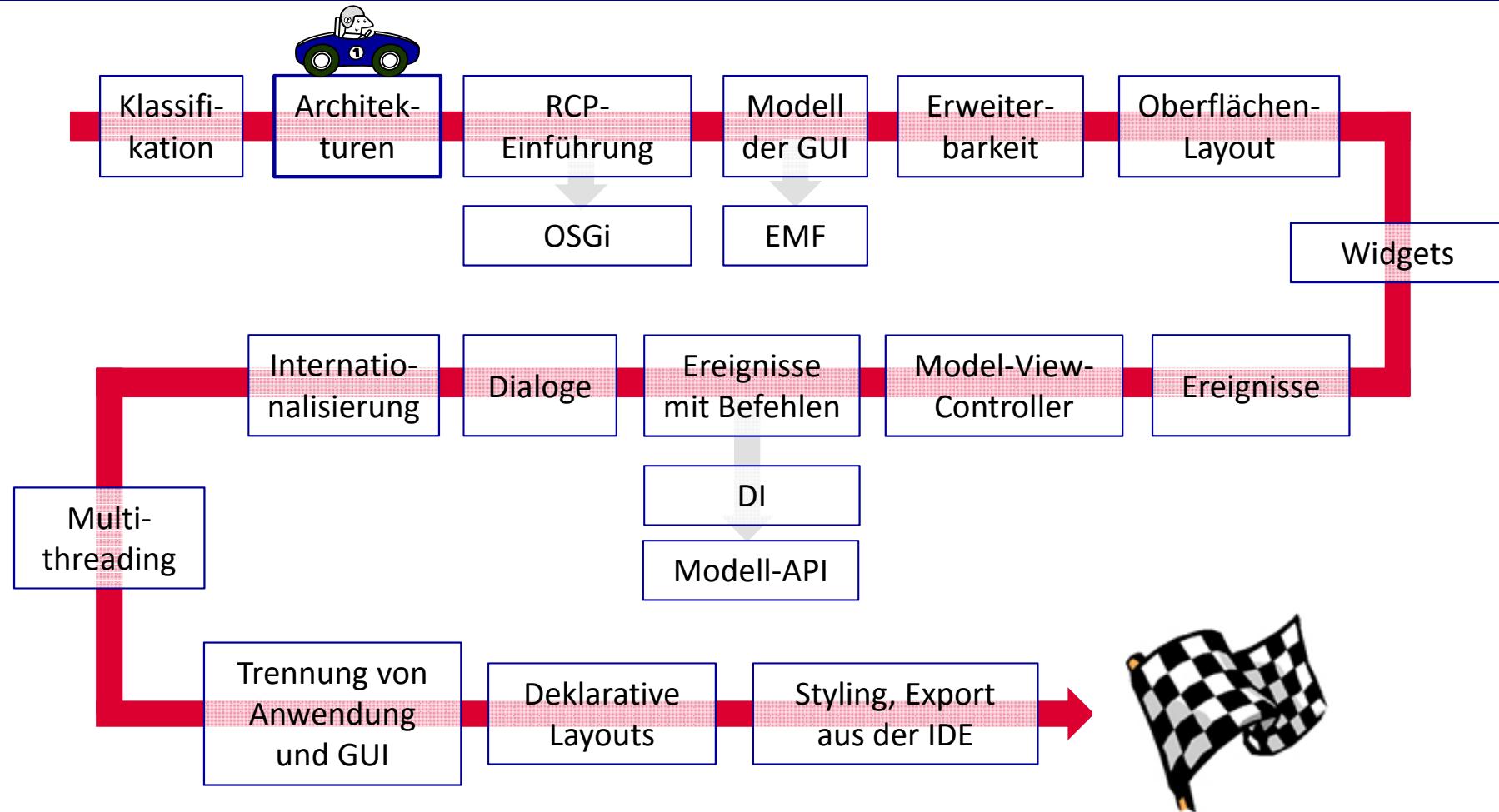


Vergleich

	Thin Client (Ajax)	Rich Thin Client (Browser + Plug-in)	Rich Fat Client (Java)	Microsoft Smart Client	Java Fat Client	Windows, Mac OS X oder Linux Fat Client
Installation	✓✓	✓✓ / ↘	✓✓ / ↘	↖	✓✓ / ↘	↖↖
Administration	✓✓	✓✓ / ↘	✓✓ / ↘	↖	✓✓ / ↘	↖
Wartung (SW)	✓	✓✓	✓✓	✓✓	✓✓	✓✓
Interaktionen	✓	✓✓	✓✓	✓✓	✓✓	✓✓
Performance	✓	✓	✓✓	✓✓	✓✓	✓✓
GUI-Konsistenz	✓	↖	✓✓	✓✓	✓✓	✓✓
Plattformunabh.	✓✓	✓	✓✓	↖↖	✓✓	↖↖
Arbeit bei Serverausfall	↖↖	↖↖	✓✓/↖↖	↖↖	✓✓/↖↖	✓✓/↖↖
Applikationslogik	Server	Server	Client und Server	Client und Server	überwiegend Client	überwiegend Client

Architekturen von Fat-Client-Anwendungen

Grundlagen





- Eigenschaften und Vorteile eines (Rich-)Fat-Clients mit Java:
 - ◆ Läuft als Desktop-Anwendung auf dem Computer des Anwenders.
 - ◆ Besitzt eine mächtige Benutzeroberfläche (Drag/Drop, ...).
 - ◆ Anwendungslogik läuft häufig lokal auf dem Client ab → funktioniert u.U. auch, wenn der Server nicht erreichbar ist.
 - ◆ Server: Datenbank, aber auch z.B. Application-Server möglich.
 - ◆ Plattformunabhängigkeit wegen Java.
 - ◆ Gute Softwareentwicklung und –wartung wegen sehr guter Entwicklungsumgebungen und Modellierungswerkzeuge.
 - ◆ GUI: Swing, SWT/JFace, AWT, JavaFX, QTJambi, ...
- Nachteile:
 - ◆ Benötigt ein (aktuelles) JRE auf dem Client → lokale Installation.
 - ◆ Client-Software selbst muss auch lokal installiert werden. Ausweg: Java Webstart.
 - ◆ U.U Geschwindigkeit



AWT (Abstract Window Toolkit)

- Ab JDK 1.x vorhanden
- Nachteile:
 - ◆ Alle Fenster- und Dialogelemente werden von dem darunterliegenden Betriebssystem zur Verfügung gestellt → schwierig, plattformübergreifend ein einheitliches Look&Feel zu realisieren.
 - ◆ Die Eigenarten jedes einzelnen Fenstersystems waren für den Anwender unmittelbar zu spüren.
 - ◆ Im AWT gibt es nur eine Grundmenge an Dialogelementen, mit denen sich aufwändige grafische Benutzeroberflächen nicht oder nur mit sehr viel Zusatzaufwand realisieren ließen.



Swing

- Eigenschaften:
 - ◆ Swing-Komponenten benutzen nur Top-Level-Fenster sowie grafische Zeichenoperationen des Betriebssystems.
 - ◆ Alle anderen GUI-Elemente werden von Swing selbst gezeichnet.
- Vorteile:
 - ◆ Plattformspezifische Besonderheiten fallen weg → einfachere Implementierung der Dialogelemente
 - ◆ einheitliche Bedienung auf unterschiedlichen Betriebssystemen
 - ◆ nicht nur Schnittmenge der Komponenten aller Plattformen verfügbar
 - ◆ Pluggable Look&-Feel: Umschaltung des Aussehens einer Anwendung zur Laufzeit (Windows, Motif, Metal, ...).
 - ◆ sehr schöne interne Struktur und saubere API



- Nachteile:
 - ◆ Swing-Anwendungen sind ressourcenintensiv. Das Zeichnen der Komponenten erfordert viel CPU-Leistung und Hauptspeicher (Unterstützung durch DirectDraw, OpenGL) → fühlt sich teilweise etwas träge an.
 - ◆ schlechte Anpassung an Besonderheiten eines Fenstersystems (z.B. Übernahme von Themes, ...)
 - ◆ abnehmende Bedeutung
- Unterstützte Plattformen:
 - ◆ alle Java-SE-Plattformen
- Wird in Zukunft durch JavaFX ersetzt werden.



JavaFX

- Eigenschaften:
 - ◆ Wurde ursprünglich mit eigener Skript-Sprache für Animationen entwickelt.
 - ◆ seit Version 2 mit Java-API, die teilweise an Swing angelehnt ist
- Vorteile:
 - ◆ Hardware-unterstützte Zeichenoperationen
 - ◆ arbeitet einheitlich auf einem Szenen-Graphen
 - ◆ Styling mit CSS
 - ◆ sehr gute Animations- und Effektmöglichkeiten
 - ◆ 3D-Unterstützung
- Nachteile:
 - ◆ zur Zeit keine große Verbreitung
 - ◆ teilweise noch nicht ganz so mächtig wie Swing



SWT/JFace

- Eigenschaften:
 - ◆ Wurde als Basis für die Entwicklung von Eclipse geschaffen.
 - ◆ AWT-Ansatz bei SWT: native Dialogelemente des Fenstersystems, wenn vorhanden (ansonsten: nachgebaut).
 - ◆ JFace bildet eine Abstraktionsschicht von SWT
 - Viewer: Trennung von Darstellung und den Daten selbst
 - Vereinfachung der Ereignisbehandlung
 - Registry für Zeichensätze, Farben und Bilder
 - vordefinierte Dialoge und „Wizards“
- Vorteile:
 - ◆ Native Dialogelemente sind häufig schneller, passen sich der Darstellung des Fenstersystems an.
 - ◆ weite Verbreitung



- Nachteile:
 - ◆ teilweise „kranke“ API (SWT)
 - ◆ nicht so mächtige Dialogelemente wie in Swing vorhanden
 - ◆ teilweise keine Garbage-Collection möglich → manuelle Freigabe von Betriebssystemressourcen!
 - ◆ keine Unterstützung von Applets
- Plattformen (teilweise 32/64-Bit):
 - ◆ Windows (Vista, 7, 8, 10)
 - ◆ Linux (x86/GTK, x86/Motif, PPC/GTK, S390/GTK)
 - ◆ Solaris (x86/GTK, Sparc/GTK)
 - ◆ AIX (PPC/Motif), HP-UX (ia64/Motif)
 - ◆ Mac OS X

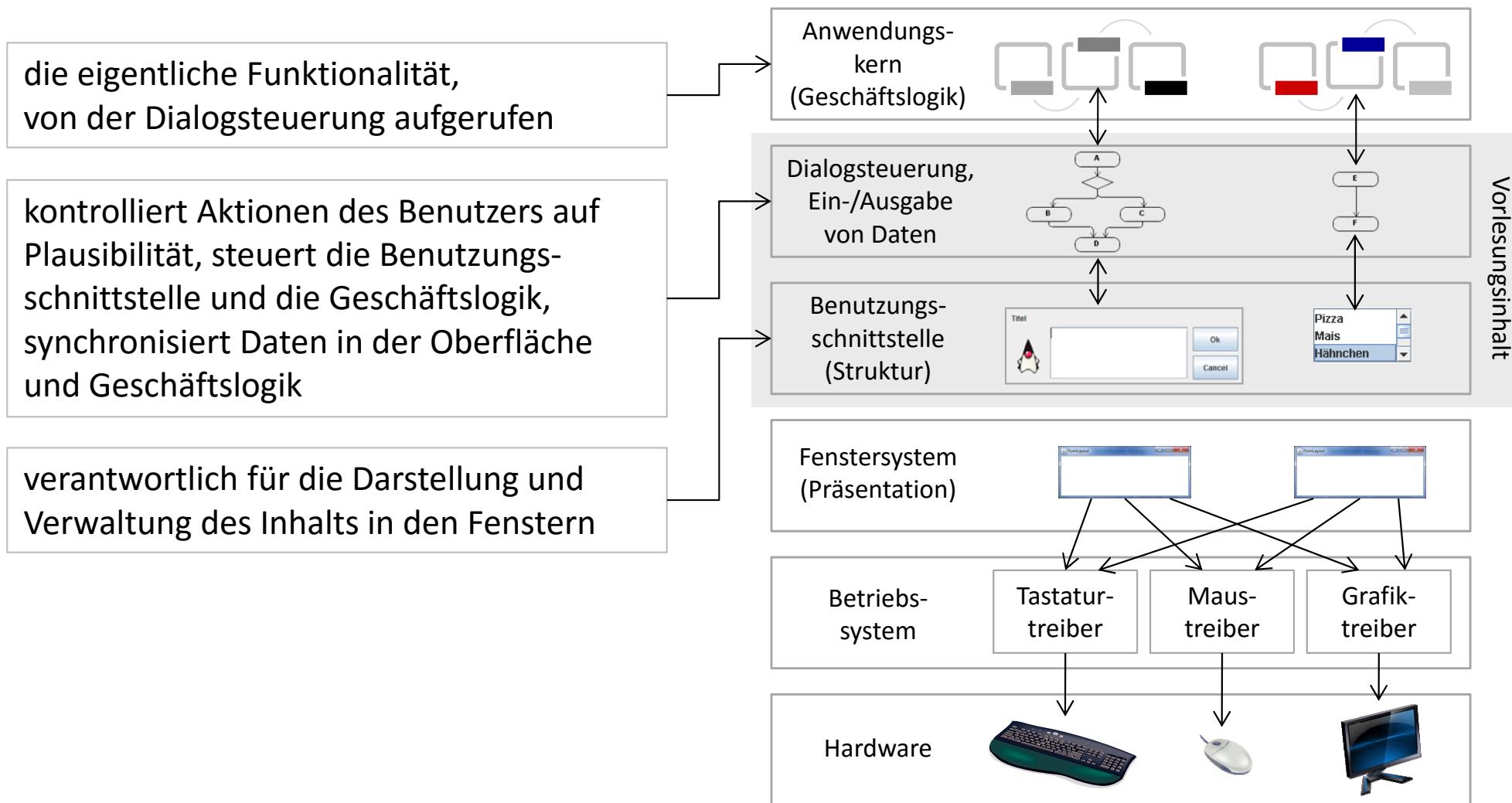


Warum SWT/JFace und RCP in der Vorlesung?

- RCP ist in der Industrie für Java-Anwendungen inzwischen stark verbreitet.
- Besitzt schöne Konzepte wie
 - ◆ Dependency Injection (seit e4)
 - ◆ modellbasierter Ansatz für die Struktur der Anwendung (seit e4)
 - ◆ Modularisierung mit OSGi
- einfache Erzeugung von Rich Thin Clients durch RAP (Rich Application Platform)
- Plattformunabhängigkeit
- Es sind viele Erweiterungen verfügbar (z.B. GEF für grafische Editoren, ...).
- Viele Konzepte werden unabhängig vom Framework vorgestellt (MVC, Befehls-Muster, Architekturansätze, ...).

Architekturen von Fat-Client-Anwendungen

Grundlagen



Architekturen von Fat-Client-Anwendungen

Grundlagen



- Struktur anhand eines sehr einfachen Beispiels

Kundenstammdaten

Eingabe der Kundendaten

Geben Sie die Stammdaten des Kunden ein.

Vorname: ←

Nachname:

Kundennummer: Im Ruhestand:

Familienstand: Kunde seit: ←

OK

Präsentation
(Farben,
Zeichensätze, ...)

Dialogsteuerung
(entsperrt, wenn die
Eingaben in Ordnung sind,
löst Aktion aus)

Struktur
(Widgets und
deren Anordnung)

Geschäftslogik
(Datum kleiner als
aktuelles Datum)

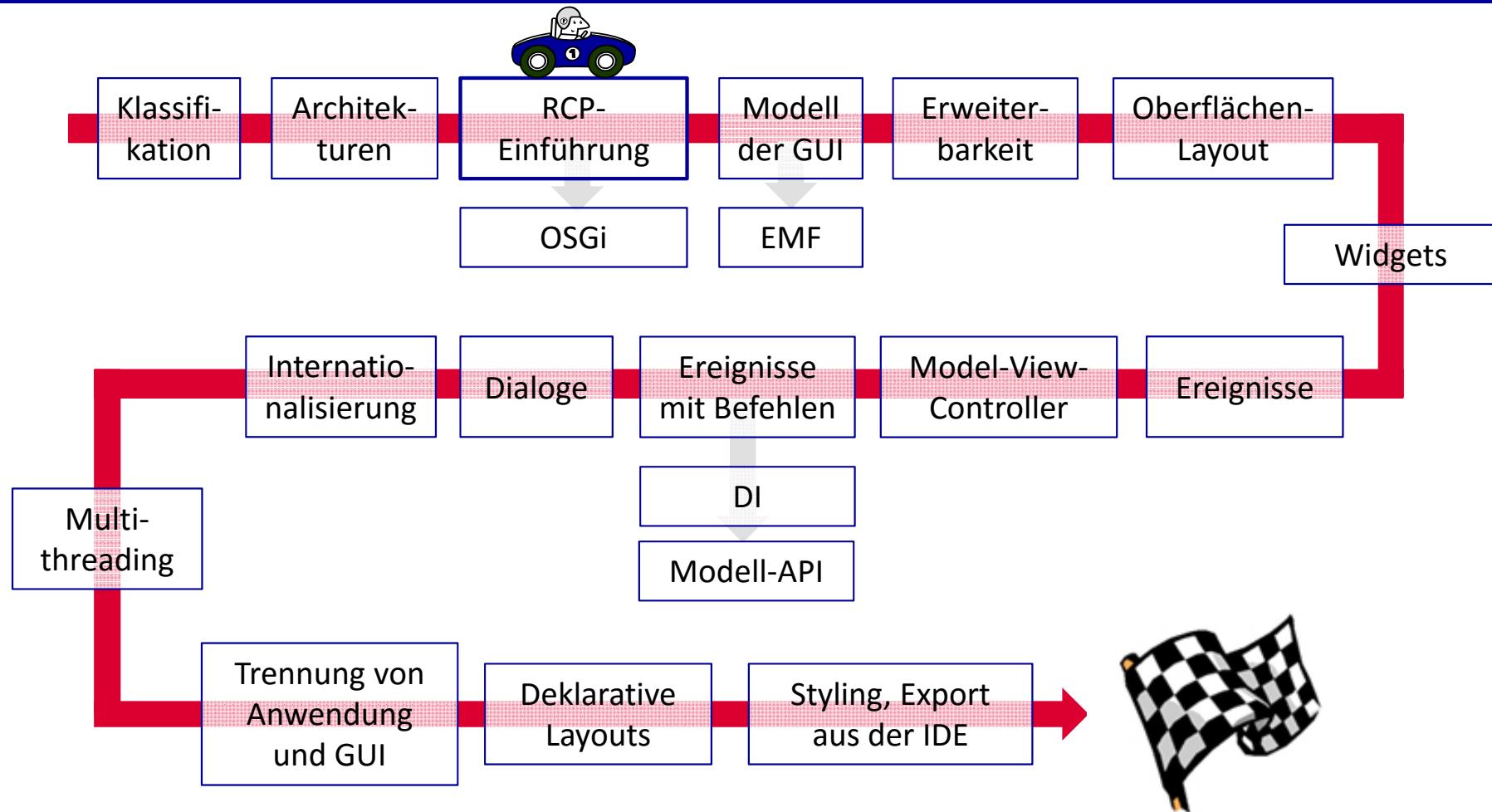
Ein- und Ausgabe
(Daten aus dem
Dialog)

```
public class Customer {  
    private String firstName;  
    private String lastName;  
    private int customerNumber;  
    private boolean retired;  
    private FamilyStatus familyStatus;  
    private Date customerSince;  
    // ...  
}
```



- Interaktionsmöglichkeiten mit einer Anwendung:
 1. Menü-Auswahl: Steuerung der Anwendung durch Menü-Kommandos
 2. Formular-basiert: Eingabe von Daten in Formulare, gut geeignet für einfach strukturierte Daten
 3. Direkte Manipulation: Bearbeitung von Objekten auf einer Arbeitsfläche durch Drag-and-Drop, kontextsensitive Menüs, ...
Beispiele:
 - Allgemein: Drag-and Drop für Dateioperationen, UML-Editoren, Zeichenprogramme, ...
 - Java-APIs: Naked Objects (jetzt Bestandteil von <http://isis.apache.org/index.html>), Eclipse GEF (<http://www.eclipse.org/gef/>), ...
 4. Sprachgesteuerte Interaktionen (natürliche Sprache, anwendungsspezifische Sprache): Anweisungen auf der Kommandozeile, eingebettete Skriptsprachen, Erkennung gesprochener Befehle, ...
- In der Vorlesung: Punkte 1-2 und etwas von Punkt 3

Die Eclipse Rich Client Platform





- Warum soll die Eclipse Rich Client Platform (RCP) in einer Benutzungsoberflächenvorlesung betrachtet werden?
 - ◆ Eclipse ist nicht nur eine IDE. Die IDE ist eine möglich Anwendung, die auf Basis der RCP entwickelt wurde.
 - ◆ Die RCP ist also eine Plattform für „Rich Clients“.
- Eigenschaften der RCP:
 - ◆ Komponenten-Modell: Sie bietet ein sauberes Komponenten-Modell. Jede Komponente (auch Plug-in genannt) besitzt
 - eine Version,
 - eine exakt festgelegte Schnittstelle nach außen und
 - genau spezifizierte Anforderungen an andere Komponenten, die sie zu ihrer Funktion benötigt.
 - ◆ Middleware: Ein Framework erleichtert die Erstellung von Anwendungen, weil viele Schritte nicht mehr manuell durchgeführt werden müssen.
 - ◆ Native Anwendungen: Durch SWT/JFace haben die Anwendungen das Aussehen wie alle anderen Anwendungen auf dem Betriebssystem.

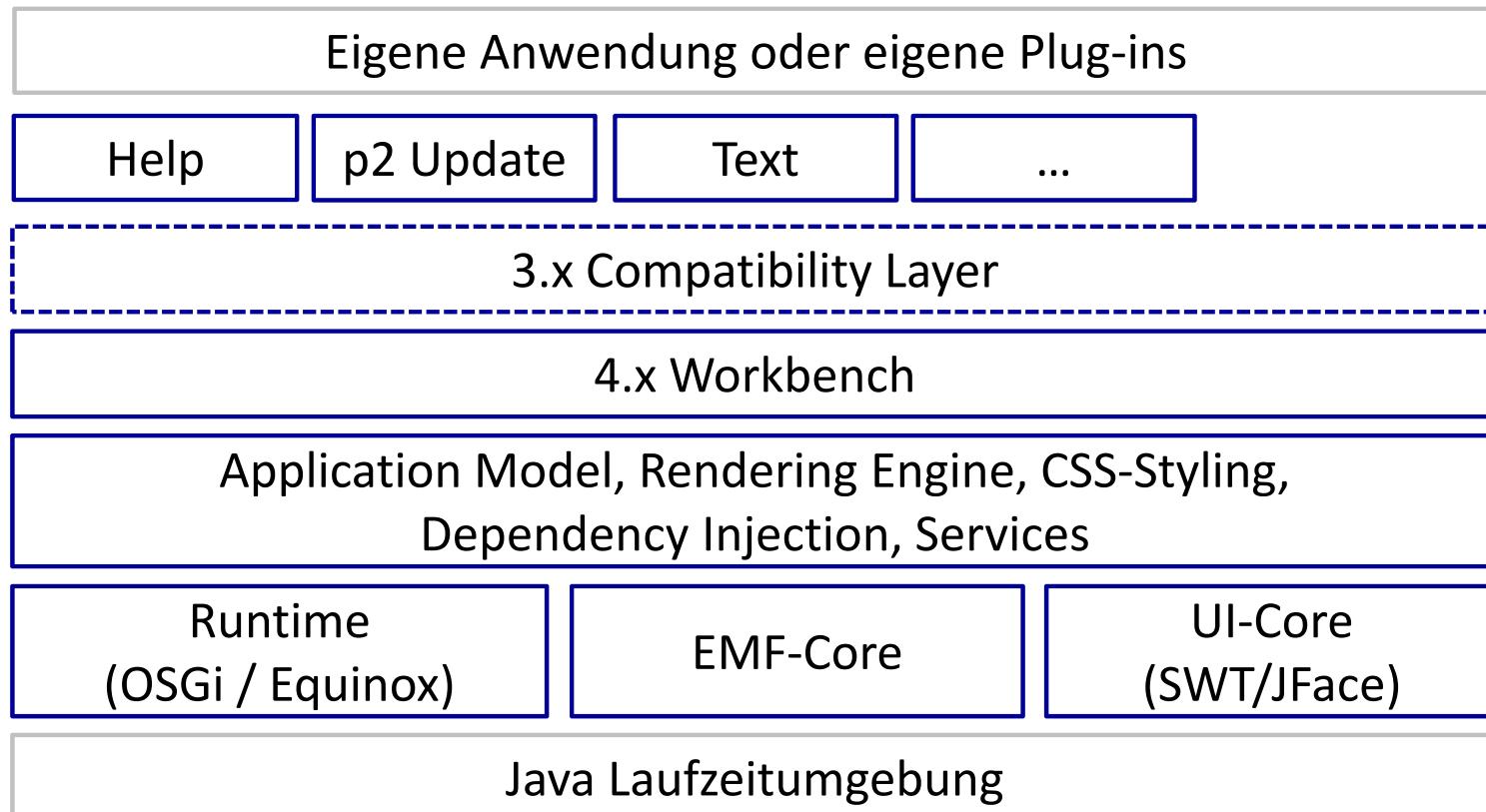


- ◆ Portabilität: Die RCP unterstützt alle wichtigen Betriebssysteme:
 - Die Anwendungen werden unabhängig vom Betriebssystem erstellt und laufen „überall“.
 - Gerätetyp: Desktop-PCs
- ◆ Installation und Updates können (automatisch) über das Netz erfolgen.
- ◆ Serverloser Betrieb: Es ist im Gegensatz zu Thin Clients kein Server erforderlich (hängt natürlich von der Anwendung ab). Client/Server-Anwendungen können bei Serverstörungen Daten in einem Cache ablegen und später synchronisieren.
- ◆ Gute Entwicklungsunterstützung durch die Eclipse IDE
- ◆ Komponentenbibliotheken: Es existieren viele Bibliotheken, die in eigene Anwendungen integriert werden können (z.B. ein Hilfe-System).

Die Eclipse Rich Client Platform



- Schematischer Aufbau einer möglichen RCP-Anwendung mit eigener Benutzeroberfläche:



angelehnt an [Lars Vogel, „Eclipse 4 Application Development“]

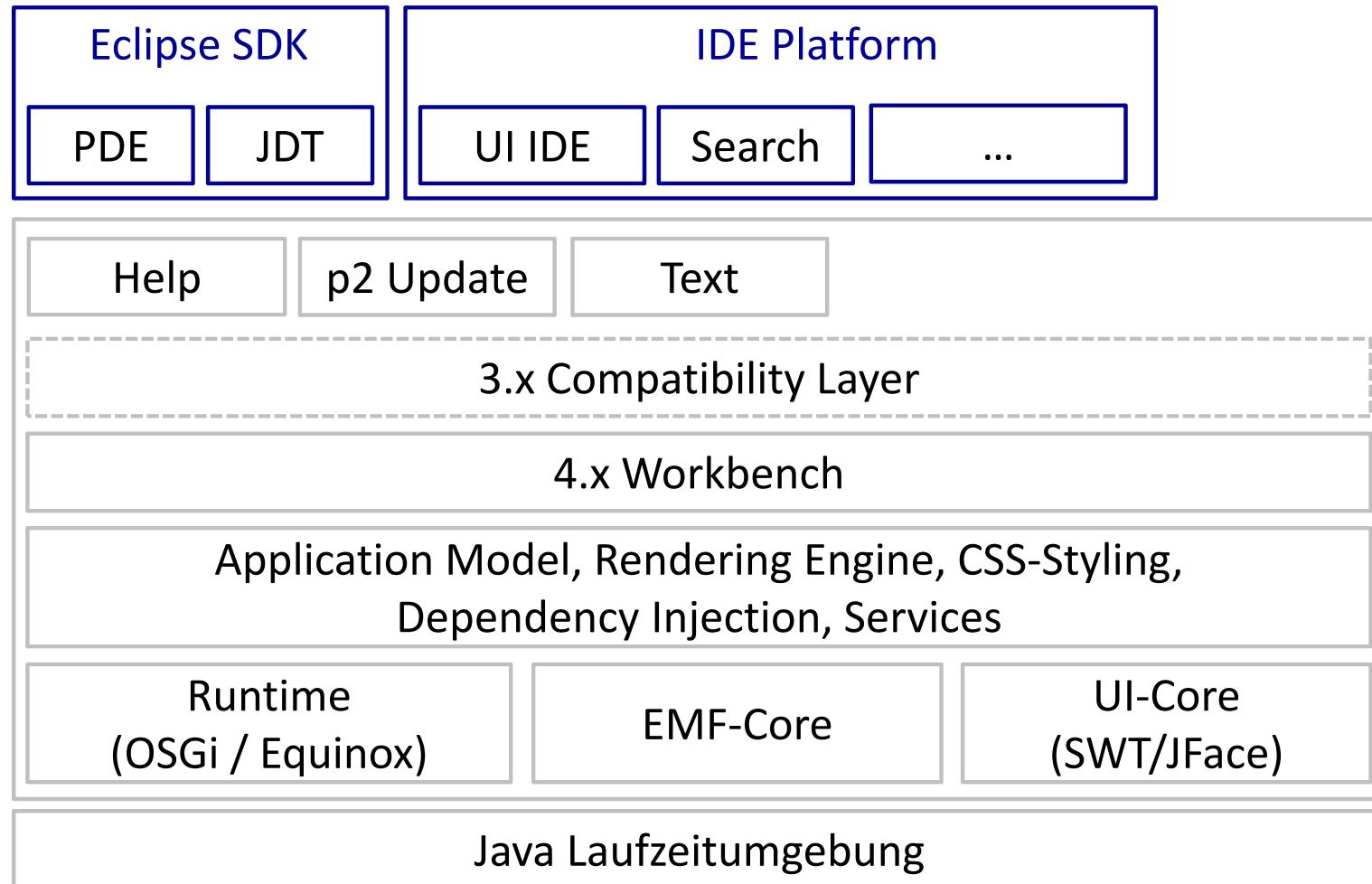


- Runtime:
 - ◆ Standard für Komponenten-basierte Anwendungen auf Java-Basis
 - ◆ Equinox ist eine Implementierung des Standards durch Eclipse.
 - ◆ Alle RCP-Komponenten sind OSGi-Plug-ins.
- Application Model, ...: Modell der Anwendung, weitere Basisdienste
- UI-Core: zur Erstellung grafischer Oberflächen (z.B. für Dialoge, Menüs, ...)
- 4.x Workbench: Sie stellt die Workbench bereit (eine leere grafische Anwendung). In die Workbench können Views und Editoren, Perspektiven, Menüs usw. integriert werden.
- 3.x Compatibility Layer: damit „alte“ Anwendungen, die auf Eclipse 3.x basieren, weiterhin funktionieren
- EMF-Core: Das Application Modell verwendet intern das EMF.
- Optionale Komponenten: Hilfesystem, Update-Manager, eine Komponente zur Realisierung web-ähnlicher Formulare (Forms)
- Anmerkung: Die Eclipse-IDE ist eine Anwendung, die diese Komponenten nutzt.

Die Eclipse Rich Client Platform



- Wie sieht es mit Eclipse als Java-IDE aus (stark vereinfacht)?:



Die Eclipse Rich Client Platform



Vorbereitung und Installation

- Anforderungen an die **IDE** selbst:
 - ◆ Am besten „Eclipse for RCP and RAP Developers“) verwenden:
<http://www.eclipse.org/downloads/> oder unter **Help → Install New Software → Mars → General Purpose Tools → Eclipse Plug-in Development Environment** installieren
 - ◆ Unter **Help → Install new Software → Mars → General Purpose Tools** diese Einträge installieren:
 - **Eclipse e4 Tools Developer Resources**
 - **Eclipse XWT**
 - **Eclipse XWT Workbench Integration**
 - ◆ Unter **Help → Install new Software → Mars → Modeling** diese Einträge installieren:
 - **Ecore Diagram Editor (SDK)**
 - **EMF – Eclipse Modeling Framework SDK**



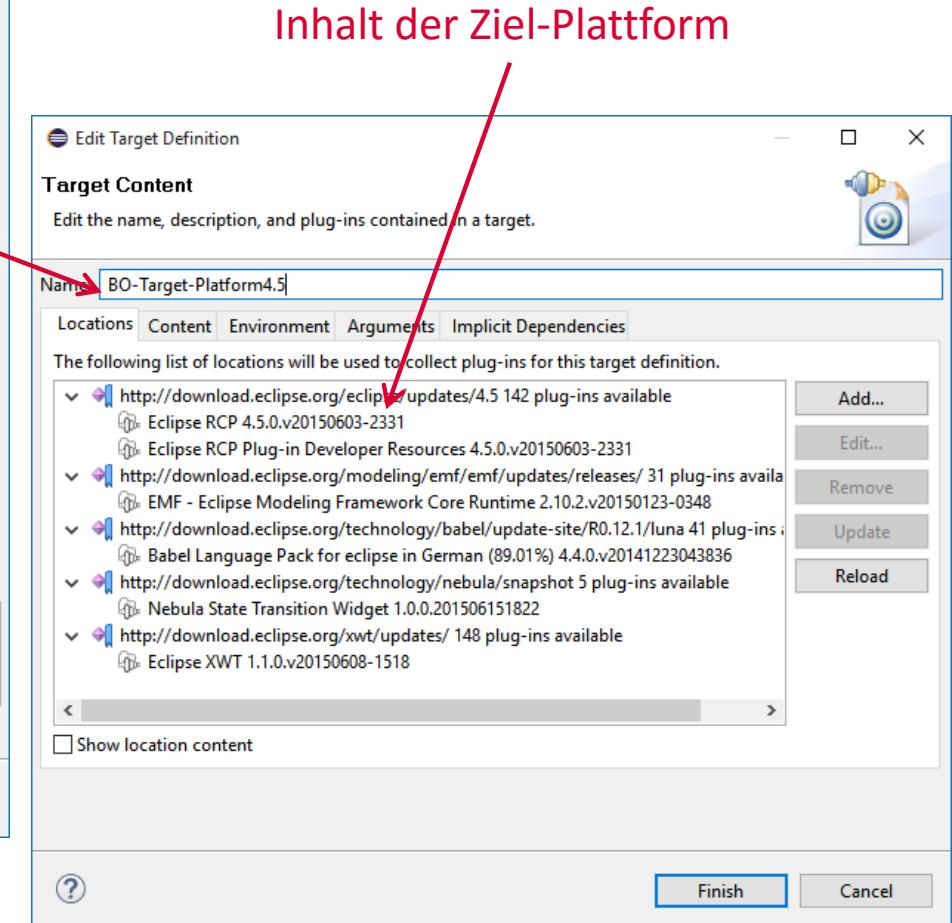
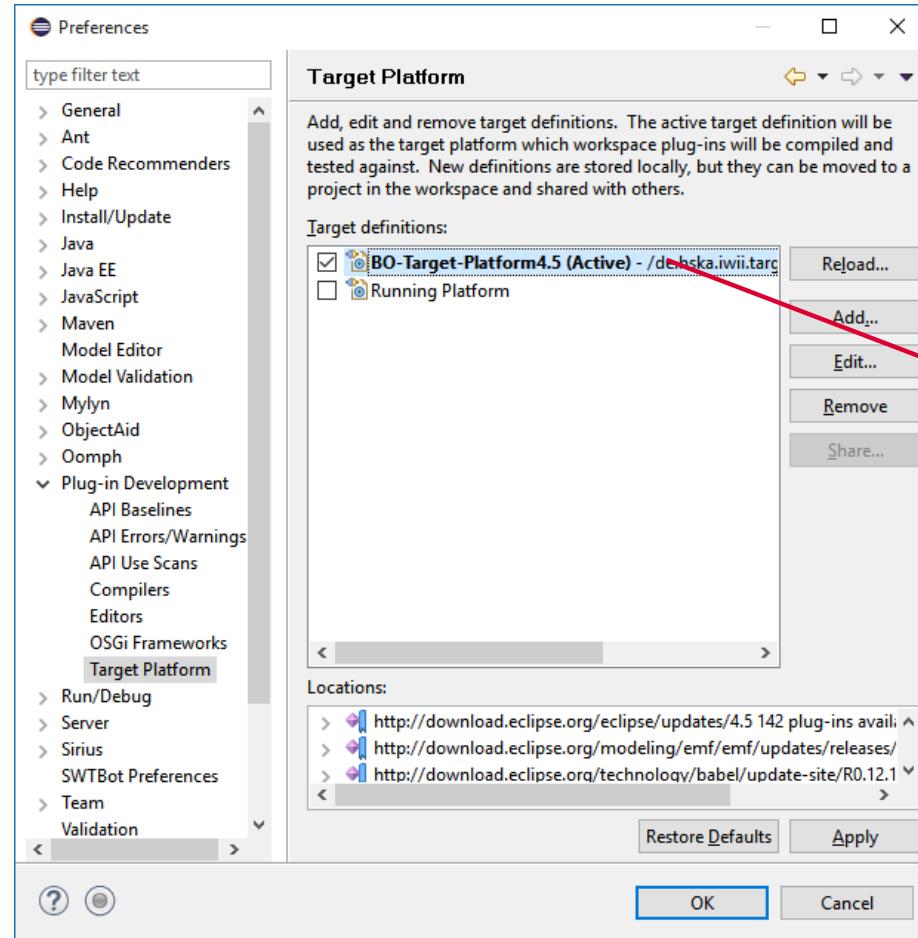
- ◆ GUI-Editor (bereits in IDE für RCP und RAP enthalten): **Help → Install New Software → Mars → General Purpose Tools → SWT Designer** und **Windows Builder Core UI** installieren
- Zusätzlich sollte das **RCP-SDK** installiert werden:
 - ◆ Vorteil: Die eigene Anwendung kann eine andere Version als die IDE verwenden.
 - ◆ Es lassen sich Abhängigkeiten zu IDE-Plug-ins vermeiden.
 - ◆ Die Basis-Komponenten werden in den Eclipse-Einstellungen als „Target-Platform“ angesprochen.
 - ◆ Verzeichnis der Target-Platform im Pool 203:
C:\Programme\eclipse\eclipse-RCP-SDK
 - ◆ Quelle:
 - Manueller Download von <http://download.eclipse.org/eclipse/downloads/>
 - Oder über Update Sites (hier in der Vorlesung verwendet)

Die Eclipse Rich Client Platform

Vorbereitung (RCP-SDK)



■ Die Ziel-Plattform:

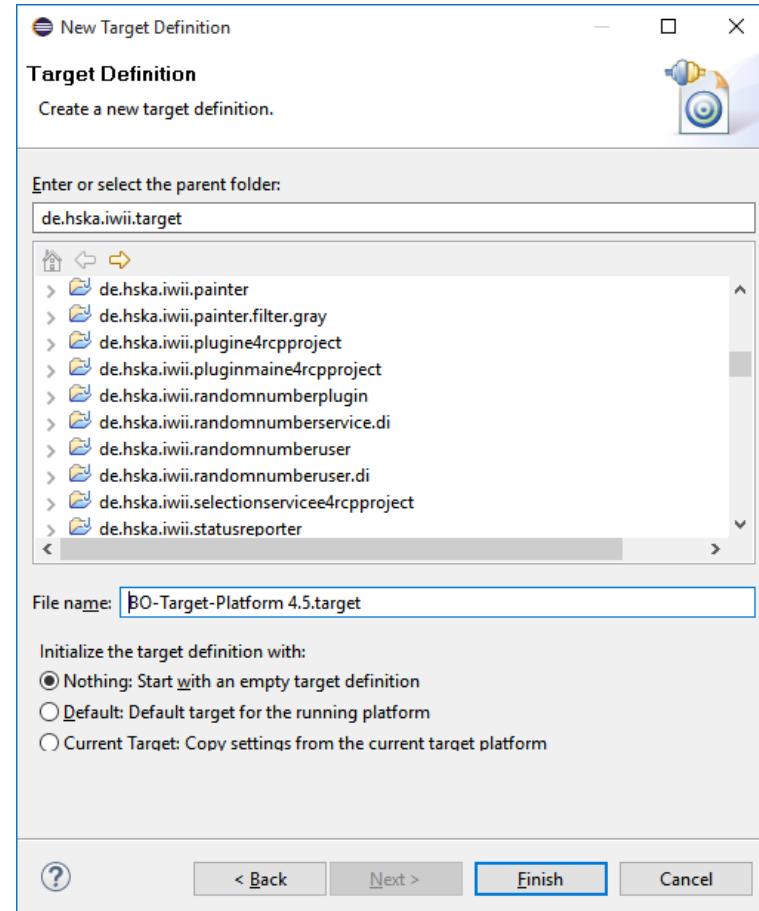


Die Eclipse Rich Client Platform

Vorbereitung (RCP-SDK)



- Bau der „Target-Platform“:
 - ◆ Zunächst ein leeres Projekt Anlegen (z.B.
de.hska.iwii.target)
 - ◆ Menü **File → New → Other...**
→ **Plug-in Development**
→ **Target Definition**
 - ◆ Aussagekräftigen Namen geben, als „parent folder“ das leere Projekt angeben.

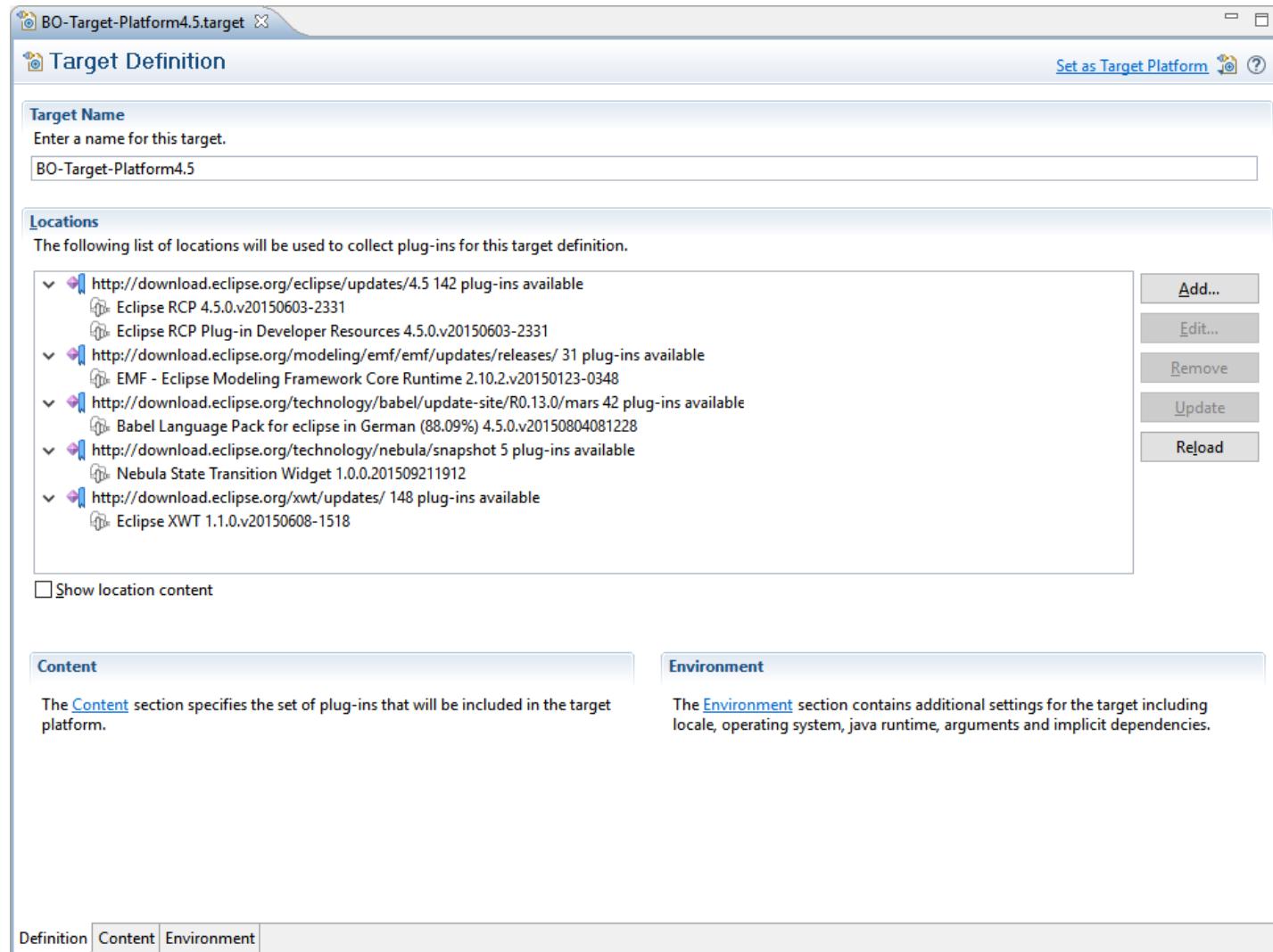


Die Eclipse Rich Client Platform

Vorbereitung (RCP-SDK)



- ◆ Hier können mit **Add...** Plug-ins aus einer „Software Site“ hinzugefügt werden:



Die Eclipse Rich Client Platform

Vorbereitung (RCP-SDK)



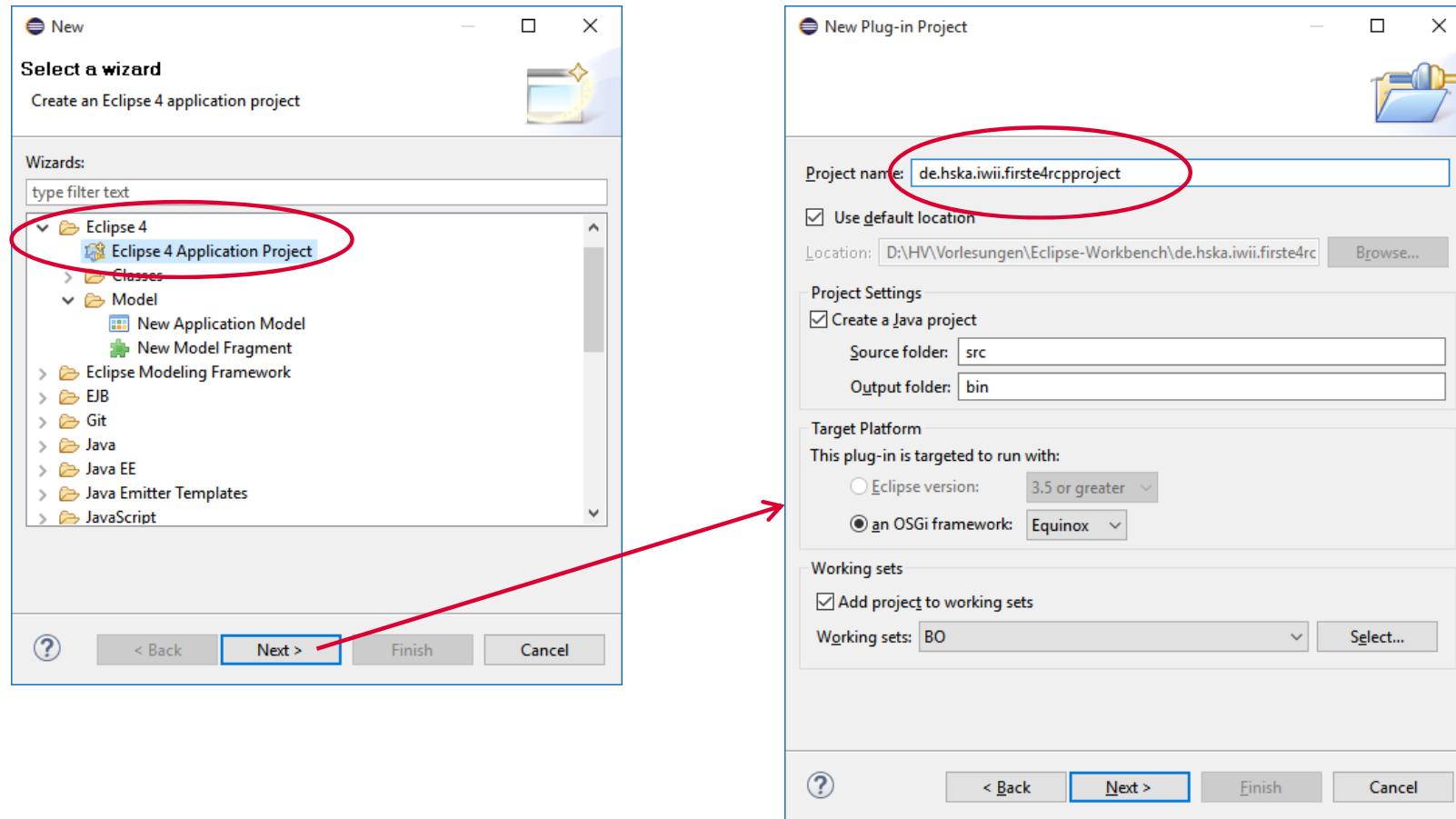
- ◆ In der Ziel-Plattform werden für die Vorlesung die folgenden Plug-ins benötigt:
 - RCP: <http://download.eclipse.org/eclipse/updates/4.5>, Plug-ins „Eclipse RCP“ und „Eclipse RCP Plug-in Developer Resources“
 - EMF: <http://download.eclipse.org/modeling/emf/emf/updates/releases>, Plug-in „EMF – Eclipse Modeling Framework Core Runtime“
 - „Babel Language Pack“, damit die internen Texte des RCP auch auf deutsch dargestellt werden: <http://download.eclipse.org/technology/babel/update-site/R0.13.0/mars>, genügt „Babel Language Pack for eclipse in <sprache>“
 - „Nebula-Widgets“ für ein Beispiel:
<http://download.eclipse.org/technology/nebula/snapshot>, für das Vorlesungsbeispiel reicht „Nebula State Transition Widget“
 - XWT: <http://download.eclipse.org/xwt/updates>, Plug-in „Eclipse XWT“
- Wenn Ihnen die Sache mit dem manuellen Zusammenbau zu kompliziert ist: Im Ilias finden Sie das Projekt mit der „Target Definition“ für die Vorlesung: Datei im „Target Editor“ öffnen, „Set as Target Platform“ anklicken

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Eclipse besitzt einen Wizard, um eine erste, leere RCP-Anwendung zu erzeugen: **File → New → Project → Eclipse 4 → Eclipse 4 Application Project**

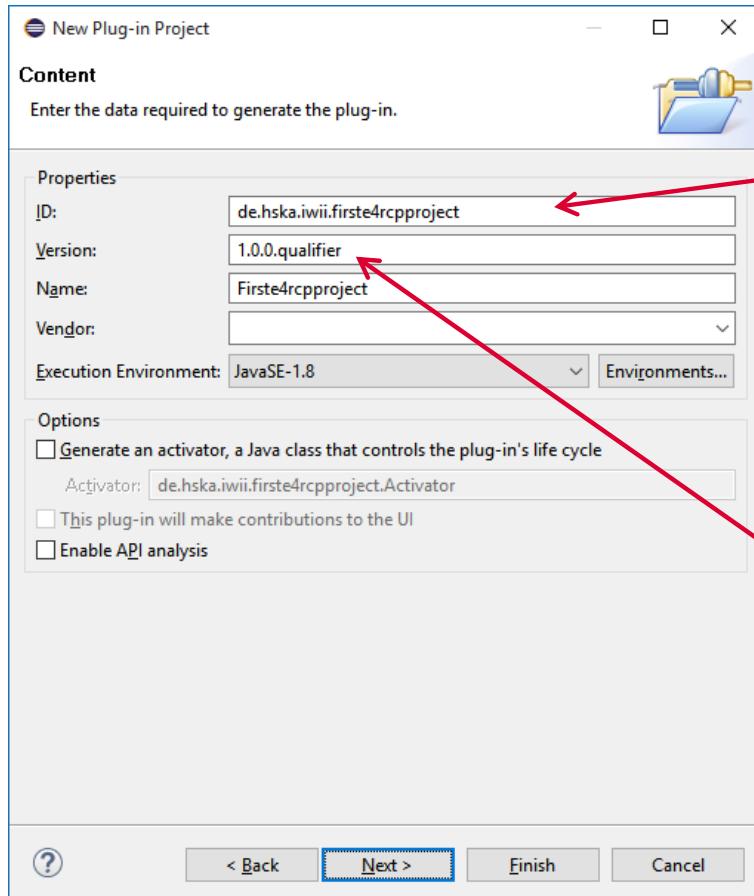


Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



■ Nach **Next:**



Eindeutige Identifikation der Anwendung

Version der Anwendung

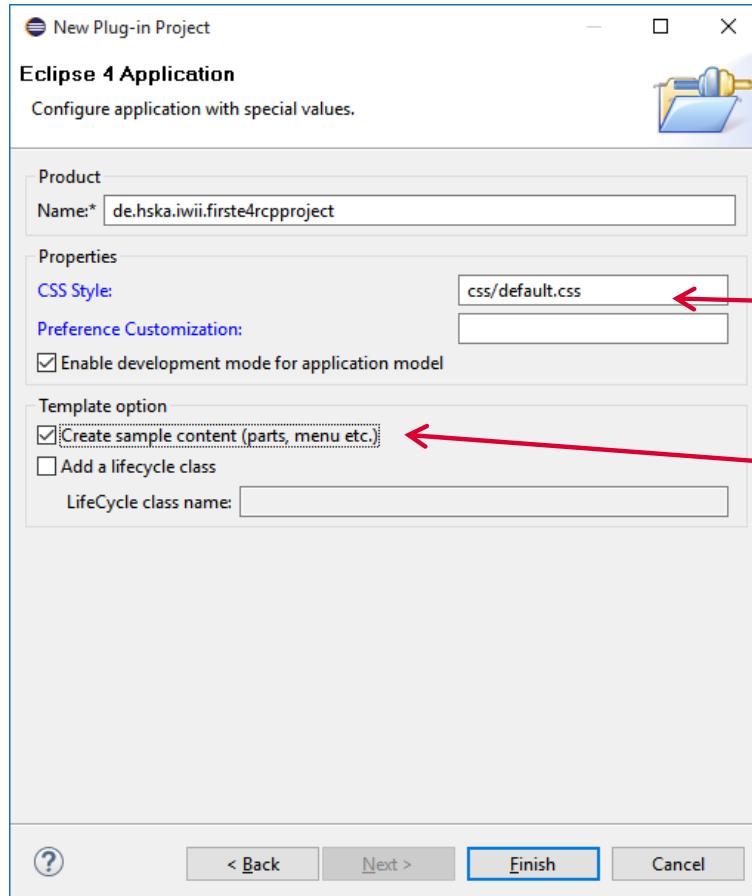
- Major number: Neue Nummer zeigt Inkompatibilität zur Vorversion.
- Minor number: Neue Version zeigt neue Funktionalität im Vergleich zur Vorversion unter Beibehaltung der Kompatibilität
- Micro number: Keine neuen Funktionen im Vergleich zur Vorversion, lediglich Fehlerbehebungen
- qualifier: Beliebiger Text, das Wort „qualifier“ wird von Eclipse bei einem Export durch Datum und Uhrzeit ersetzt (=Build-Nummer)

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Nach **Next** noch einige kleine Voreinstellungen übernehmen:



Dateiname der CSS-Datei

Das Projekt soll schon einige Bestandteile wie Parts usw. enthalten.

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Nach **Finish** wird die Anwendung erzeugt:

The screenshot shows the 'Overview' tab of the Eclipse RCP Project Properties dialog for the project 'de.haska.iwii.firste4rcpproject'. The 'General Information' section displays the plugin's ID as 'de.haska.iwii.firste4rcpproject', version as '1.0.0.qualifier', name as 'Firste4rcpproject', and勾选ed 'This plug-in is a singleton'. The 'Execution Environments' section lists 'JavaSE-1.8' with buttons for 'Add...', 'Remove', 'Up', and 'Down'. The 'Plug-in Content' section describes dependencies and runtime libraries. The 'Extension / Extension Point Content' section details contributions made to the platform. The 'Testing' section provides links to launch the application or RAP application. The 'Exporting' section outlines the steps to package and export the plugin. The bottom navigation bar includes tabs for Overview, Dependencies, Runtime, Extensions, Extension Points, Build, MANIFEST.MF, plugin.xml, and build.properties.

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Datei **<ID>.product** öffnen:

The screenshot shows the Eclipse Product Configuration Editor window for a project named "de.hska.iwii.erste4rcpproject.product". The window has tabs at the bottom: Overview, Dependencies, Configuration, Launching, Splash, Branding, Customization, Licensing, and Updates. The Overview tab is selected.

General Information
This section describes general information about the product.

ID: de.hska.iwii.erste4rcpproject

Version: 1.0.0.qualifier

Name: de.hska.iwii.erste4rcpproject

The product includes native launcher artifacts

Product Definition
This section describes the launching product extension identifier and application.

Product: de.hska.iwii.erste4rcpproject.product

Application: org.eclipse.e4.ui.workbench.swt.E4Application

The [product configuration](#) is based on: plug-ins features

Testing

1. [Synchronize](#) this configuration with the product's defining plug-in.
2. Test the product by launching a runtime instance of it:
 - [Launch an Eclipse application](#)
 - [Launch a RAP Application](#)
 - [Launch an Eclipse application in Debug mode](#)
 - [Launch a RAP Application in Debug mode](#)

Exporting

Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

To export the product to multiple platforms see the [Cross Platform Wiki Page](#).

Overview | Dependencies | Configuration | Launching | Splash | Branding | Customization | Licensing | Updates

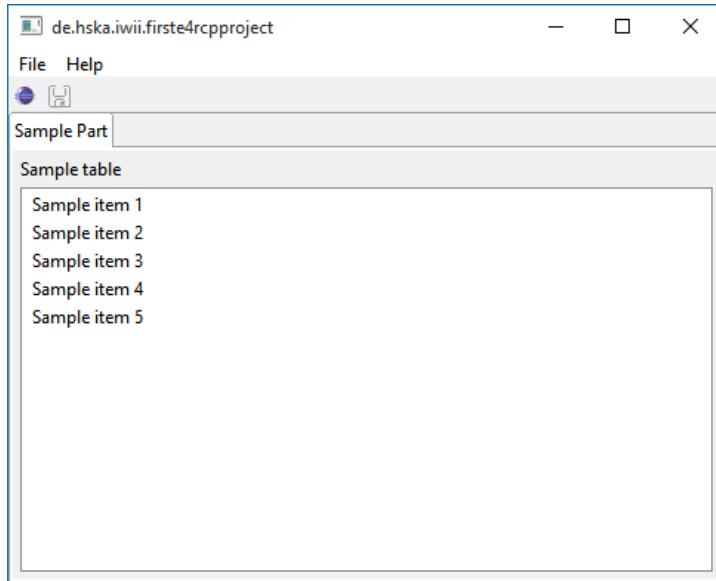
Starten

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Gestartete Anwendung mit Menü, Toolbar und Workbench-Fenster:



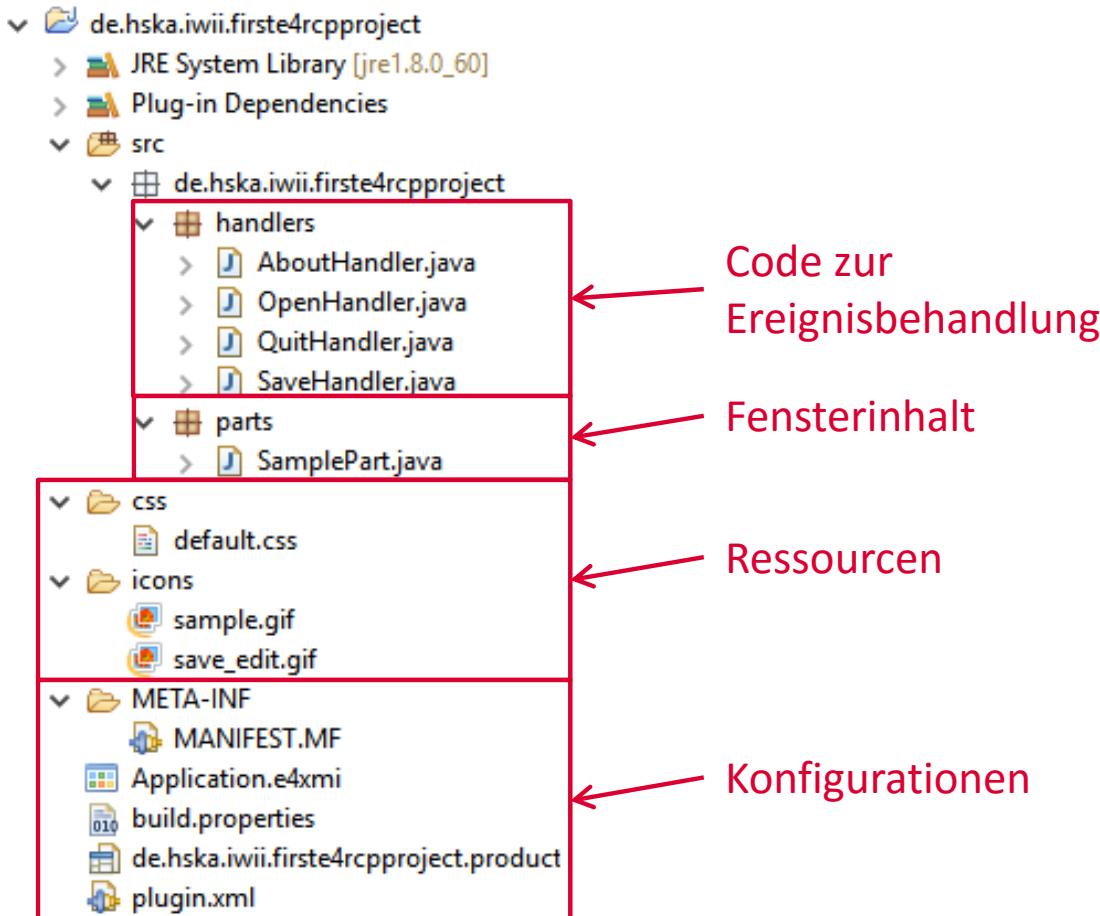
- Die wichtigsten Eigenschaften der RCP werden später vorgestellt. Die RCP dient hier erst einmal als Basis für die SWT- und JFace-Einführungen.

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Erzeugte Dateien im Projekt:

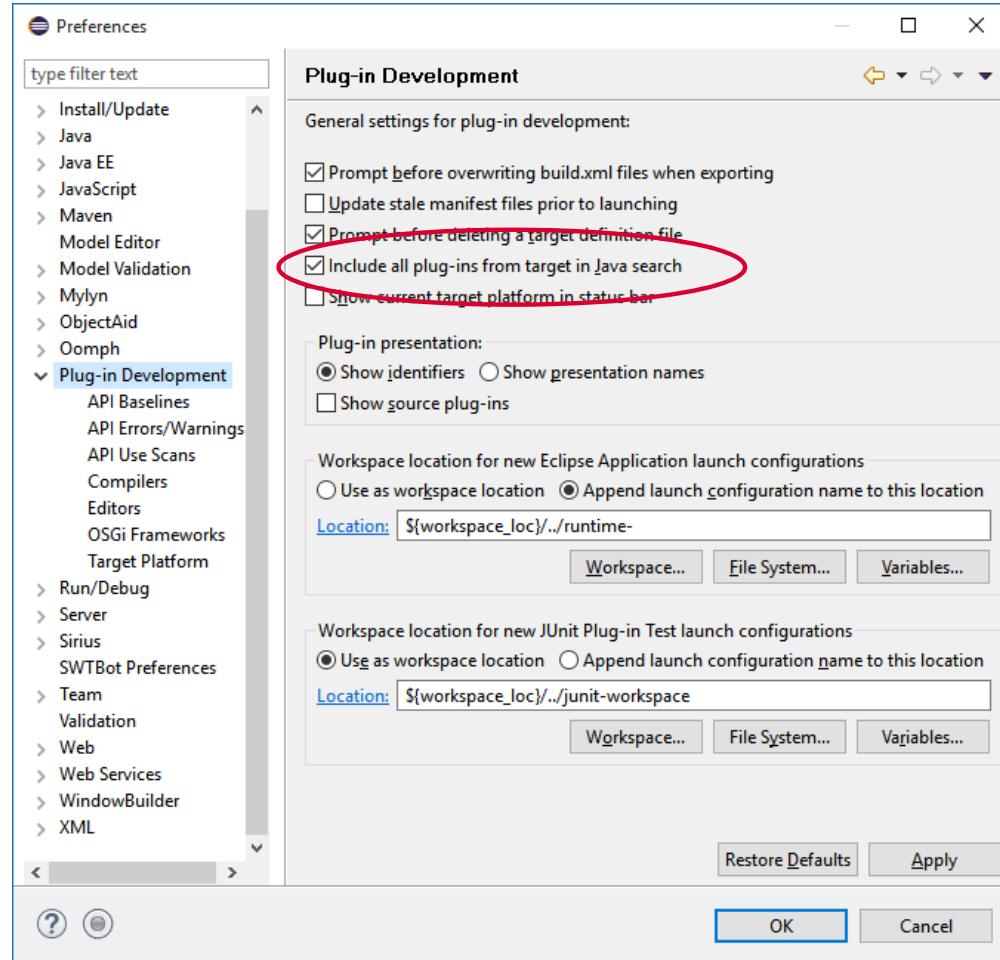


Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- JavaDoc-Unterstützung für die Eclipse-API einschalten:





- Einige wichtige Klassen und Dateien im erzeugten Projekt:
 - ◆ **META-INF/MANIFEST.MF**: OSGi-Konfiguration wie ID des „Programmes“, Abhängigkeiten von Plug-ins, Version, ...
 - ◆ **plugin.xml**: Erweiterungen
 - ◆ **<id>.product**: Das Produkt beschreibt das Projekt zur Entwicklungszeit. Dazu gehören Abhängigkeiten von Plug-ins, Splash-Screen, Konfigurationsdatei beim Start, Icons, notwendiges JRE, „About“-Dialog, ...
 - ◆ **Application.e4xmi**: das „Application Model“
 - ◆ **build.properties**: Eigenschaften, die für den Zusammenbau der Anwendung benötigt werden, wie Ausgabeverzeichnis, ...
 - ◆ **css/**: CSS-Dateien zur Gestaltung der Oberfläche
 - ◆ **icons/**: Verwendete Icons (z.B. in Toolbars)
 - ◆ **<package>/handlers/**: Klassen zur Ereignisbehandlung
 - ◆ **<package>/parts**: Klassen, die „Fensterinhalte“ definieren → kommt noch genauer

Die Eclipse Rich Client Platform

Aufbau einer RCP-Anwendung



- Abhängigkeiten von anderen Plug-ins, die automatisch eingefügt werden:

The screenshot illustrates the automatic dependency management in the Eclipse RCP Platform. On the left, the 'Dependencies' view shows a list of required plug-ins. A red arrow points from the 'Required Plug-ins' list to the 'Manifest' view on the right, where the corresponding dependency entries are highlighted with a blue selection bar.

Dependencies View (Top Left):

- Required Plug-ins:
 - javax.inject (1.0.0)
 - org.eclipse.core.runtime (3.9.0)
 - org.eclipse.swt (3.102.0)
 - org.eclipse.e4.ui.model.workbench (1.0.0)
 - org.eclipse.jface (3.9.0)
 - org.eclipse.e4.ui.services (1.0.0)
 - org.eclipse.e4.ui.workbench (1.0.0)
 - org.eclipse.e4.core.di (1.3.0)
 - org.eclipse.e4.ui.di (1.0.0)
 - org.eclipse.e4.core.contexts (1.3.0)

Manifest View (Bottom Left):

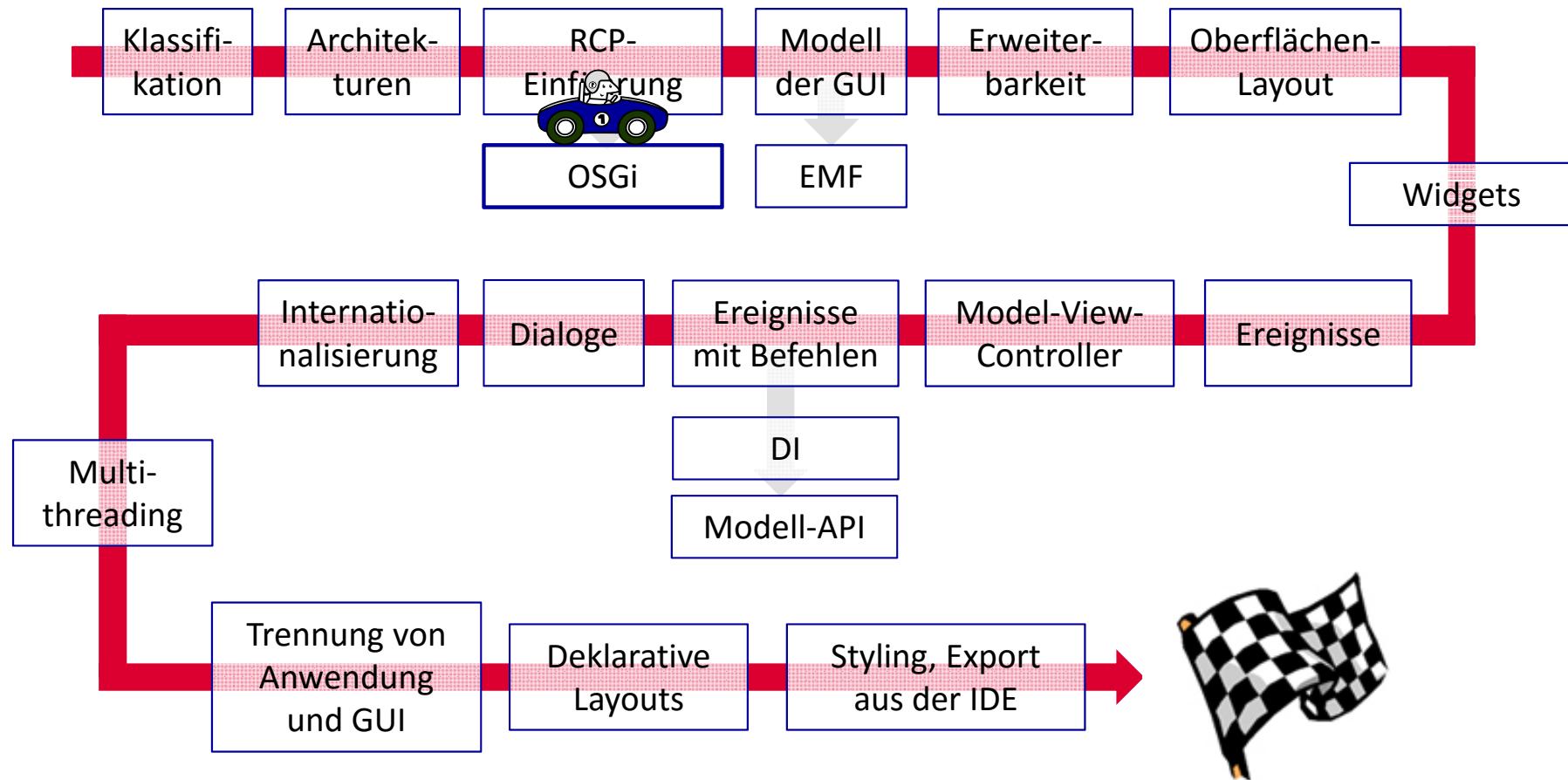
```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Firste4rcpproject
Bundle-SymbolicName: de.haska.iwii.firste4rcpproject;singleton:=true
Bundle-Version: 1.0.0.qualifier
Require-Bundle: javax.inject;bundle-version="1.0.0",
    org.eclipse.core.runtime;bundle-version="3.9.0",
    org.eclipse.swt;bundle-version="3.102.0"
    org.eclipse.e4.ui.model.workbench;bundle-version="1.0.0",
    org.eclipse.jface;bundle-version="3.9.0",
    org.eclipse.e4.ui.services;bundle-version="1.0.0",
    org.eclipse.e4.ui.workbench;bundle-version="1.0.0",
    org.eclipse.e4.core.di;bundle-version="1.3.0",
    org.eclipse.e4.ui.di;bundle-version="1.0.0",
    org.eclipse.e4.core.contexts;bundle-version="1.3.0"
Import-Package: javax.annotation;version="1.0.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
```

Hierarchical view of plug-ins required by 'de.haska.iwii.firste4rcpproject (1.0.0.qualifier)' (Right):

- de.haska.iwii.firste4rcpproject (1.0.0.qualifier)
 - javax.annotation (1.2.0.v201401042248)
 - javax.inject (1.0.0.v20091030)
 - org.eclipse.core.runtime (3.11.0.v20150405-1723)
 - javax.annotation (1.2.0.v201401042248)
 - javax.inject (1.0.0.v20091030)
 - org.eclipse.core.contenttype (3.5.0.v20150421-2214)
 - org.eclipse.core.jobs (3.7.0.v20150330-2103)
 - org.eclipse.equinox.app (1.3.300.v20150423-1356)
 - org.eclipse.equinox.common (3.7.0.v20150402-1709)
 - org.eclipse.equinox.preferences (3.5.300.v20150408-1437)
 - org.eclipse.equinox.registry (3.6.0.v20150318-1503)
 - org.eclipse.osgi (3.10.100.v20150529-1857)
 - org.eclipse.e4.core.contexts (1.4.0.v20150421-2214)
 - org.eclipse.e4.core.di (1.5.0.v20150421-2214)
 - org.eclipse.e4.ui.di (1.1.0.v20150422-0725)
 - org.eclipse.e4.ui.model.workbench (1.1.100.v20150407-1430)
 - org.eclipse.e4.ui.services (1.2.0.v20150422-0725)
 - org.eclipse.e4.ui.workbench (1.3.0.v20150531-1948)
 - org.eclipse.jface (3.11.0.v20150602-1400)
 - org.eclipse.swt (3.104.0.v20150528-0211)

OSGi

Problemstellung

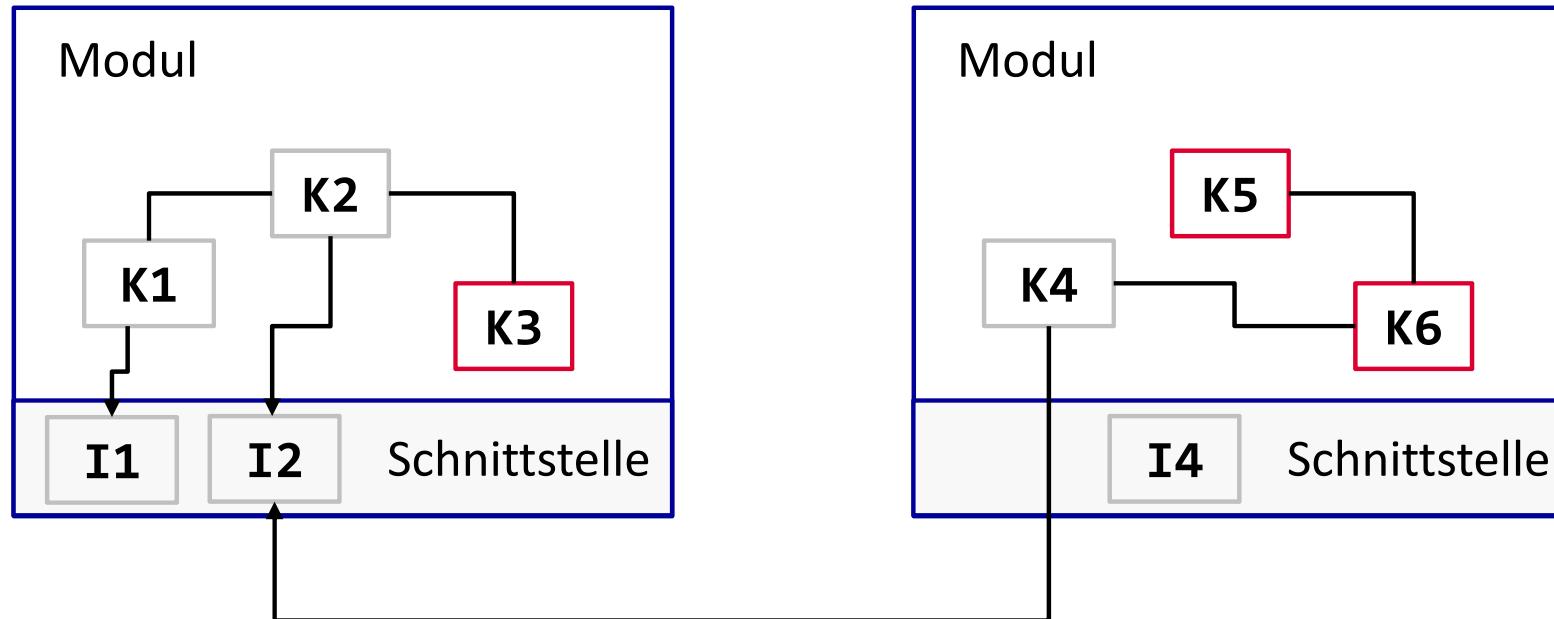




- Bisher tauchte immer wieder der Begriff „Plug-in“ auf. Was bedeutet er?
- Rückblick auf „Informatik 2“:
 - ◆ Ein Modul (eine Komponente)
 - kapselt ein Anliegen („Concern“),
 - besitzt eine (möglichst) unveränderliche öffentliche Schnittstelle und
 - besitzt eine interne Implementierung der Schnittstelle, die nicht nach „außen“ sichtbar ist. Die Implementierung kann sich durchaus ändern.
 - ◆ Module haben Beziehungen untereinander, indem sie die Schnittstellen anderer Module kennen.
- Fragen:
 - ◆ Wie lassen sich die internen Implementierungen verbergen?
 - ◆ Wie können die Verbindungen zwischen den Modulen beschrieben werden?
 - ◆ Wie können mehrere Versionen eines Moduls parallel eingesetzt werden?
 - ◆ Wie lassen sich Module zur Laufzeit austauschen?
 - ◆ Wie können Module aktiviert und deaktiviert werden?



- Beispiel:



K4 verwendet K2,
kennt aber nur dessen
Schnittstelle I2



- OSGi (früher: **Open Services Gateway initiative**, jetzt ein eigenständiger Name) wurde ursprünglich für die Gebäudesystemtechnik entworfen.
- Es läuft auf Gateways, die Geräte mit dem Internet verbinden. Ziele:
 - ◆ Leichte Einbindung neuer Geräte durch neue Dienste auf dem Gateway.
 - ◆ Installation und Verwaltung der Software auf dem Gateway über das Internet ohne
 - die Software auf einem Gateway anhalten zu müssen und
 - ohne das Gateway neu starten zu müssen.
- OSGi ist eine Lösung für Komponenten-orientierte Java-Software.
- Nach der Standardisierung (siehe <http://www.osgi.org/>) hat OSGi Einzug in viele Bereiche gehalten:
 - ◆ Smartphones
 - ◆ Automobil-Bereich (z.B. im 5er BMW für Telematik und Unterhaltung)
 - ◆ Desktop-Anwendungen (hauptsächlich Eclipse)
 - ◆ Serverseitige Dienste



Was bietet OSGi?

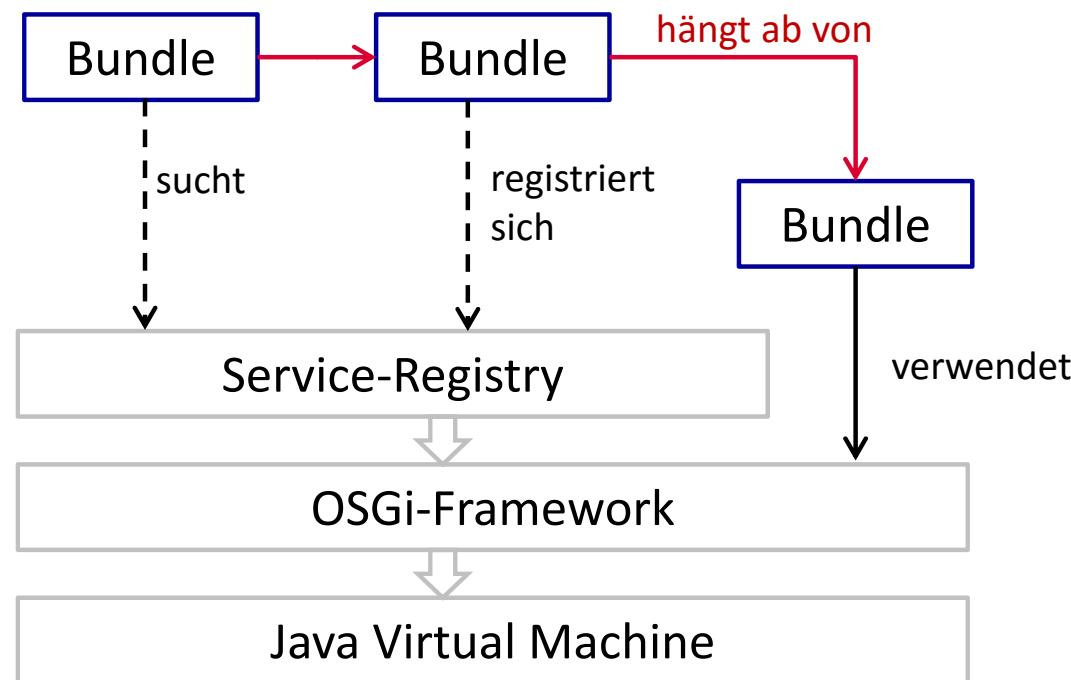
- Modularisierung:
 - ◆ Ein Bundle (= ein Plug-in, = ein Modul) besteht aus vielen Paketen.
 - ◆ Es wird deklarativ festgelegt,
 - welche Pakete die öffentliche Schnittstelle des Bundles darstellen und
 - welche Abhängigkeiten dieses Bundle von anderen hat.
 - ◆ Auf interne Pakete kann nicht zugegriffen werden → jedes Bundle besitzt einen eigenen Classloader.
 - ◆ Jedes Bundle besitzt eine Version.
 - Ein Bundle kann von einer bestimmten Version oder einem Versionsbereich eines anderen Bundles abhängig sein.
 - Es kann ein Bundle in unterschiedlichen Versionen parallel verwendet werden.
 - ◆ Bundles lassen sich zur Laufzeit austauschen, ohne die komplette Anwendung stoppen zu müssen.



- Dienstorientiertes („service oriented“) Programmiermodell:
 - ◆ Ein Modul stellt lediglich Schnittstellen, aber keine konkreten Implementierungen nach außen zur Verfügung.
 - ◆ Die Schnittstellen mit ihren Implementierungen werden als Dienste in einer zentralen „Registry“ hinterlegt.
 - ◆ Soll ein Dienst verwendet werden, dann wird versucht, aus der „Registry“ die Schnittstellenreferenz auszulesen.
 - ◆ Der Anwender eines Moduls weiß gar nicht, welches Bundle den gesuchten Dienst zur Verfügung stellt → OSGi löst die Abhängigkeiten auf.
- Der OSGi-Standard definiert Dienste (Services), die allen anderen Modulen zur Verfügung stehen:
 - ◆ Logging
 - ◆ HTTP-Zugriffe (einfacher Web-Server als Dienst)
 - ◆ I/O-Zugriffe (Einführung neuer Protokolle)
 - ◆ zentrale Dienstkonfiguration durch einen Administrator
 - ◆ ...

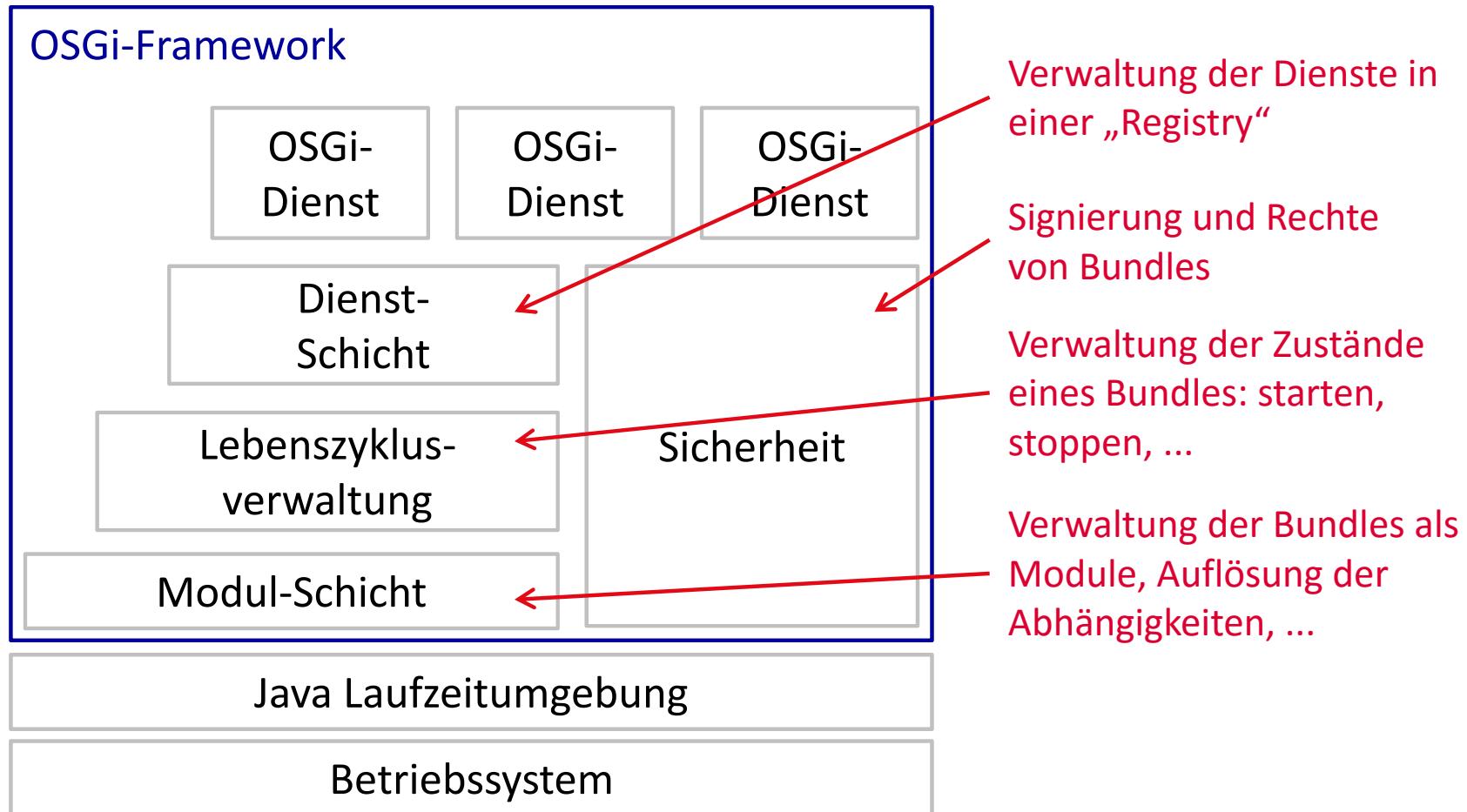


- Übersicht:



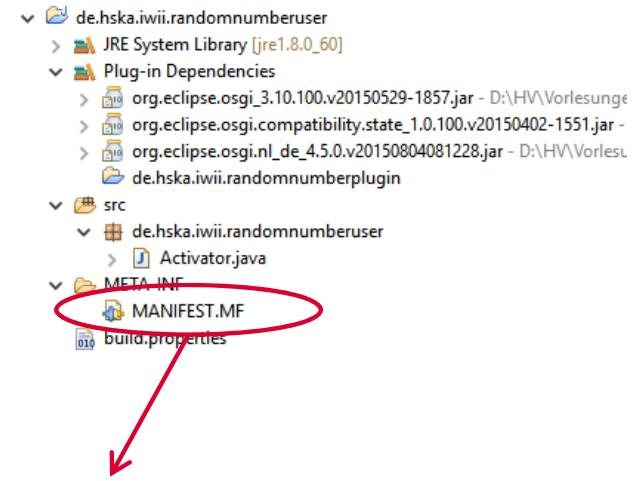


- Logische Schichten in OSGi:





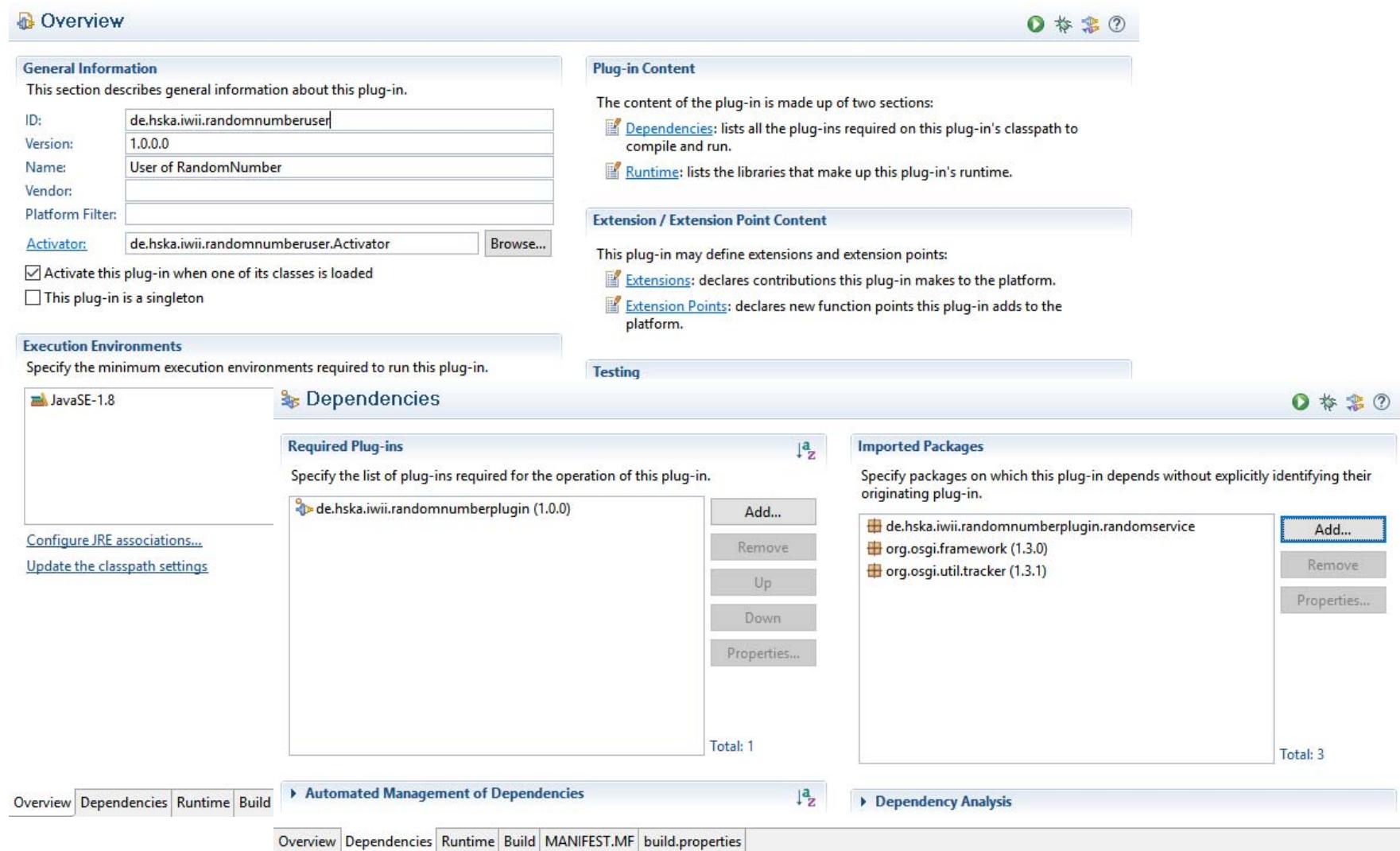
- Modul-Schicht:
 - ◆ Ein Bundle (Modul) ist eine JAR-Datei mit einer darin enthaltenen Manifest-Datei.
 - ◆ Die Manifest-Datei beschreibt das Modul:
 - Versionsnummer
 - eindeutiger symbolischer Modulname
 - beschreibender Modulname
 - exportierte Pakete
 - Abhängigkeiten von anderen Modulen
 - ...



```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: User of RandomNumber
Bundle-SymbolicName: de.haska.iwii.randomnumberuser
Bundle-Version: 1.0.0.0
Bundle-Activator: de.haska.iwii.randomnumberuser.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: de.haska.iwii.randomnumberplugin.randomservice,
    org.osgi.framework;version="1.3.0",
    org.osgi.util.tracker;version="1.3.1"
Require-Bundle: de.haska.iwii.randomnumberplugin;bundle-version="1.0.0"
```



- ◆ Eclipse bietet zur Manipulation der Manifest-Datei einen Editor:



The screenshot shows the Eclipse OSGi Configuration Editor interface. At the top, there's a toolbar with icons for Overview, Dependencies, Runtime, and Build. Below the toolbar, there are several tabs: General Information, Dependencies, Imported Packages, and so on.

General Information: This section contains fields for ID (de.hsk.a.iwii.randomnumberuser), Version (1.0.0.0), Name (User of RandomNumber), Vendor, Platform Filter, Activator (de.hsk.a.iwii.randomnumberuser.Activator), and checkboxes for "Activate this plug-in when one of its classes is loaded" and "This plug-in is a singleton".

Dependencies: This section lists required plug-ins under "Required Plug-ins". One entry is shown: de.hsk.a.iwii.randomnumberplugin (1.0.0). There are buttons for Add..., Remove, Up, Down, and Properties... A total count of 1 is displayed.

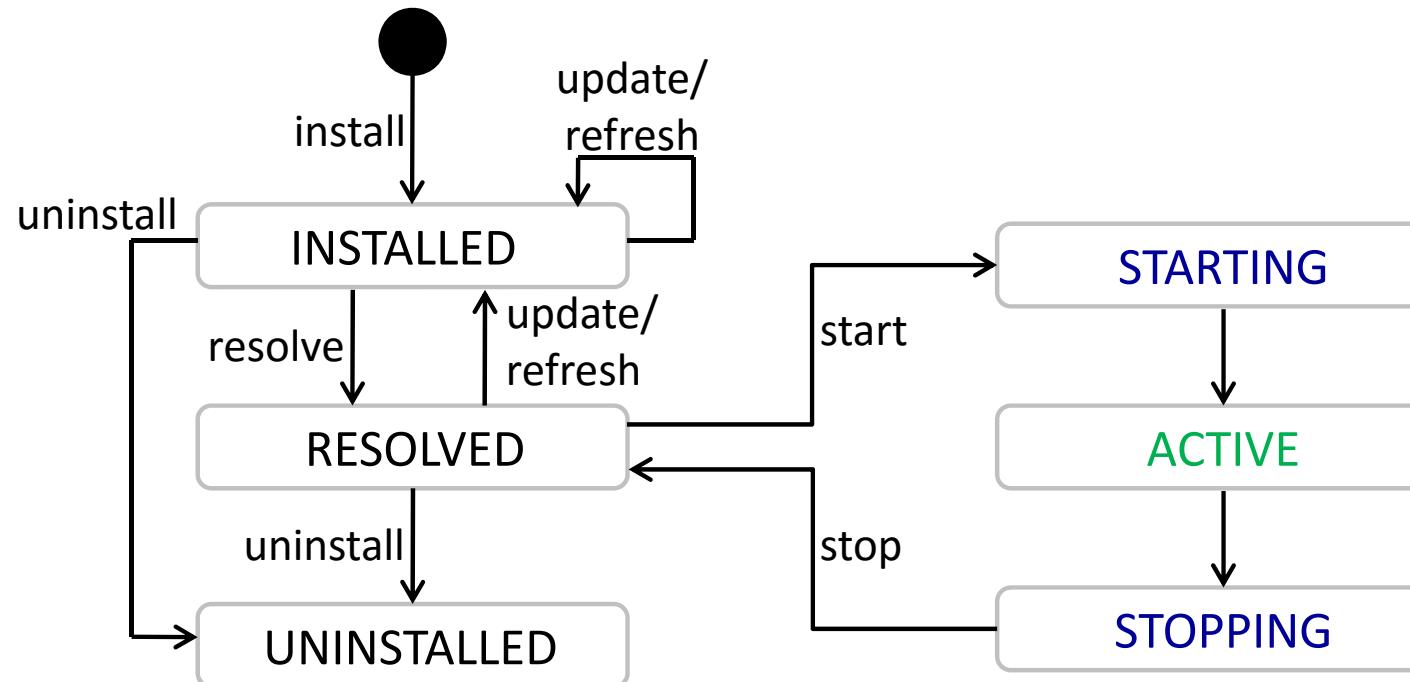
Imported Packages: This section lists imported packages under "Imported Packages". Three entries are shown: de.hsk.a.iwii.randomnumberplugin.randomservice, org.osgi.framework (1.3.0), and org.osgi.util.tracker (1.3.1). There are buttons for Add..., Remove, and Properties... A total count of 3 is displayed.

Automated Management of Dependencies: This tab is active at the bottom, showing the dependencies listed above.

Dependency Analysis: This tab is also present at the bottom.

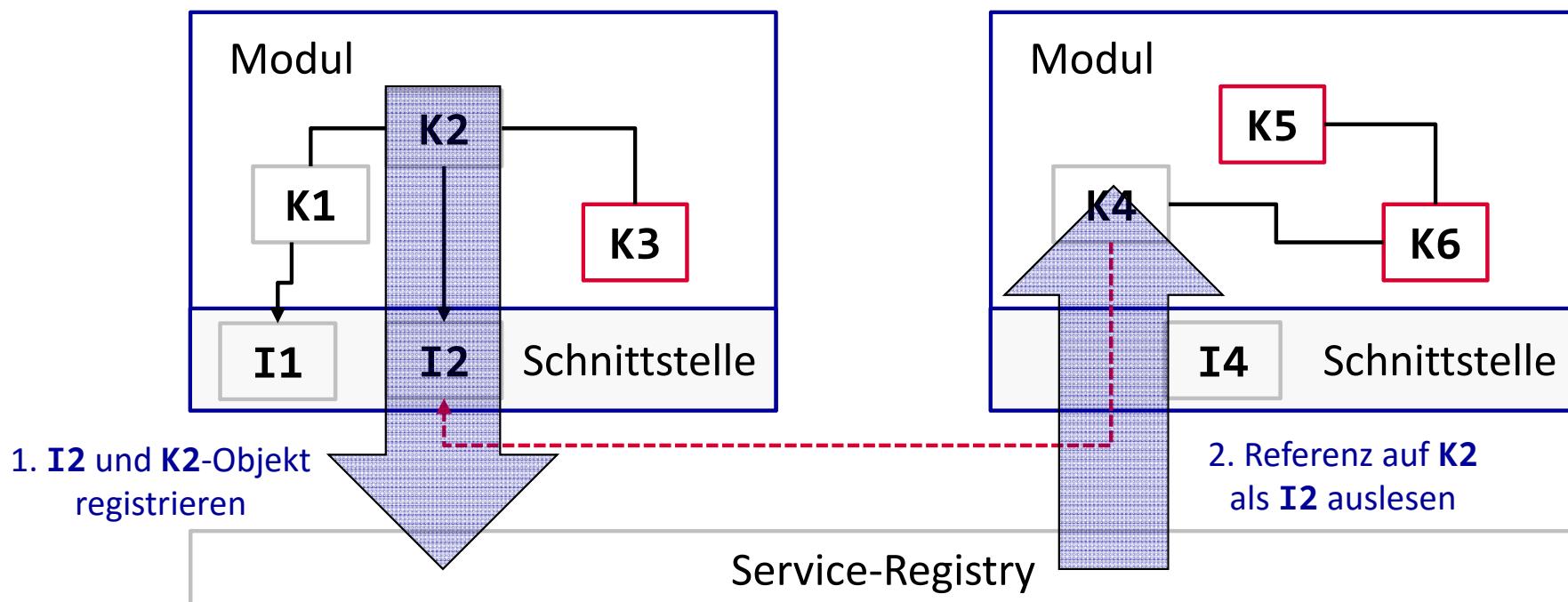


- Lebenszyklus-Verwaltung: Bundles lassen sich dynamisch starten und beenden.





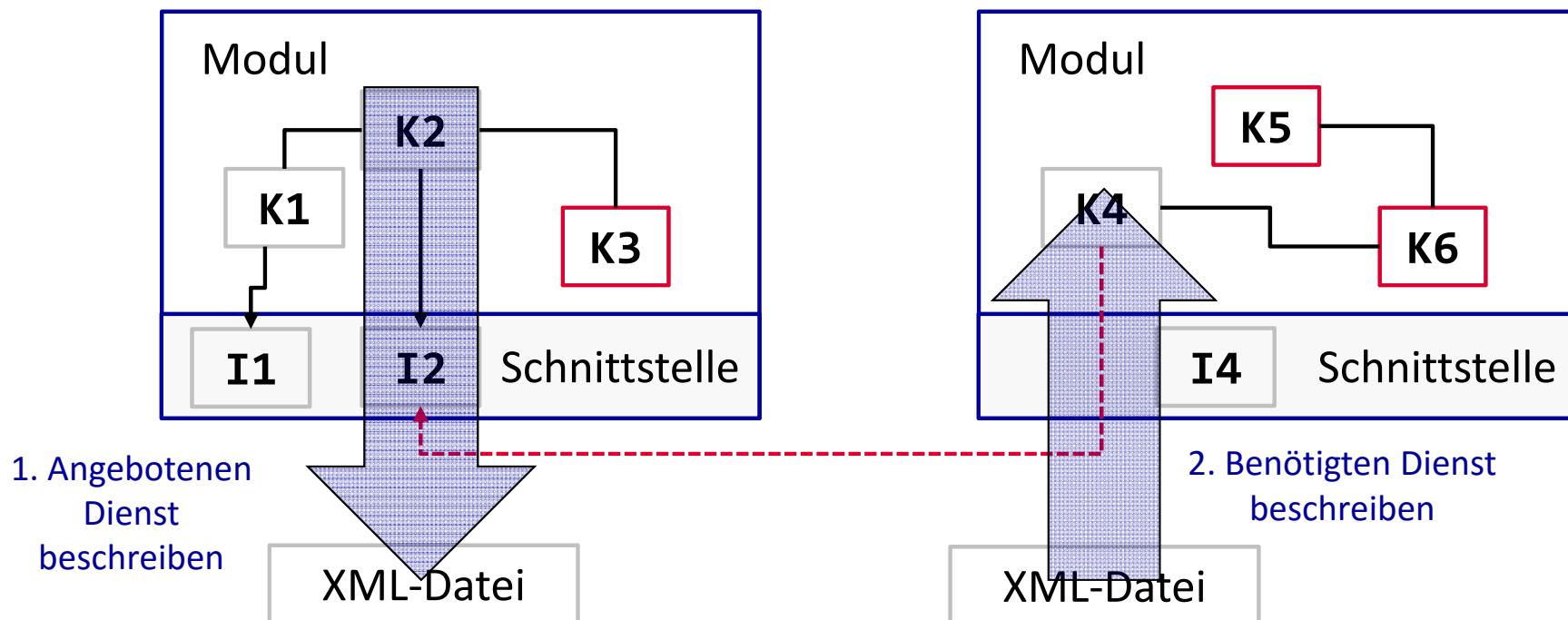
- Wie lassen sich die Dienste der Bundles verknüpfen?
 1. Jedes Bundle, das einen Dienst anbietet, registriert sich selbst in der zentralen Registry mit den folgenden Daten:
 - Schnittstelle, die der Dienst implementiert
 - Objekt, das den Dienst anbietet
 - Optional: **Dictionary** mit zusätzlichen Name/Wert-Paaren





2. Deklarativ durch „Dependency Injection“ als „Declarative Services“:

- Angebotene Dienste eines Bundles werden in XML-Dateien beschrieben.
- Jedes Bundle, das einen Dienst benötigt, beschreibt das auch in seiner XML-Datei.
- Die Beziehungen werden automatisch „injiziert“.

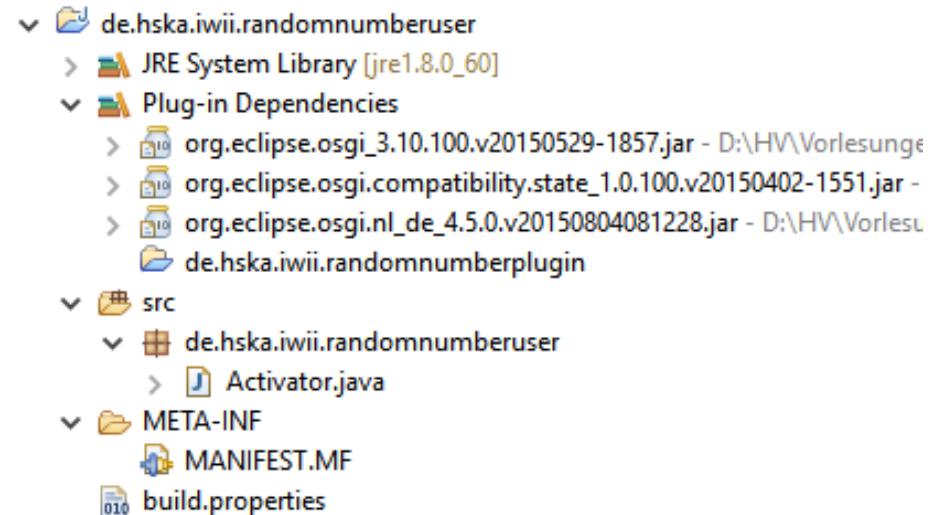
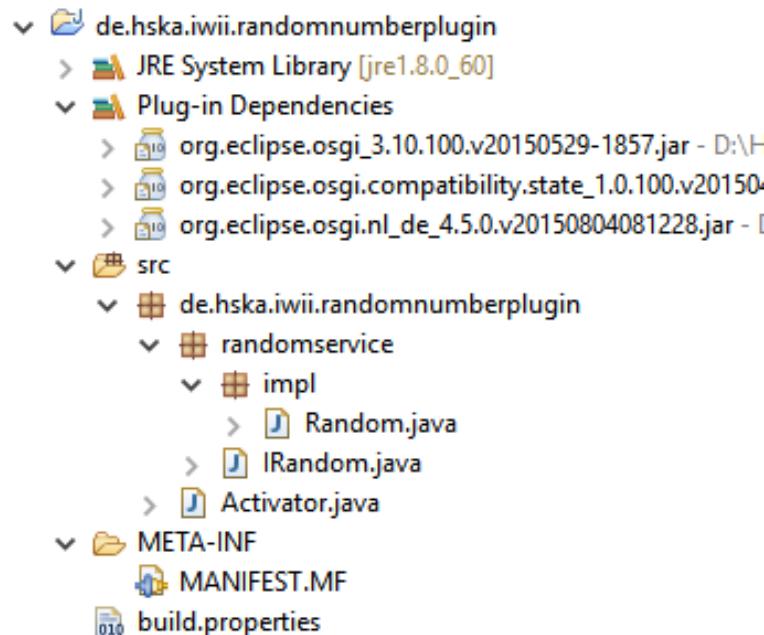




3. Deklarativ mit „Dependency Injection“ durch Gemini Blueprint → Soll hier nicht betrachtet werden.
4. Deklarativ mit „Dependency Injection“ durch Google Guice (<http://code.google.com/p/google-guice/>) und Peaberry (<http://code.google.com/p/peaberry/>) → Soll hier ebenfalls nicht betrachtet werden.
5. Weitere Frameworks für „Dependency Injection“



- Beispiel mit zwei Bundles:
 - ◆ Das Bundle **de.hska.iwii.randomnumberplugin** stellt einen Dienst zur Verfügung, der eine ganzzahlige Zufallszahl in einem Bereich zwischen zwei Grenzen liefert.
 - ◆ Das Bundle **de.hska.iwii.randomnumberuser** nutzt das erste Bundle und gibt die Zufallszahl aus.





- Klassen in **RandomNumberPlugin**:

- ◆ **IRandom**: Schnittstelle des Dienstes, wird exportiert

```
public interface IRandom {  
    int getNumber(int from, int to);  
}
```

- ◆ **Random**: Implementierung des Dienstes, nach außen nicht sichtbar

```
public class Random implements IRandom {  
    public int getNumber(int from, int to) {  
        return from + (int) ((to - from + 0.5) * Math.random());  
    }  
}
```



- ◆ **Activator:** Wird aufgerufen, wenn das Bundle gestartet oder beendet wird (in der Manifest-Datei konfiguriert), registriert den einzigen Dienst.

General Information
This section describes general information about this plug-in.

ID:	de.hska.iwii.randomnumberplugin
Version:	1.0.0.0
Name:	RandomNumberPlugin
Vendor:	
Platform Filter:	
Activator:	de.hska.iwii.randomnumberplugin.Activator

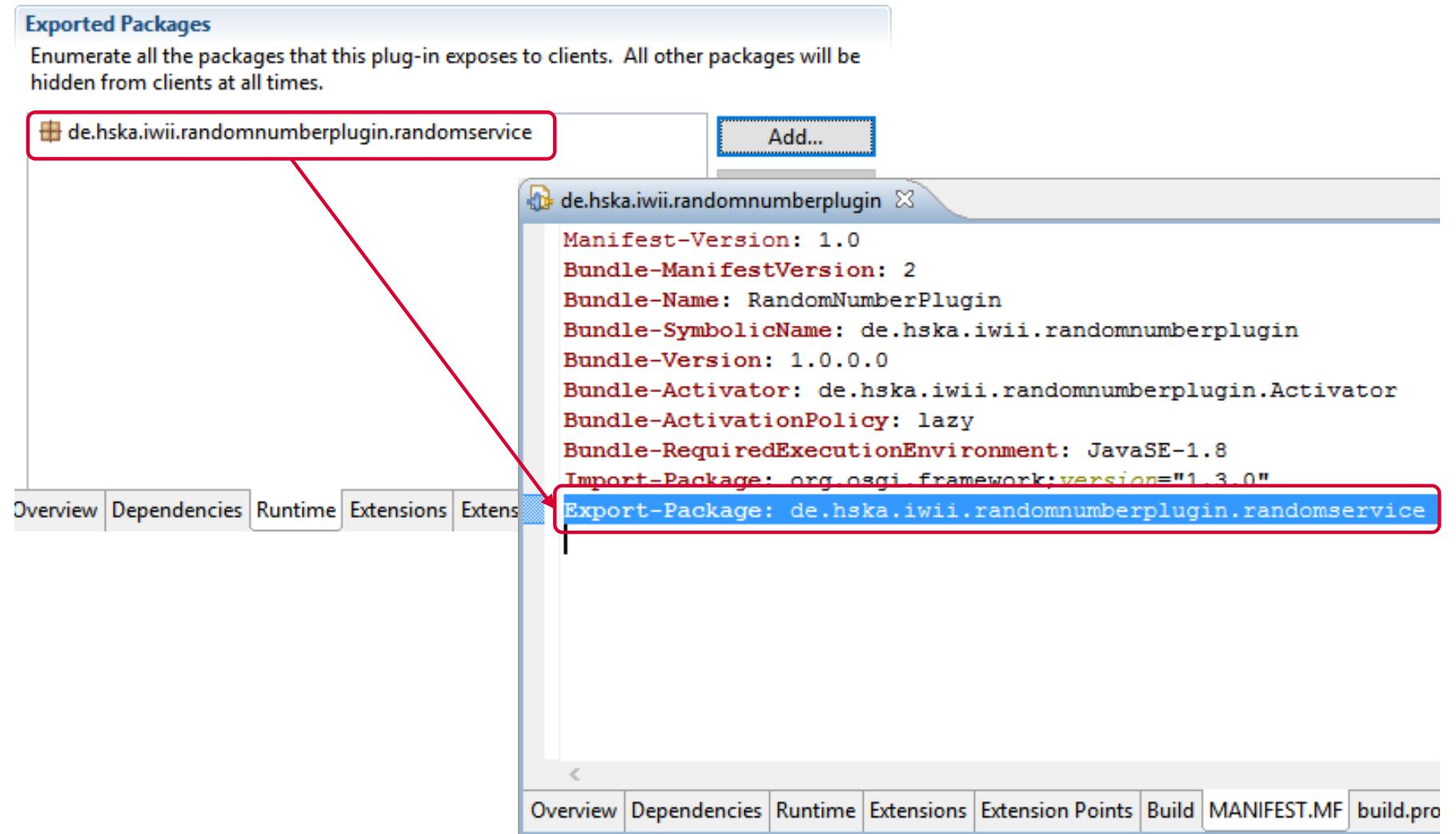
Activate this plug-in when one of its classes is loaded
 This plug-in is a singleton



```
public class Activator implements BundleActivator {  
    private ServiceRegistration<?> registration;  
  
    public void start(BundleContext context) throws Exception {  
        registration = context.registerService(IRandom.class.getName(),  
                                         new Random(), null);  
    }  
  
    public void stop(BundleContext context) throws Exception {  
        registration.unregister();  
    }  
}
```



- ◆ Export eines Paketes des Bundles, um es außerhalb nutzen zu können:





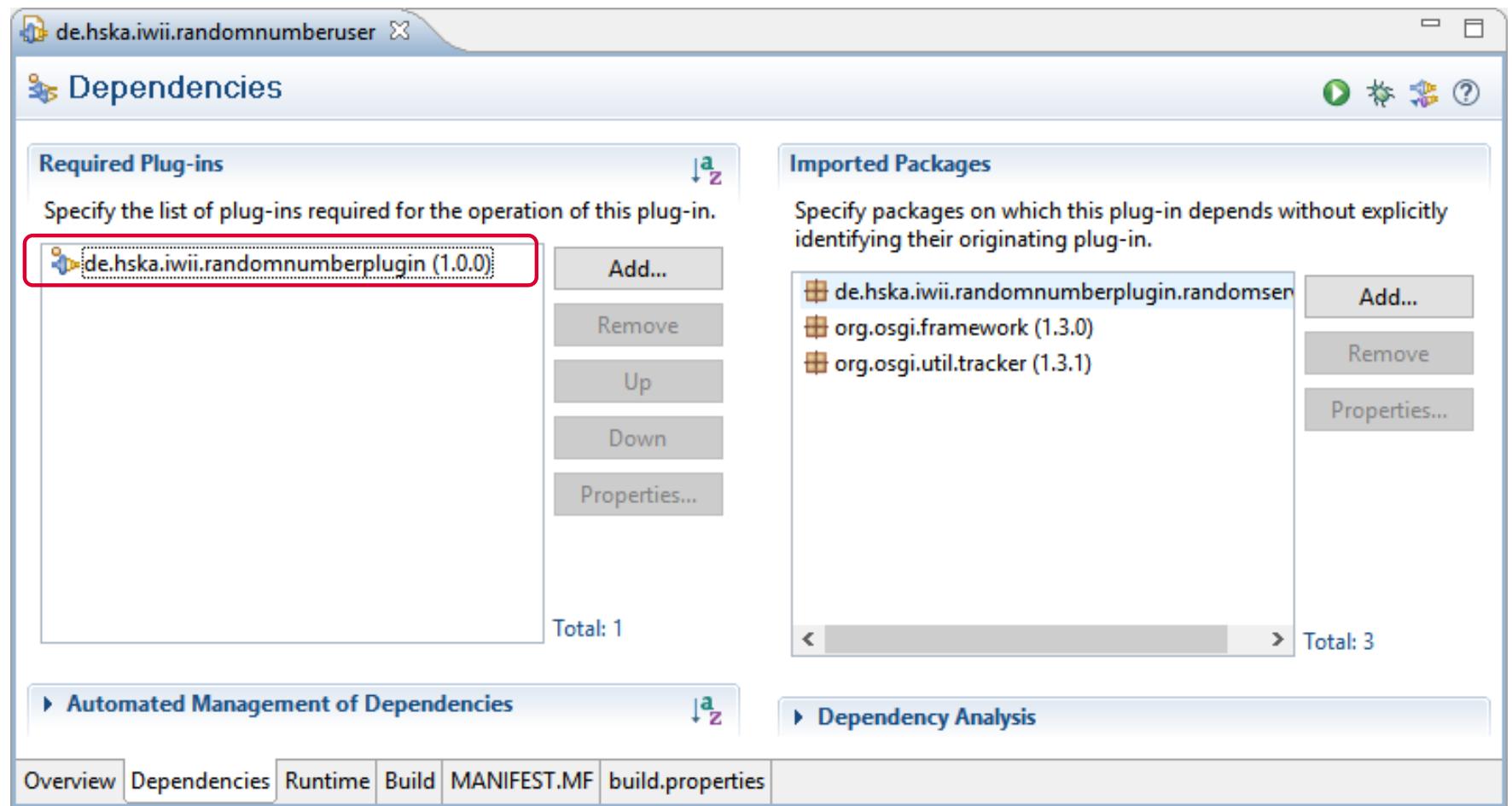
- In **RandomNumberUser**:

- ◆ **Activator**: Wird aufgerufen, wenn das Bundle gestartet oder beendet wird.
Beim Start wird das Bundle **RandomNumberPlugin** verwendet.

```
public class Activator implements BundleActivator {  
  
    public void start(BundleContext context) throws Exception {  
        System.out.println("RandomServiceUser started");  
        // Indirektion über ServiceReference, um auch die beim  
        // Registrieren übergebenen Properties auslesen zu können.  
        ServiceReference<?> ref =  
            context.getServiceReference(IRandom.class.getName());  
        // ref kann null sein --> Test hier ausgelassen  
        IRandom rand = (IRandom) context.getService(ref);  
        System.out.println(rand.getNumber(10, 20));  
    }  
  
    public void stop(BundleContext context) throws Exception {  
    }  
}
```



- ◆ Abhängigkeit von einem anderen Paket konfigurieren:





- ◆ Bzw. in der Manifest-Datei:

```
de.hska.iwii.randomnumberuser X
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: User of RandomNumber
Bundle-SymbolicName: de.hska.iwii.randomnumberuser
Bundle-Version: 1.0.0.0
Bundle-Activator: de.hska.iwii.randomnumberuser.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: de.hska.iwii.randomnumberplugin.randomservice,
    org.osgi.framework;version="1.3.0",
    org.osgi.util.tracker;version="1.3.1"
Require-Bundle: de.hska.iwii.randomnumberplugin;bundle-version="1.0.0"

Overview Dependencies Runtime Build MANIFEST.MF build.properties
```

- ◆ Alternativ: Statt Bundle-Abhängigkeit → Paket-Import (flexibler)



- Dieses Beispiel hat Nachteile:
 - ◆ Das **RandomNumberPlugin** lässt sich zur Laufzeit nicht austauschen (kein Update).
 - ◆ Bei komplexeren Bundles kann es vorkommen, dass diese nicht immer erreichbar sind (z.B. wenn sie eine funktionierende Netzwerkverbindung benötigen) → Das verwendete Bundle kann nicht deaktiviert werden.
- Lösung: Es gibt sogenannte „Service-Tracker“, die den Zustand von Diensten beobachten können:
 - ◆ Werden bei Aktivierungen und Deaktivierungen von Diensten benachrichtigt.
 - ◆ Sie erlauben die Angabe von Filtern, die die Benachrichtigungskriterien steuern (soll hier nicht betrachtet werden).



- Beispiel zur Benachrichtigung bei
 - ◆ Vorhandensein bzw. Abwesenheit des Dienstes
 - ◆ Aktivierung und Deaktivierung des Dienstes **IRandom**
- → Zustands- und ereignisbasierte Benachrichtigung

```
public class RandomServiceTracker extends ServiceTracker<IRandom, IRandom> {  
  
    public RandomServiceTracker(BundleContext context) {  
        super(context, IRandom.class.getName(), null);  
    }  
  
    // Dienst IRandom ist verfügbar.  
    public IRandom addingService(ServiceReference<IRandom> reference) {  
        IRandom service = super.addingService(reference);  
        // Mache etwas mit dem Service (z.B. intern merken)  
        return service;  
    }  
  
    // Rückgabetyp  
    // Service-Klasse  
    // kein Filter
```



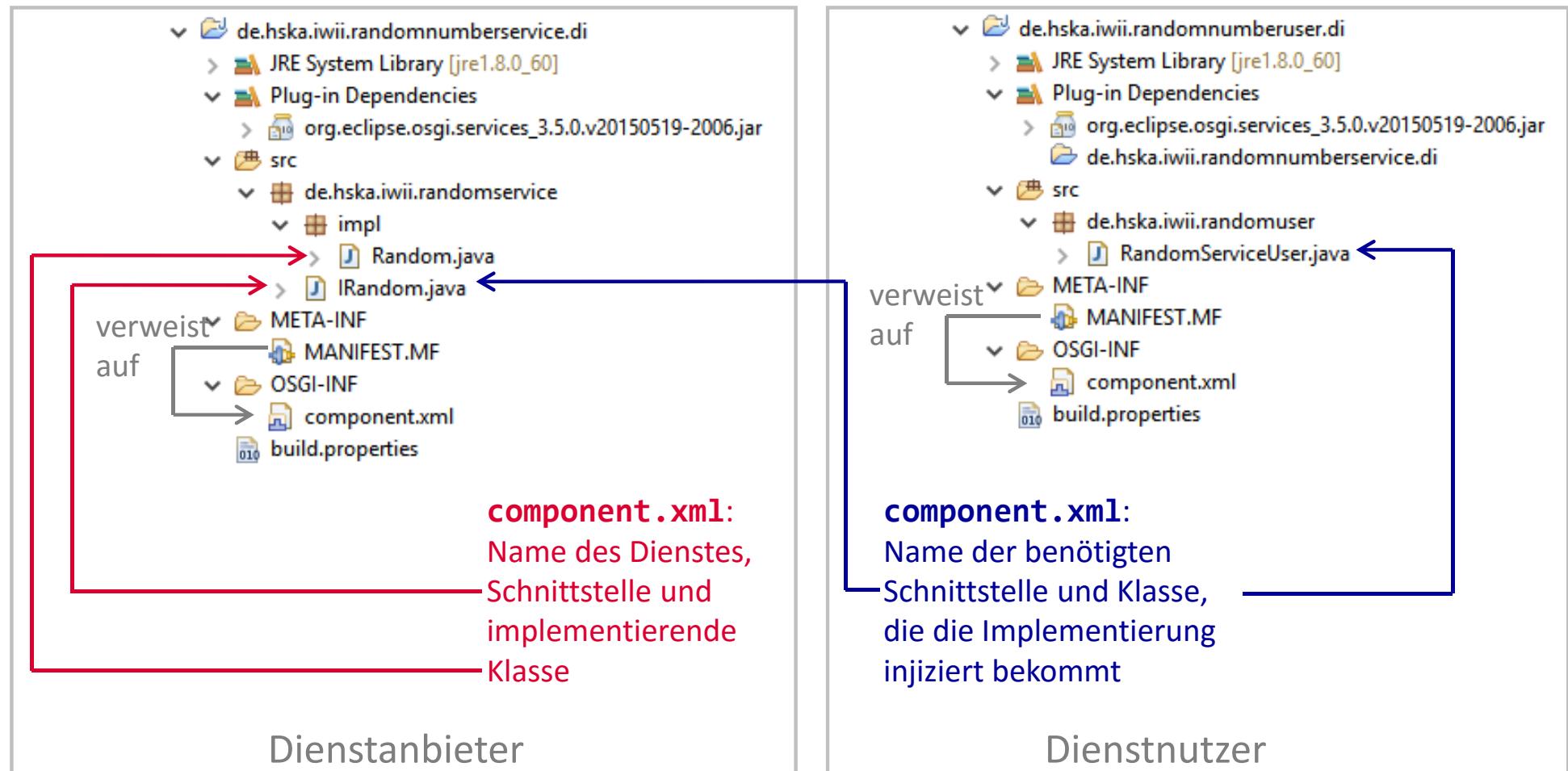
```
// Dienst IRandom ist nicht länger verfügbar.  
public void removedService(ServiceReference<IRandom> reference,  
                           IRandom service) {  
    // Mache etwas (z.B. interne Referenz löschen)  
    super.removedService(reference, service);  
}  
}
```



- Plug-ins können auch per „Dependency Injection“ verbunden werden → siehe unter Stichwort „OSGi Declarative Services“
 - ◆ Das Plug-in erhält seine Abhängigkeiten „von außen“ injiziert.
 - ◆ Zwei Bestandteile sind erforderlich:
 - Plug-in-Klasse(n), beschrieben durch eine Schnittstelle
 - XML-Konfiguration zur Beschreibung der Abhängigkeiten und exportierten Pakete
 - ◆ Die OSGi-Laufzeitumgebung ruft Setter-Methoden auf, um die Abhängigkeiten zu injizieren.
 - ◆ Die Namen der Setter-Methoden werden in der XML-Datei festgelegt.
 - ◆ Die zwei Methoden **activate** und **deactivate** werden bei der Aktivierung bzw. Deaktivierung des Plug-ins aufgerufen.
 - ◆ Das Plug-in wird normalerweise erst dann aktiv, wenn es benötigt wird.
 - ◆ Die XML-Datei kann Parameter an die **activate**-Methode übergeben.
- Umbau des Zufallszahlen-Beispiels auf den folgenden Seiten



Übersicht





- Schnittstelle, die den Service anbietet:

```
public interface IRandom {  
    int getNumber(int from, int to);  
}
```

- Klasse, die die Schnittstelle (= den Dienst) implementiert:

```
public class Random implements IRandom {  
    private String name;  
  
    protected void activate(ComponentContext componentContext) {  
        // Parameter "name" aus der XML-Datei auslesen  
        // (wird nicht verwendet -> nur kleine Demo)  
        name = (String) componentContext.getProperties().get("name");  
    }  
  
    protected void deactivate(ComponentContext componentContext) {  
    }  
  
    public int getNumber(int from, int to) {  
        return from + (int) ((to - from + 0.5) * Math.random());  
    }  
}
```



- Dienst in der Datei **OSGI-INF/component.xml** deklarieren:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="de.hska.iwii.randomnumberservice.di">
    <implementation class="de.hska.iwii.randomservice.impl.Random"/>
    <service>
        <provide interface="de.hska.iwii.randomservice.IRandom"/>
    </service>
    <property name="name" type="String" value="Zufallszahlen als OSGi-Dienst"/>
</scr:component>
```



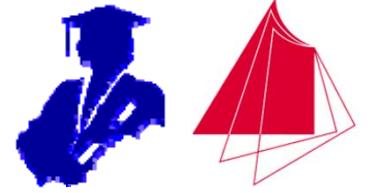
- Es fehlen noch der Verweis in der **Manifest.mf**-Datei auf die XML-Deklaration sowie der Export des Paketes, in dem die Schnittstelle liegt:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: RandomNumberService
Bundle-SymbolicName: de.hska.iwii.randomnumberservice.di
Bundle-Version: 1.0.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: org.osgi.service.component;version="1.1.0"
Export-Package: de.hska.iwii.randomservice
Service-Component: OSGI-INF/component.xml
```



- Klasse, die den Dienst nutzen möchte:

```
public class RandomServiceUser {  
    private IRandom randomNumberService;  
  
    protected void activate(ComponentContext componentContext) {  
        System.out.println(randomNumberService.getNumber(10, 20));  
    }  
  
    protected void deactivate(ComponentContext componentContext) {  
        this.randomNumberService = null;  
    }  
  
    // Die Referenz auf den Dienst wird automatisch injiziert.  
    protected void setRandomNumberService(IRandom randomNumberService) {  
        this.randomNumberService = randomNumberService;  
    }  
  
    // Der Dienst wurde entfernt oder gestoppt: Referenz darauf löschen  
    protected void unsetRandomNumberService(IRandom randomNumberService) {  
        this.randomNumberService = null;  
    }  
}
```

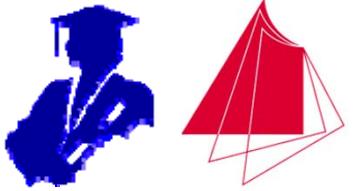


- XML-Deklaration der Nutzung:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="de.hska.iwii.randomnumberuser.di">
    <!-- Klasse des Dienstnutzers -->
    <implementation class="de.hska.iwii.randomuser.RandomServiceUser"/>

    <!-- Zu injizierender Dienst -->
    <reference
        <!-- Name der Methode, über die injiziert wird -->
        bind="setRandomNumberService"
        <!-- Anzahl zu injizierender Dienste (genau einer) -->
        cardinality="1..1"
        <!-- Schnittstelle, die der Dienst implementieren muss -->
        interface="de.hska.iwii.randomservice.IRandom"
        <!-- Name der Bindung, erfolgt statisch (unveränderlich) -->
        name="randomNumberService" policy="static"
        <!-- Name der Methode, über die die Bindung gelöst wird -->
        unbind="unsetRandomNumberService"/>
    </scr:component>
```

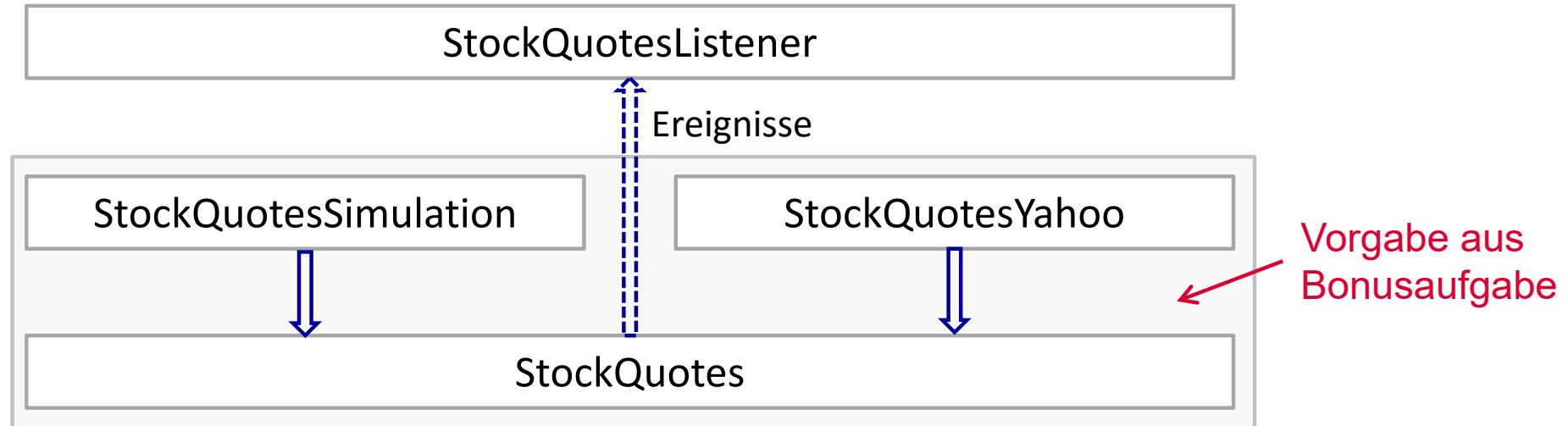
- Für dynamisch austauschbare Dienste: **policy="dynamic"** (sonst wird das nutzende Plug-in neu gestartet)



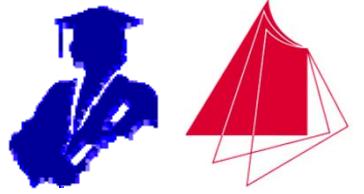
- XML-Hölle? Ab OSGi-Spezifikation R6 auch mit Annotationen möglich → soll hier entfallen, scheint unter Equinox auch nicht zu funktionieren (??).
- Beispiel angelehnt an die Bonusaufgabe:
 - ◆ Abholen von Börsenkursen aus verschiedenen Quellen:
 - Yahoo-Server
 - Simulationsdaten
 - usw. möglich (aber hier nicht verwendet)
 - ◆ Die Daten werden zyklisch abgeholt → Thread oder Zeitsteuerung.
 - ◆ Es gibt Nutzer, die die Daten benötigen:
 - Anzeige in einer Tabelle
 - Ablage in einer Datenbank
 - Statistiken daraus berechnen



- Ansatz für eine gut erweiterbare Plug-in-Struktur mit Diensten (leichter Overkill für das kleine Beispiel):



- **StockQuotes**: allgemeine Dienste, zeitgesteuertes Abholen
- **StockQuotesSimulation**: Erzeugen von Simulationsdaten
- **StockQuotesYahoo**: Abholen echter Daten vom Server
- **StockQuotesListener**: Empfang der von **StockQuotes** bereitgestellten Daten als Ereignisse
→ Einsatz des Ereignisdienstes von OSGi (Event-Admin-Service)



- Event-Admin-Service:
 - ◆ Allgemeiner Dienst für den Versand und Empfang von Ereignissen:
 - synchroner Versand: innerhalb des Threads des Senders
 - asynchroner Versand: innerhalb eines eigenen Threads des Event-Admin-Services
 - ◆ Ereignisse werden anhand eines sogenannten Event-Topics (eindeutiger String) identifiziert. Der String hat den Namensaufbau eines Java-Paketes:
"de/hska/iwii/stockquotes/Fetch"
 - ◆ Ereignisse können zusätzliche Schlüssel-/ Wertepaare als Eigenschaften besitzen.
- Erforderliche Plug-ins:
 - ◆ **org.osgi.service.event**
 - ◆ **org.osgi.framework**
- Der Event-Admin-Service wird über „Dependency Injection“ dem Sender injiziert.
- Der Ereignisempfänger implementiert die Schnittstelle **EventHandler**.



- Sender des Ereignisses, am vereinfachten Beispiel (Event-Admin wird per „Dependency Injection“ eingefügt → siehe XML-Dokument):

```
public class StockQuotesProvider extends TimerTask {  
    private EventAdmin eventAdmin;  
    private Timer timer;  
  
    // Beim Start des Plug-ins den Timer starten (Auslösung jede Sekunde)  
    protected void activate(ComponentContext componentContext) {  
        timer = new Timer();  
        timer.scheduleAtFixedRate(this, 0, 1000);  
    }  
  
    // Beim Deaktivieren des Plug-ins den Timer anhalten.  
    protected void deactivate(ComponentContext componentContext) {  
        timer.cancel();  
    }  
  
    // Event-Admin-Service per Dependency Injection einfügen  
    public void setEventAdmin(EventAdmin eventAdmin) {  
        this.eventAdmin = eventAdmin;  
    }  
}
```



```
// Event-Admin-Service wurde gestoppt
public void unsetEventAdmin(EventAdmin eventAdmin) {
    this.eventAdmin = null;
}

// Diese Methode wird jede Sekunde vom Timer aufgerufen
public void run() {
    if (eventAdmin != null) {
        // Parameter, die dem Ereignis mitgegeben werden (hier nur die
        // aktuelle Uhrzeit) -> Schlüssel-/Wertepaare
        Dictionary<String, Object> properties =
            new Hashtable<String, Object>();
        properties.put("Time", System.currentTimeMillis());
        // Ereignisobjekt mit Topic und Parametern erzeugen
        Event event = new Event("de/hska/iwii/stockquotes/Fetch",
                               properties);
        // Ereignis asynchron senden
        eventAdmin.postEvent(event);
    }
}
```



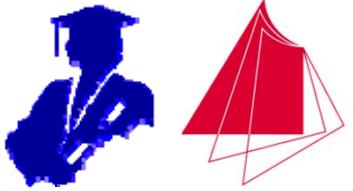
- Versandarten:
 - ◆ asynchron: **eventAdmin.postEvent(event)**
 - ◆ synchron: **eventAdmin.sendEvent(event)**
- Die Ereignis-Parameter sollten unveränderlich sein!
- In der Datei **component.xml** wird deklariert, dass der Event-Admin-Service injiziert werden soll:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
                name="de.hska.iwii.stockquotes.net.provider">
    <!-- Klasse, die den Event admin Service injiziert bekommen möchte -->
    <implementation class="de.hska.iwii.stockquotes.net.StockQuotesProvider"/>
    <!-- Bindung auf den Event-Admin-Service -->
    <reference bind="setEventAdmin" cardinality="1..1"
               interface="org.osgi.service.event.EventAdmin"
               name="EventAdmin" policy="static"/>
</scr:component>
```



- Empfang von Nachrichten:
 - ◆ Implementierung der Schnittstelle **EventHandler**
 - ◆ Implementierung der Methode **handleEvent**
- Vereinfachtes Beispiel:

```
public class StockQuotesReceiver implements EventHandler {  
    // Angebotener Service: Behandlung von Events an diesen Provider.  
    @Override  
    public void handleEvent(Event event) {  
        System.out.println("Event received: " + event.getTopic()  
                           + ": " + (new Date((Long) event.getProperty("Time"))));  
    }  
}
```



- In der Datei **component.xml** wird deklariert, dass die Klasse einen Event-Handler-Service anbietet:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
                 name="de.hska.iwii.stockquotesreceiver">
    <implementation
        class="de.hska.iwii.stockquotesreceiver.StockQuotesListener"/>
    <!-- Ereignisse, die erwartet werden -->
    <property name="event.topics" type="String"
              value="de/hska/iwii/stockquotes/Fetch"/>
    <!-- Angebotener Dienst, der per Dependency Injection dem
        Event Admin Service injiziert wird -->
    <service>
        <provide interface="org.osgi.service.event.EventHandler"/>
    </service>
</scr:component>
```

- Erwartete Ereignisse (Topics) können auch mit einem Muster angegeben werden.
Beispiel:
"de/hska/iwii/*"



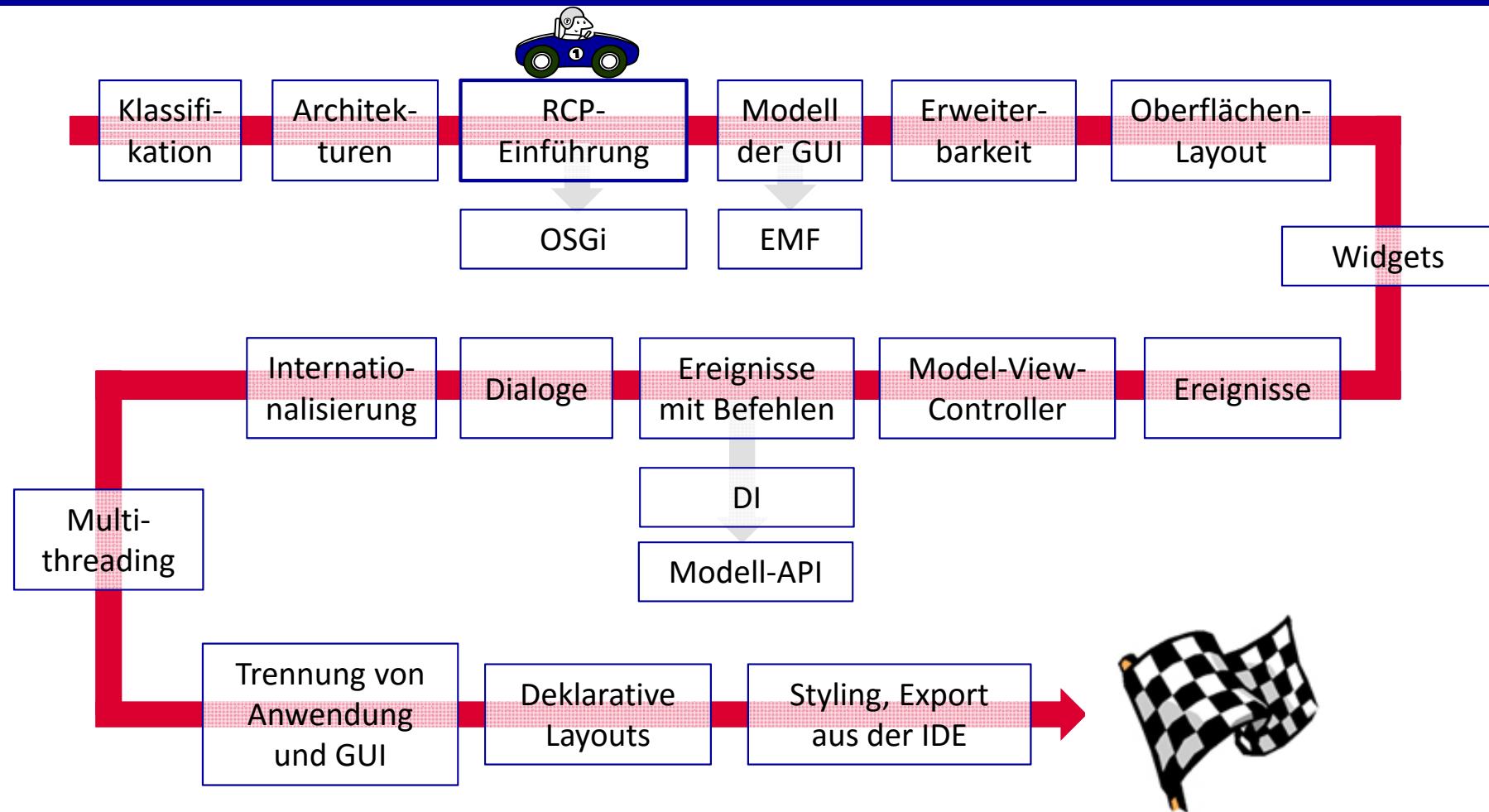
- Auch andere Dienste wie der Log-Service verwendet intern den Event-Admin-Service:
 - ◆ Der Erzeuger einer Log-Meldung sendet diese an den Log-Service.
 - ◆ Der Log-Service speichert die Nachrichten intern.
 - ◆ Alle registrierten Log-Listener werden bei Meldungen informiert.
- Hinweise zur Startkonfiguration unter Eclipse: Aus der Target-Plattform werden die folgenden Bundles benötigt (siehe Folgeseite):
 - ◆ **javax.servlet**
 - ◆ **org.eclipse.equinox.common**
 - ◆ **org.eclipse.equinox.ds**
 - ◆ **org.eclipse.equinox.event**
 - ◆ **org.eclipse.equinox.util**
 - ◆ **org.eclipse.osgi**
 - ◆ **org.eclipse.osgi.services**



- Wie sieht es mit den Modulabhängigkeiten aus?
 - ◆ Direkt über die Registry: Der Abhängigkeitsgraph der Module bestimmt deren Startreihenfolge → extrem aufwändig bei großen Projekten.
 - ◆ Beobachter an der Registry: Mühsame Beobachtung der Bundle- oder Dienstzustände (**context.addBundleListener**, **context.addServiceListener**, **ServiceTracker**).
 - ◆ Mittels „Dependency Injection“: Der Abhängigkeitsgraph wird automatisch von OSGi verwaltet.
 - ◆ Die Module kennen sich gar nicht und versenden lediglich Nachrichten (Event-Admin-Service).
- Seit OSGi 4.2 gibt es auch ein „Distributed OSGi“ für verteilte Anwendungen → sinnvoll, wenn auf dem Server auch OSGi läuft.
- Ein Eclipse-Plug-in ist ein OSGi-Bundle.

Die Eclipse Rich Client Platform

Plug-ins und Features





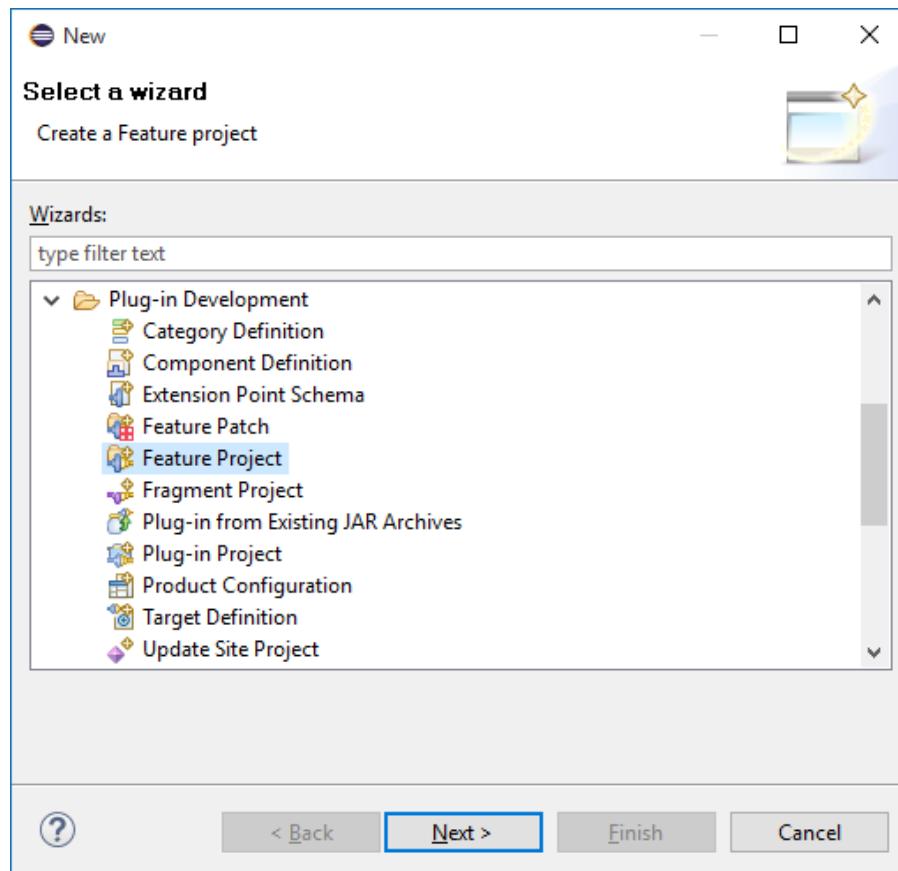
- Eclipse Plug-ins sind OSGi-Bundles.
- Wie sieht es aus, wenn „jemand etwas verwenden soll“, das aus vielen Plug-ins besteht?
 - ◆ Muss z.B. ein UML-Editor, der intern sauber durch Plug-ins strukturiert wurde, als lose Plug-in-Sammlung weitergegeben werden?
 - ◆ Wie können also mehrere Plug-ins zusammen eine Einheit bilden (etwas, was zusammen verwendet wird)?
 - ◆ Dafür gibt es sogenannte „Features“ → kommt gleich
- Was passiert, wenn in einem Feature ein Plug-in Plattformabhängigkeiten besitzt, die anderen Plug-ins aber plattformunabhängig sind?
 - ◆ Ein „Fragment“ ist ein Plug-in, was nicht eigenständig existiert, sondern ein anderes Plug-in erweitert.
 - ◆ Ziel: Der plattformabhängige Code gehört in einzelne Fragmente, von denen beim Programmstart automatisch das richtige ausgewählt und zum „Vater-Plug-in“ hinzugefügt wird.

Die Eclipse Rich Client Platform

Plug-ins und Features



- Erzeugen eines Feature-Projekts:

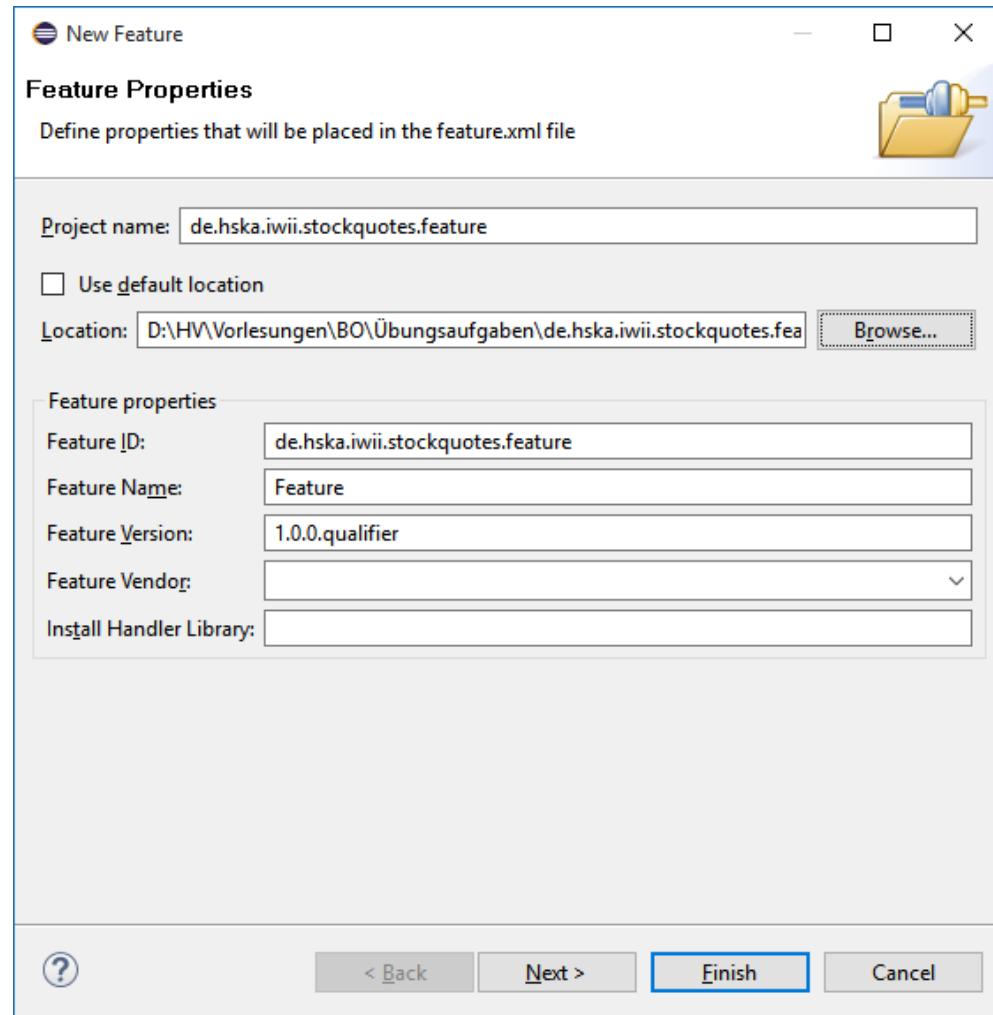


Die Eclipse Rich Client Platform

Plug-ins und Features



- Eintragen des Projektnamens:

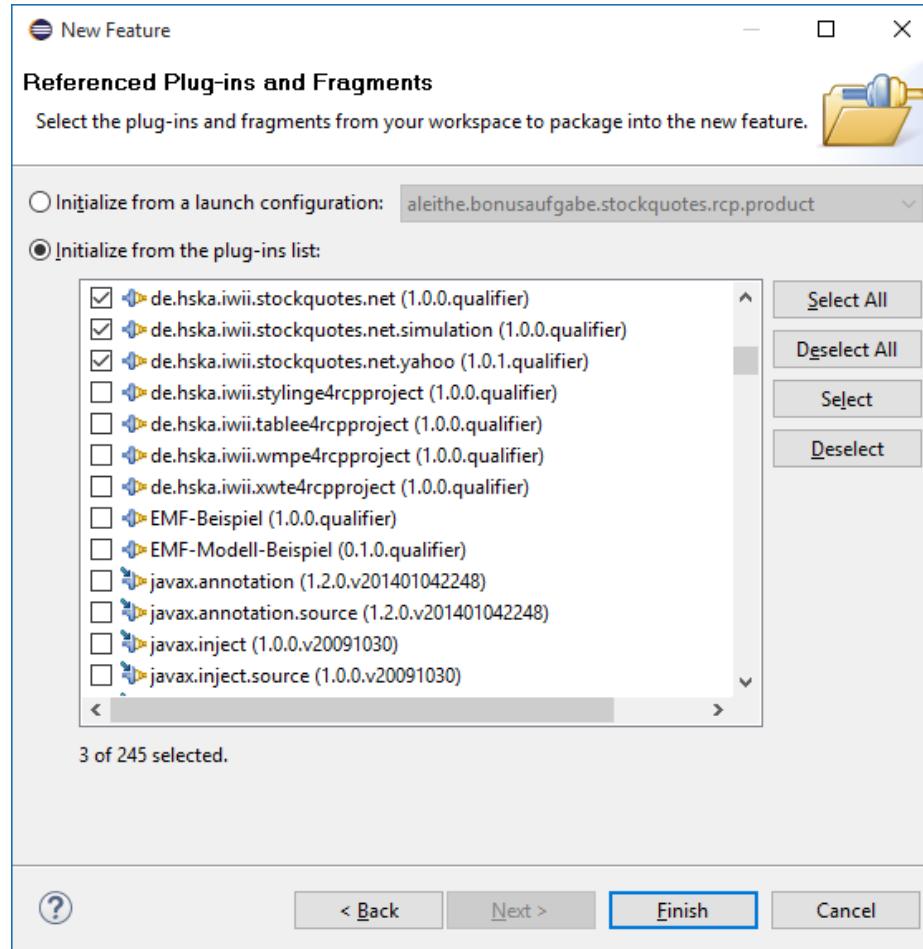


Die Eclipse Rich Client Platform

Plug-ins und Features



- Hinzufügen der Plug-ins, die zum Feature gehören sollen:



Die Eclipse Rich Client Platform

Plug-ins und Features



- Erzeugen des Features für eine spätere Verteilung:

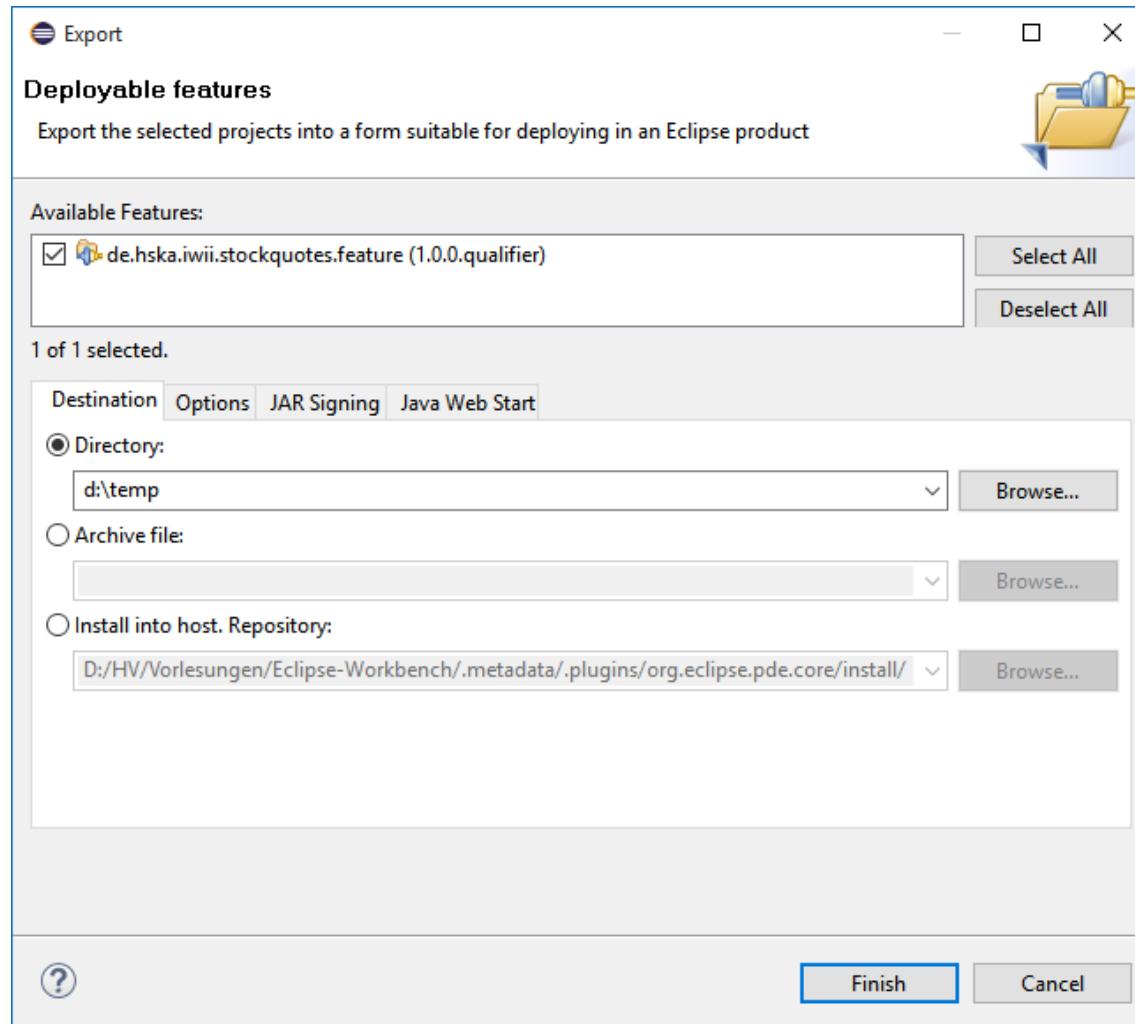
The screenshot shows the Eclipse Feature Editor interface for the feature 'de.hska.iwii.stockquotes.feature'. The window title is 'de.hska.iwii.stockquotes.feature'. The left pane contains sections for 'General Information' (with fields for ID, Version, Name, Vendor, Branding Plug-in, Update Site URL, and Update Site Name), 'Supported Environments' (with fields for Operating Systems, Window Systems, Languages, and Architecture), and tabs for 'Overview', 'Information', 'Plug-ins', 'Included Features', 'Dependencies', 'Build', 'feature.xml', and 'build.properties'. The right pane is titled 'Feature Content' and includes sections for 'Information', 'Plug-ins', 'Included Features', 'Dependencies', 'Exporting', and 'Publishing'. A red circle highlights the 'Plug-ins' icon in the top right corner of the editor window.

Die Eclipse Rich Client Platform

Plug-ins und Features



- Export starten:



Die Eclipse Rich Client Platform

Plug-ins und Features



■ Ergebnis:

Name	Datum	Typ	Größe	Markierungen
features	02.10.2015 08:10	Dateiordner		
plugins	02.10.2015 08:10	Dateiordner		
artifacts.jar	02.10.2015 08:10	Executable Jar File	1 KB	
content.jar	02.10.2015 08:10	Executable Jar File	2 KB	

Die Eclipse Rich Client Platform

Plug-ins und Features



- Erstellen eines Update-Site-Projekts, mit dessen Hilfe das Feature auf einem Server abgelegt und von dort installiert werden kann:

The screenshot shows the Eclipse Feature Editor interface for the feature 'de.hska.iwii.stockquotes.feature'. The window title is 'de.hska.iwii.stockquotes.feature X'. The left sidebar shows the feature name 'Stockquotes'. The main content area is divided into several sections:

- General Information:** Describes the feature's ID (de.hska.iwii.stockquotes.feature), Version (1.0.0.qualifier), Name (Stockquotes), Vendor (empty), Branding Plug-in (empty), Update Site URL (empty), and Update Site Name (empty).
- Supported Environments:** Allows specifying supported environments for installation.
- Feature Content:** Lists five sections:
 - Information:** Holds general information about the feature.
 - Plug-ins:** Lists the plug-ins that make up the feature.
 - Included Features:** Lists features included in this feature.
 - Dependencies:** Lists other features and plug-ins required by this feature when installed.
- Exporting:** Instructions for exporting the feature.
- Publishing:** Instructions for publishing the feature on an update site, which is circled in red.

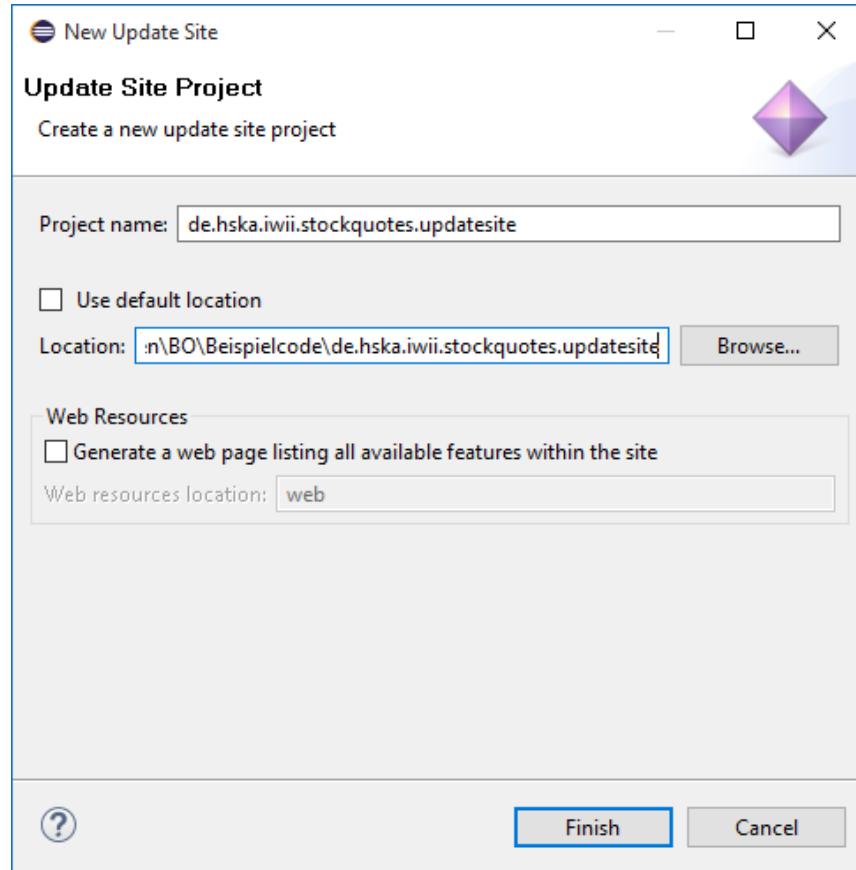
At the bottom, there is a navigation bar with tabs: Overview, Information, Plug-ins, Included Features, Dependencies, Build, feature.xml, and build.properties. The 'Information' tab is currently selected.

Die Eclipse Rich Client Platform

Plug-ins und Features



- Eintragen des Projektnamens:

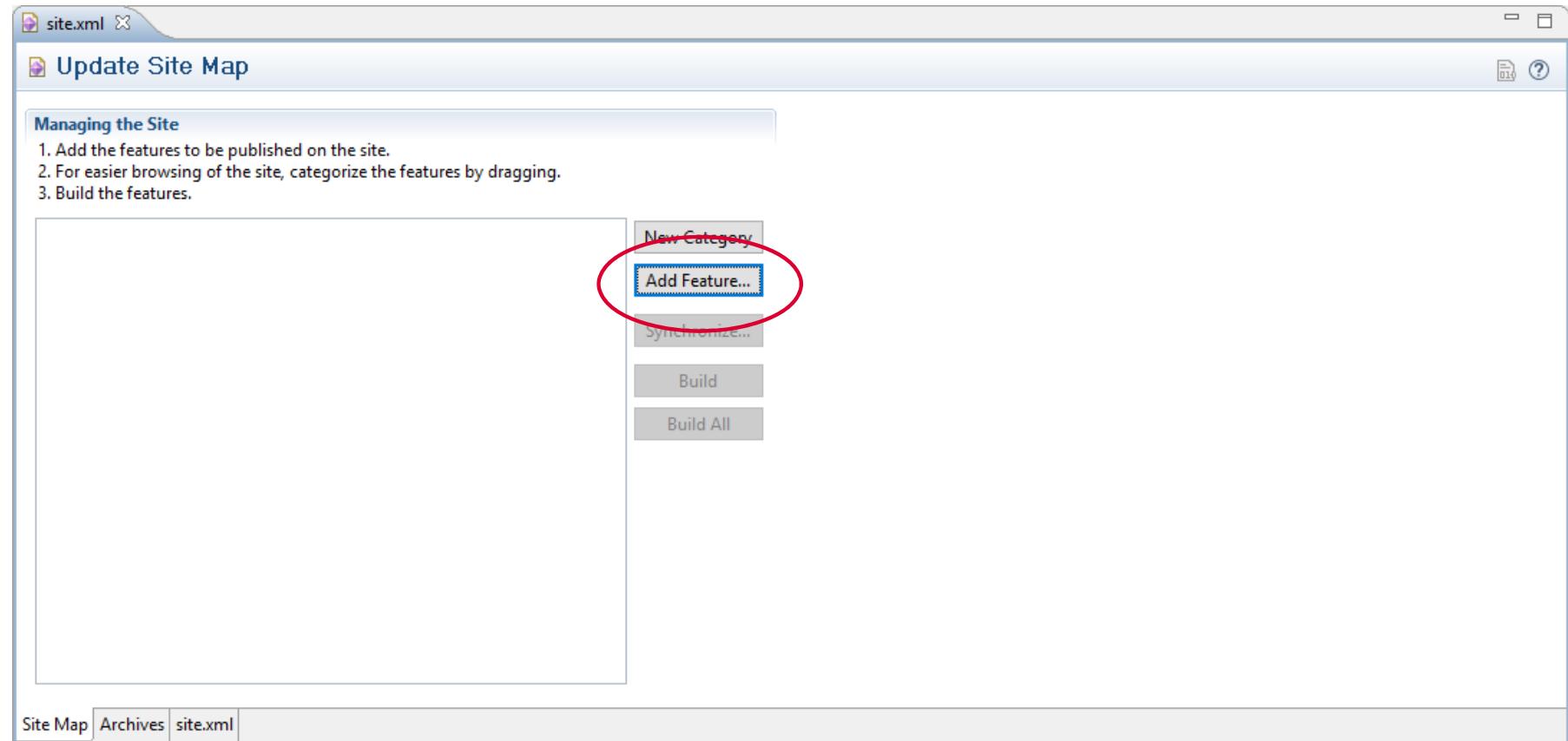


Die Eclipse Rich Client Platform

Plug-ins und Features



- Ansicht der Site-Beschreibung:

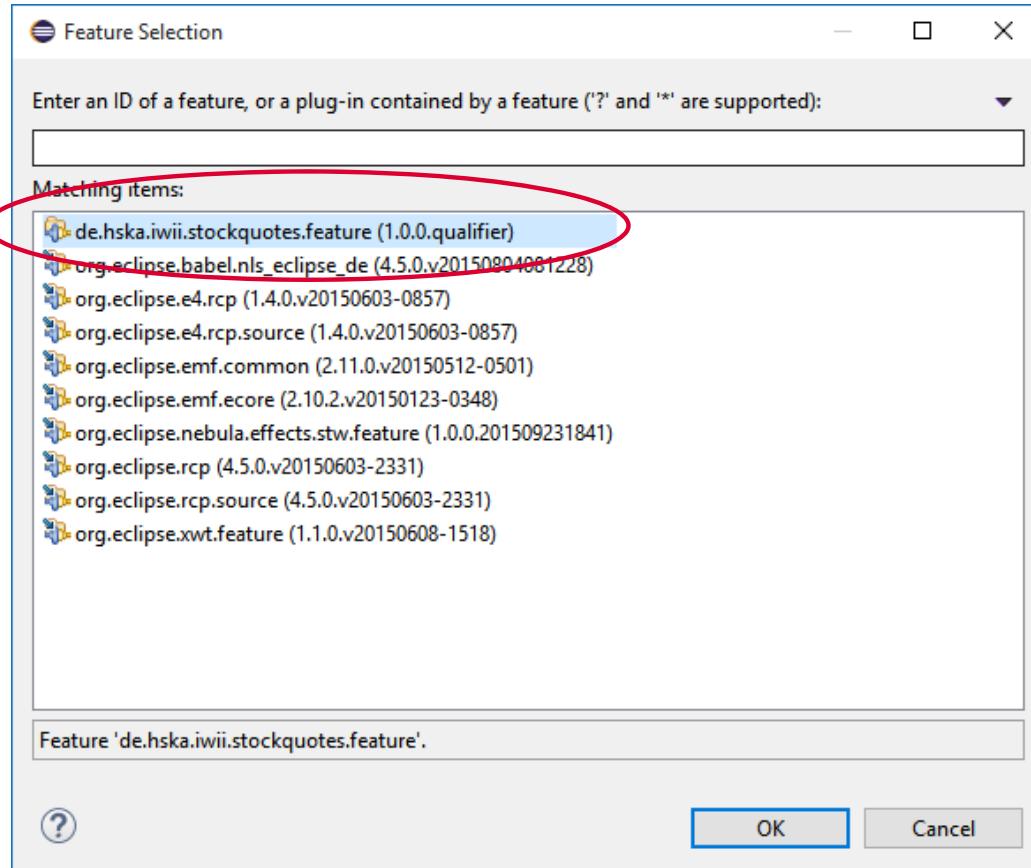


Die Eclipse Rich Client Platform

Plug-ins und Features



- Feature zur Site-Beschreibung hinzufügen:

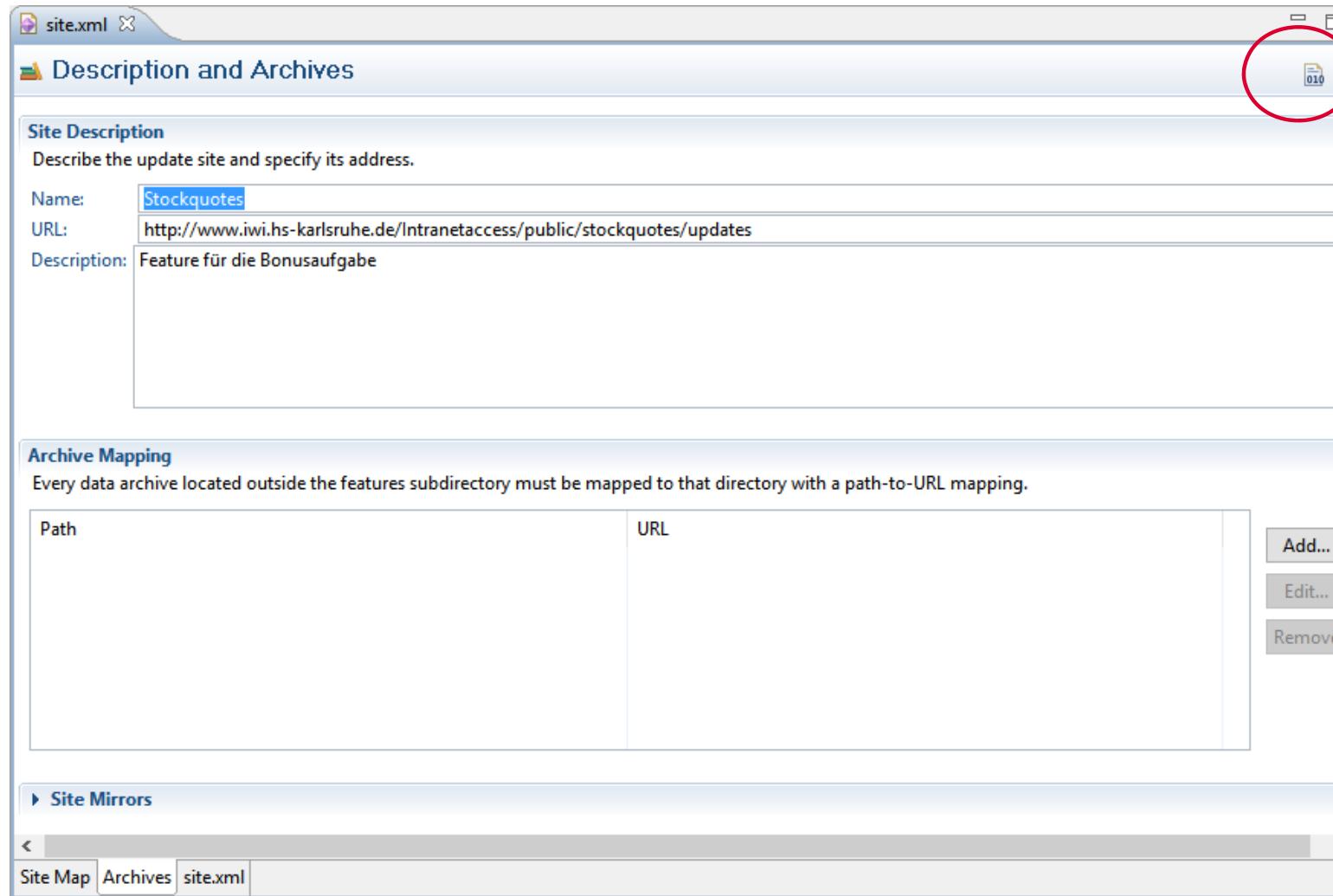


Die Eclipse Rich Client Platform

Plug-ins und Features



- URL und Namen eintragen und bauen:



bauen



- Jetzt können Sie die Plug-ins für die Börsenaufgabe direkt von dieser Update-Site in die Target-Platform übernehmen (ist bei der Target-Platform im Ilias bereits integriert):

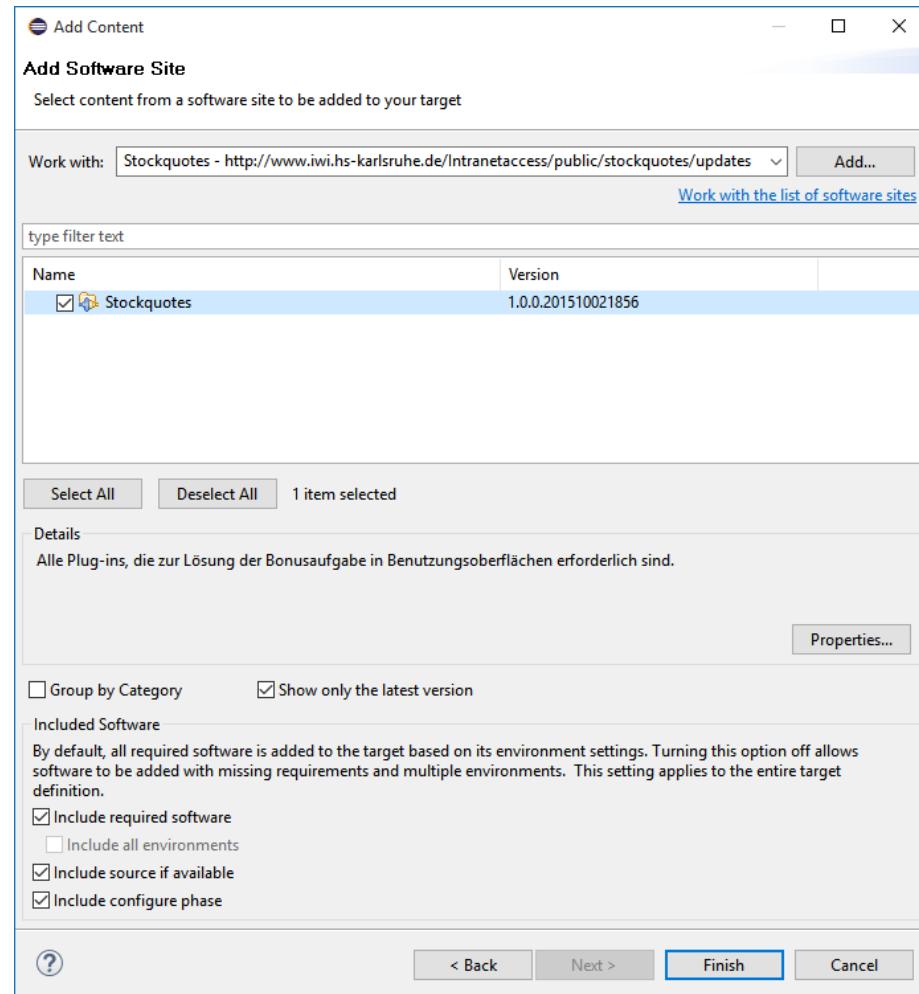
<http://www.iwi.hs-karlsruhe.de/Intranetaccess/public/stockquotes/updates>

Die Eclipse Rich Client Platform

Plug-ins und Features

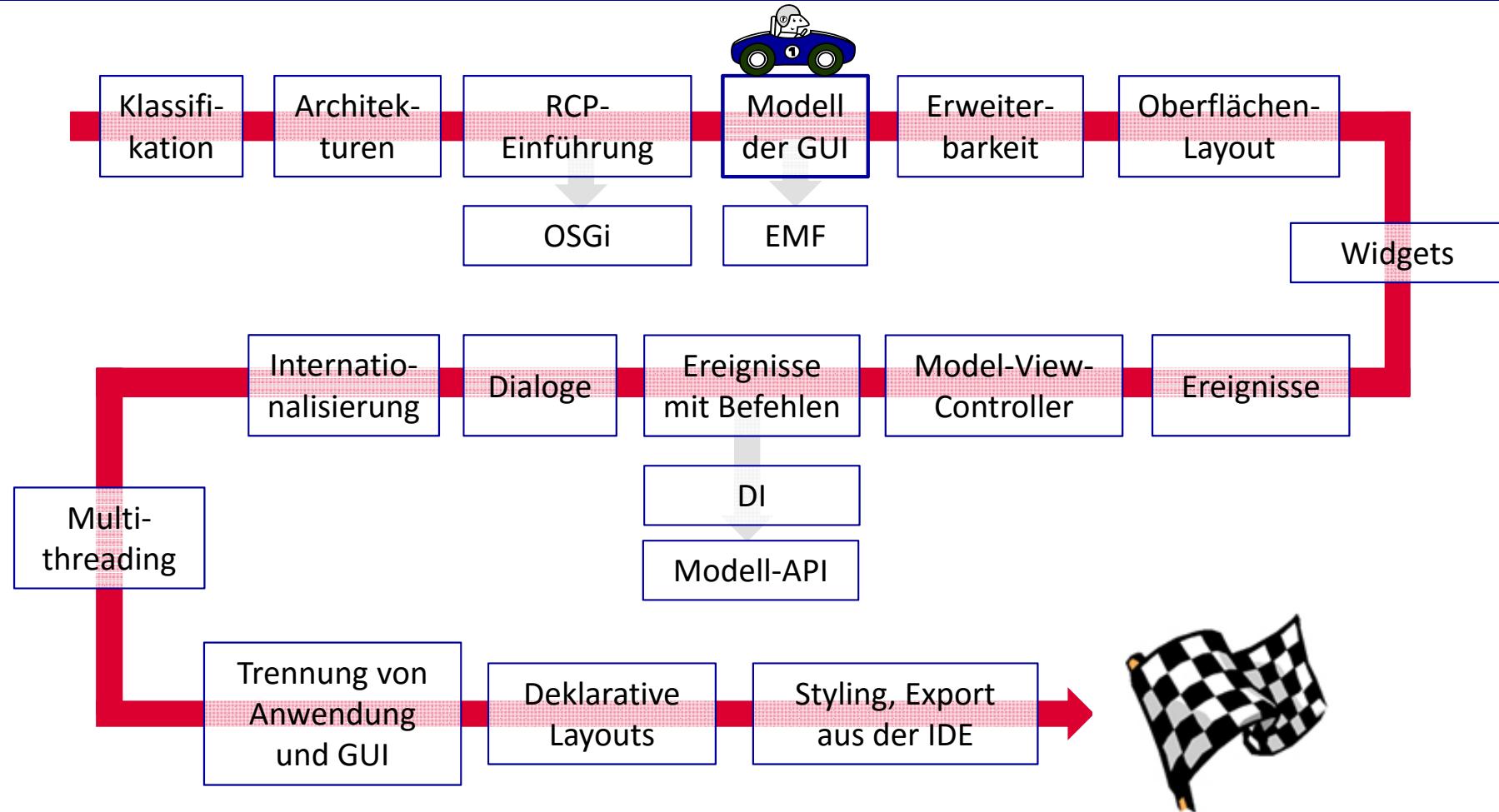


■ Auswahl des Features:



Modell der Oberfläche

Einführung

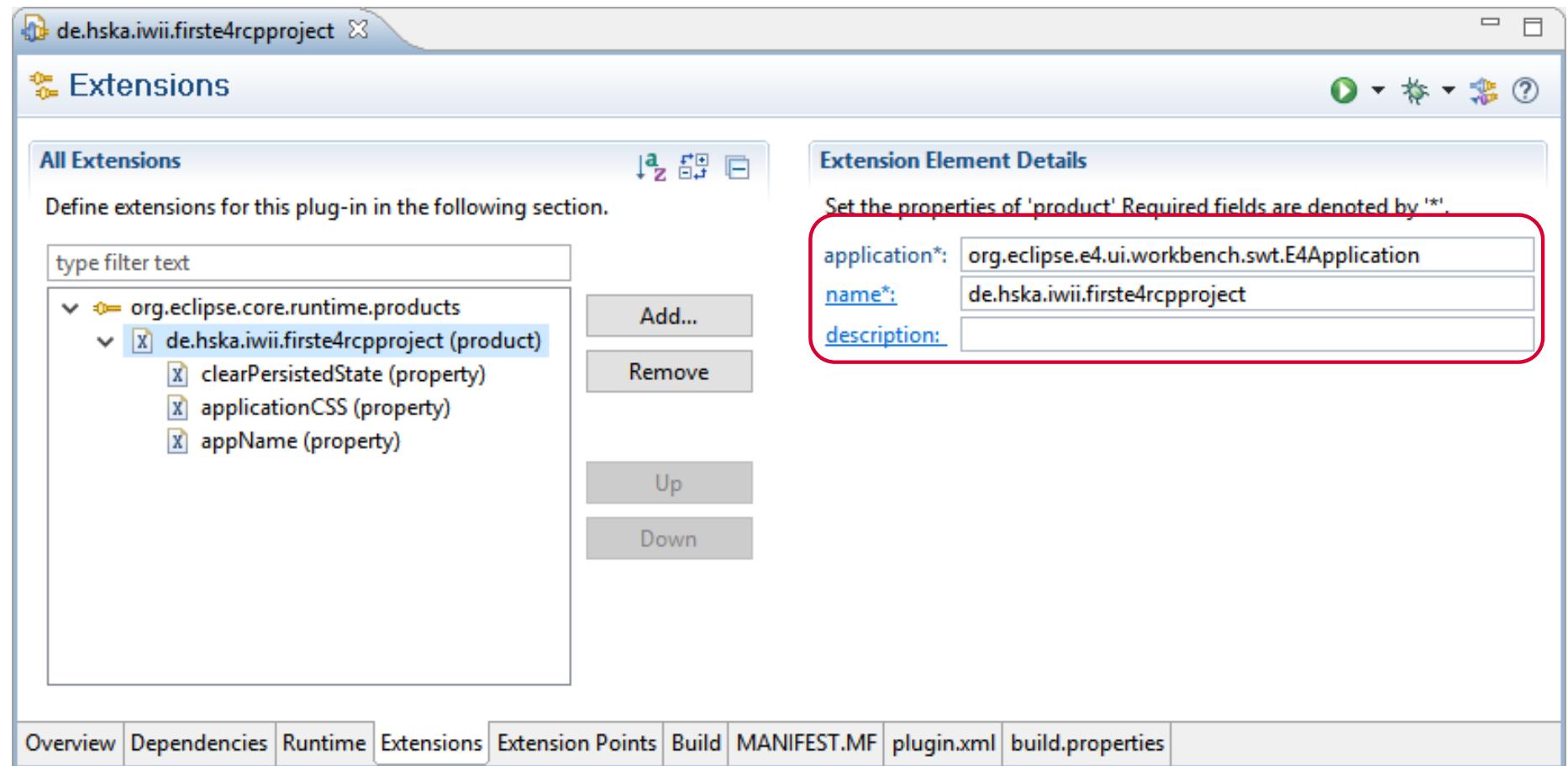


Modell der Oberfläche



Einführung

- Woher stammt in dem ersten Beispiel die Oberfläche? Sie wurde ja gar nicht gebaut.
- Was passiert beim Start der Anwendung eigentlich?
- Registrierung des Modells an **org.eclipse.core.runtime.products**:

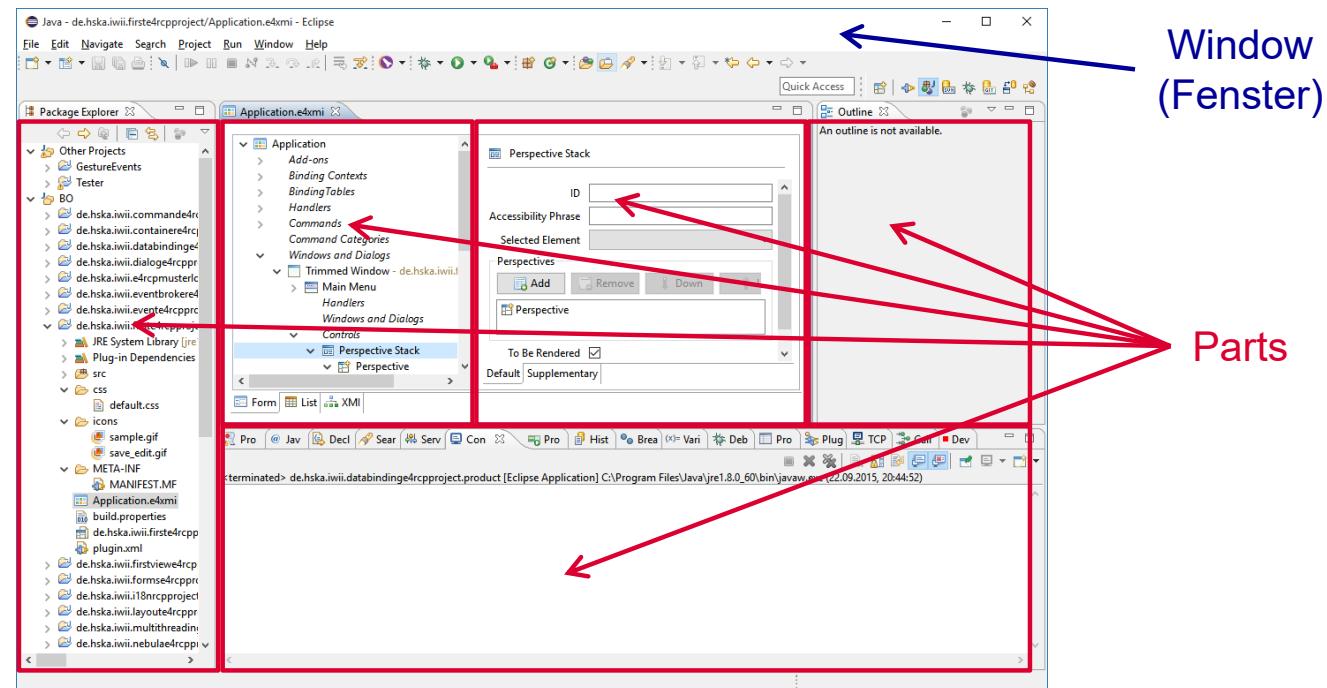


Modell der Oberfläche

Einführung

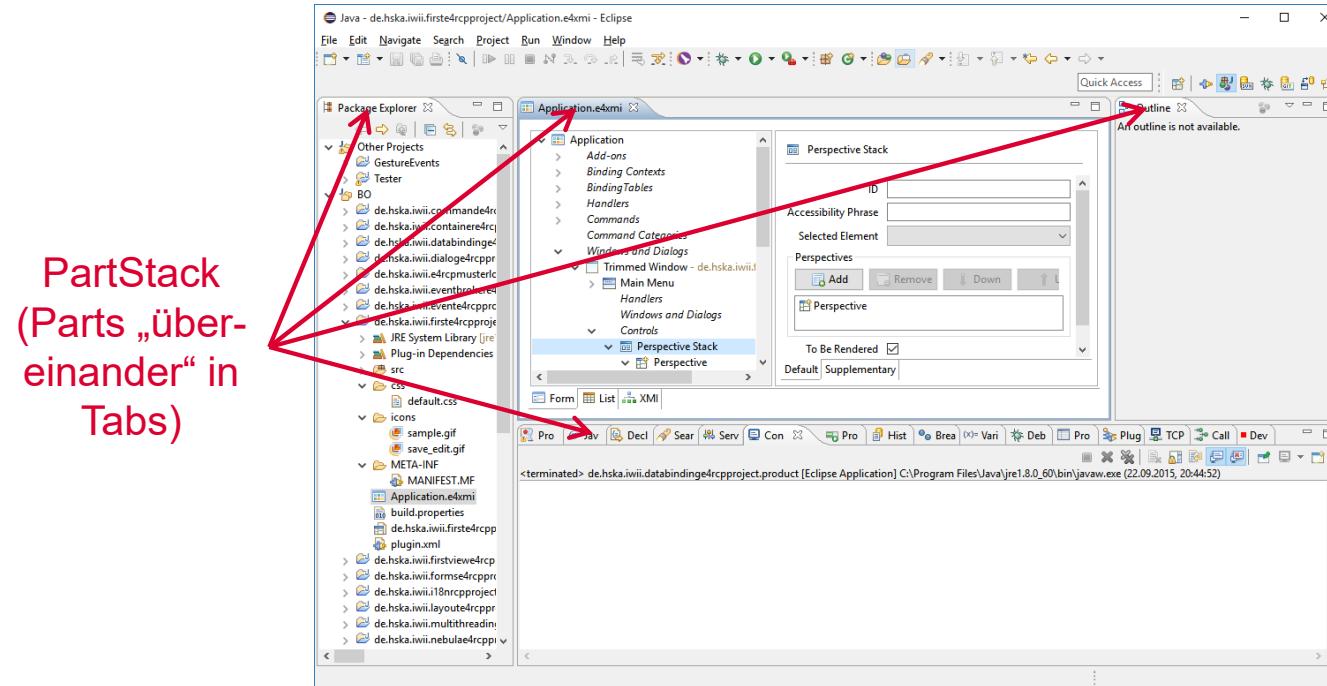


- Das Projekt besitzt eine XML-Datei mit den Namen **Application.e4xmi** in dem Plug-in **de.hska.iwii.firste4rcpproject**.
- Diese Datei beschreibt das Modell der Oberfläche!
- Beim Programmstart wird die Datei vom Framework ausgelesen, um daraus die Oberfläche zu erzeugen.
- Für das Verständnis sind einige Begriffe erforderlich.



Modell der Oberfläche

Einführung

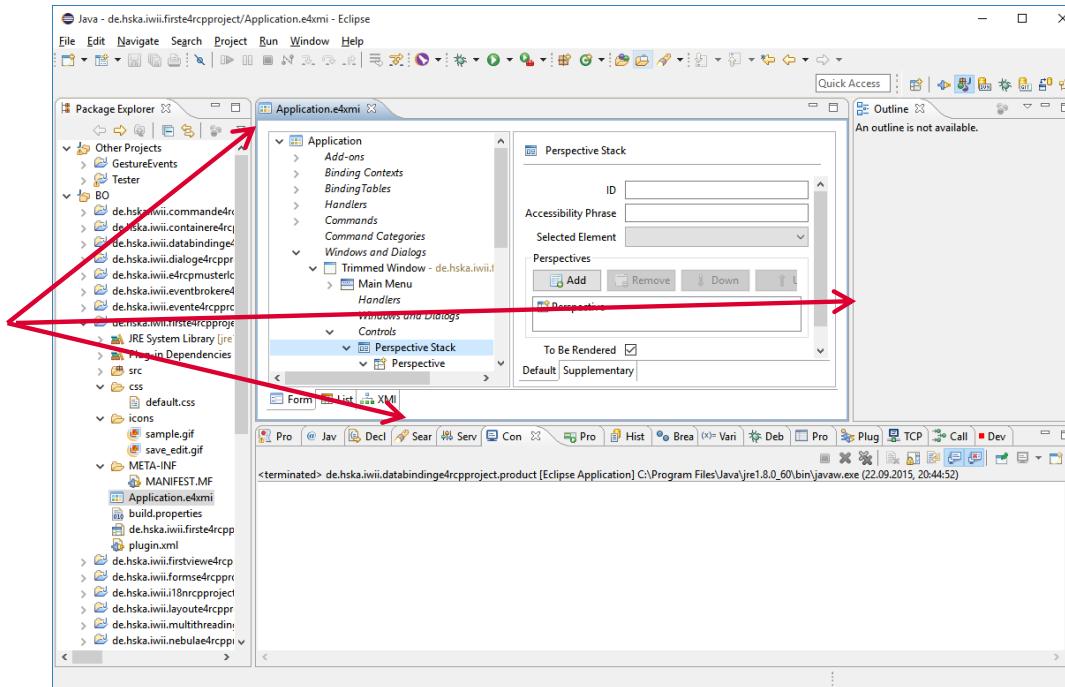


Modell der Oberfläche

Einführung



PartSashContainer
(Parts neben-
einander)

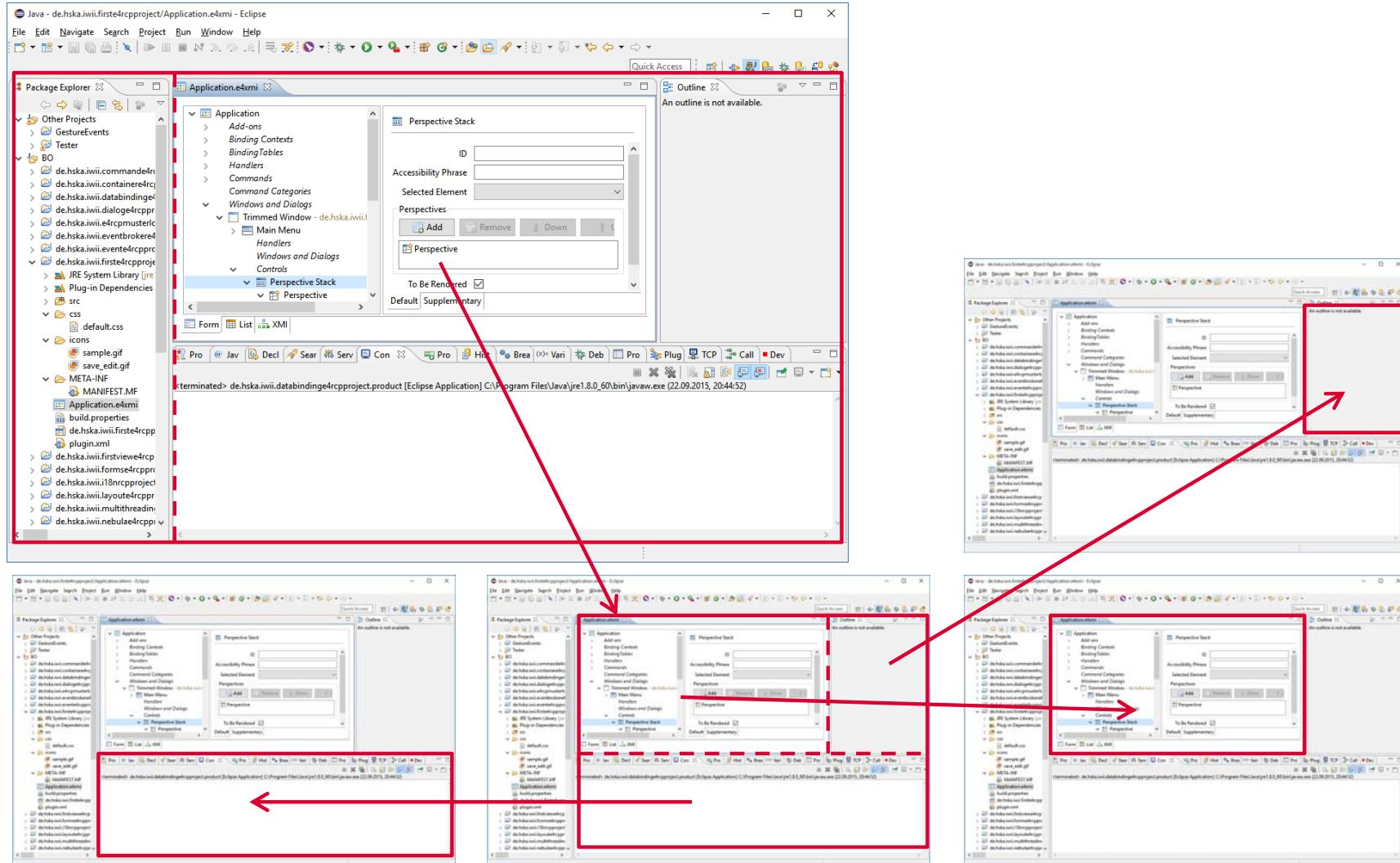


Modell der Oberfläche

Einführung



- PartSashContainer sind in dem Beispiel also geschachtelt:





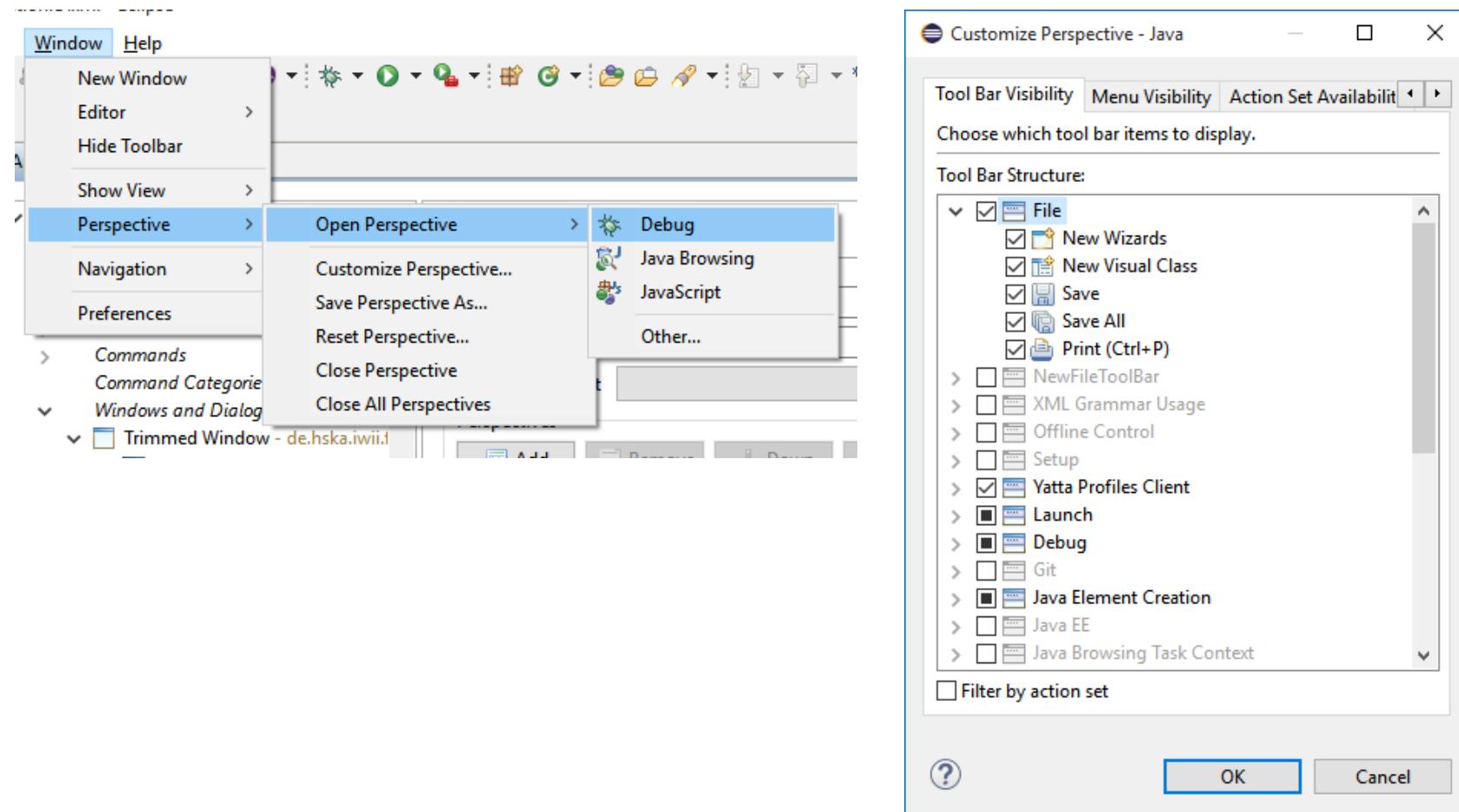
- Eclipse kennt zusätzlich noch den Begriff der Perspektive = eine Sicht auf eine Aufgabe.
- In der IDE gibt es u.A. diese Perspektiven: Java, Debug, SVN Repository Exploring, Plug-in Development, ...
- Eine Perspektive ordnet eine Menge von Parts für eine bestimmte Aufgabe an und lässt andere Parts weg.
- Eine Anwendung zeigt also immer nur eine Aufgabe (eine Perspektive) zu einem Zeitpunkt an.
- Das Anwendungsmodell kann beliebig viele Perspektiven auf einem „Perspective Stack“ verwalten.

Modell der Oberfläche

Einführung



- Die Perspektive (und damit die Anordnung der Sichten und Editoren) kann ausgewählt oder verändert werden, hier am Beispiel der IDE:

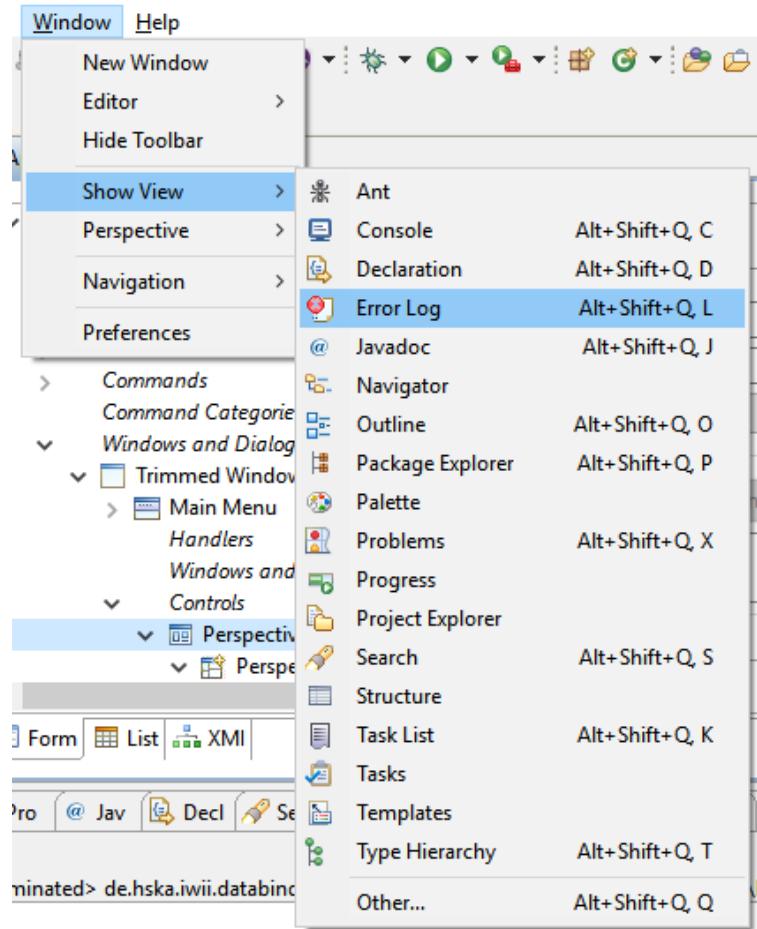


Modell der Oberfläche

Einführung



- In einer Perspektive lassen sich zusätzliche Parts einblenden:

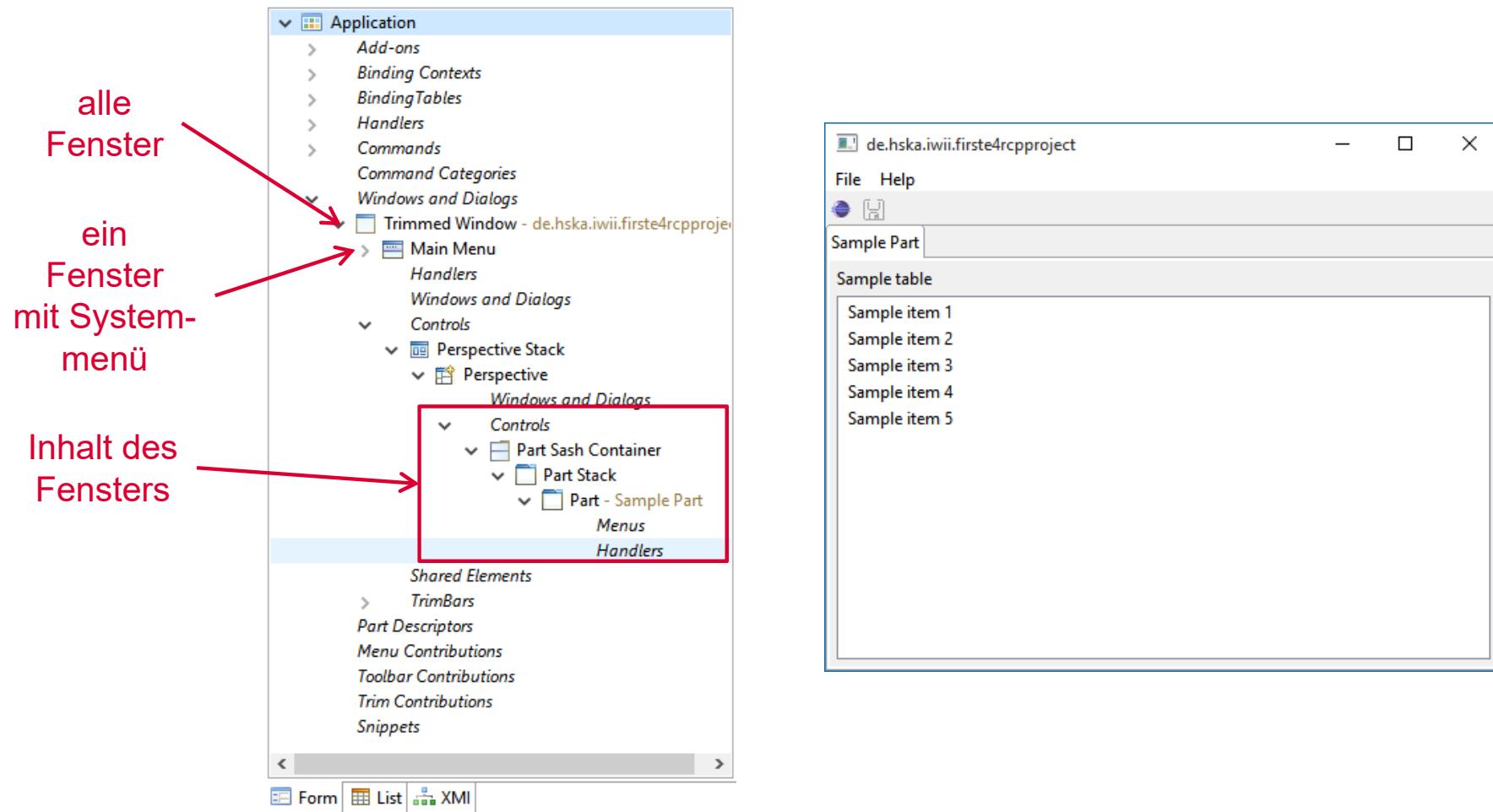


Modell der Oberfläche

Einführung



- Das Anwendungsmodell beschreibt u.A. die Fenster mit ihren Inhalten bis hinunter zu den Parts, nicht aber die Inhalte der Parts.



Modell der Oberfläche

Einführung



- Idee dahinter:
 - ◆ Das Modell beschreibt die Struktur der Anwendung: Wie setzt sie sich logisch aus den Inhalten unterschiedlicher Plug-ins zusammen.
 - ◆ Die Inhalte einzelner Parts werden von den Plug-ins selbst festgelegt.

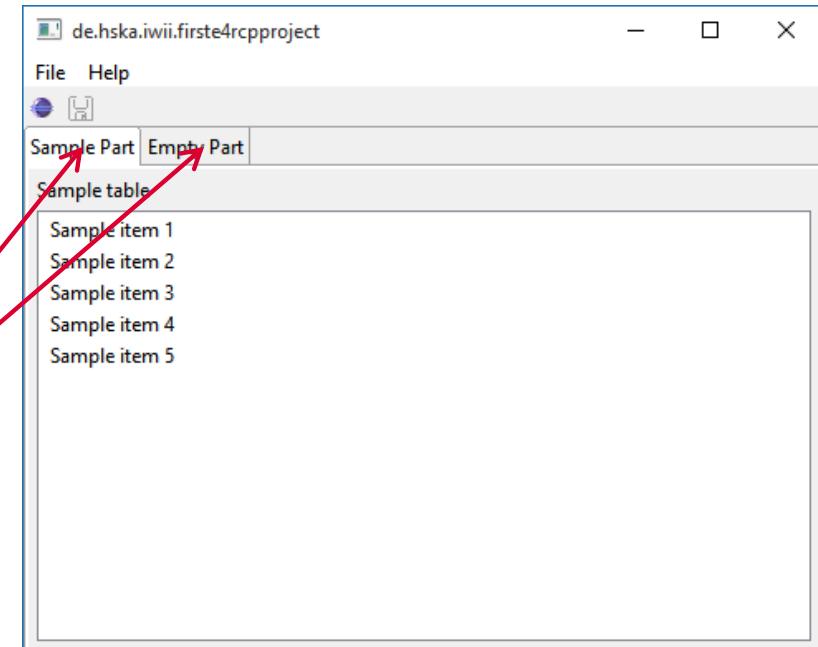
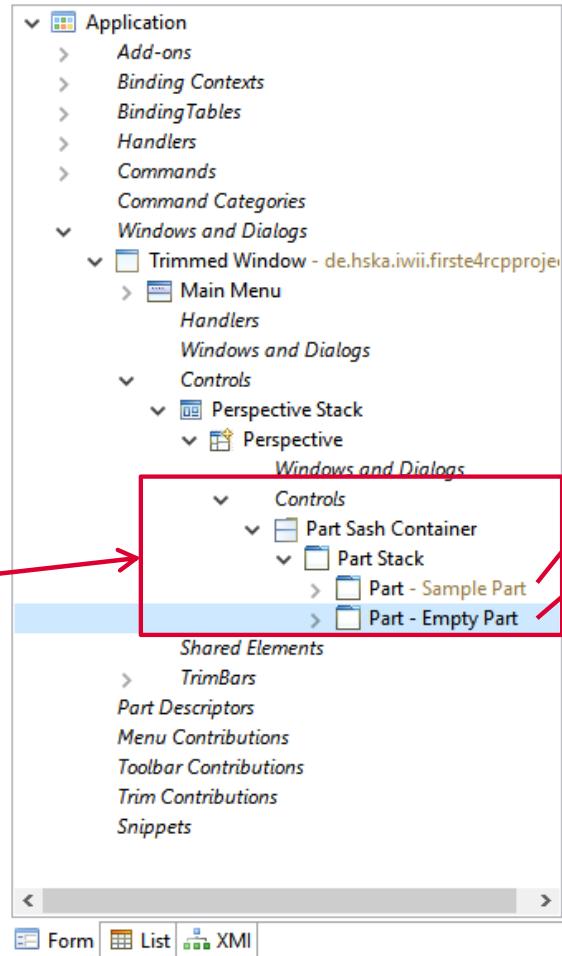
Modell der Oberfläche

Einführung



- Das Modell kann interaktiv in dem Editor bearbeitet werden („Rechtsklick“ mit der Maus ist hilfreich...). Nach dem Hinzufügen eines weiteren Parts:

Inhalt des Fensters
(mit Parts)





- Hinzufügen einer Ansicht (= eines Parts) zu einem Fenster:
 - ◆ Schreiben einer Klasse, die den Inhalt des Parts beschreibt.
 - ◆ Den Tastaturfokus auf ein Widget im Part setzen → muss erfolgen!
- Beispiel:

```
public class TableDataPart {  
  
    private ArrayList<Person> personModel = new ArrayList<Person>();  
  
    @Inject  
    public TableDataPart(Composite parent) {  
        // Oberfläche aufbauen → kommt noch  
    }  
  
    @Focus  
    public void setFocus() { // beliebiger Methodenname  
        einWidgetImPart.setFocus();  
    }  
    // ...  
}
```

- ◆ **@Inject**: Das Vaterelement **parent** wird per „Dependency Injection“ übergeben.

Modell der Oberfläche

Einführung



- Die Ansicht (Part) muss in das Modell eingetragen werden:

The screenshot shows the Eclipse Modeling Tools (EMF) interface for editing an E4 application model. The left pane displays the application structure under 'Application.e4xmi'. A 'Part Stack' node is selected. The right pane shows the configuration for a specific part named 'Personen'. The 'ID' field contains the value 'de.hsk.a.iwii.firstviewe4rcpproject.part.tabledata'. The 'Class URI' field contains the value 'bundleclass://de.hsk.a.iwii.firstviewe4rcpproject/part/TableData'. A red arrow points from the 'Part Stack' selection in the left pane to the 'ID' field in the right pane. Another red arrow points from the 'Class URI' field to the explanatory text below.

als Part hier innerhalb des PartStacks abgelegt

ID des Parts

Plug-in und Klassenname der Ansicht



- Anmerkungen zu den URLs innerhalb des Modells:
 - ◆ **bundleclass://<Bundle-SymbolicName>/<package>. <classname>:**
Name einer Klasse innerhalb des Plug-ins mit dem symbolischen Namen **Bundle-SymbolicName** (siehe **Manifest.mf**).
 - ◆ **platform:/plugin/<Bundle-SymbolicName>/<path>/<filename.extension>:** Ressource wie z.B. ein Bild innerhalb des Plug-ins **Bundle-SymbolicName** mit dem Pfad **path** und dem Dateinamen **filename.extension**
- Die Einträge können also aus ganz unterschiedlichen Plug-ins stammen:
 - ◆ Eine modulare Anwendung besteht aus vielen Plug-ins.
 - ◆ Die Plug-ins steuern ihrerseits Oberflächenelemente zur „Anwendung“ bei.
 - ◆ Eclipse als IDE besteht aus vielen Plug-ins, von denen einige eigene Oberflächenelemente beitragen → kommt noch genauer!

Modell der Oberfläche

Einführung



- Ergebnis:

The screenshot shows a Windows application window titled "de.hska.iwii.firstviewe4rcpproject". The menu bar includes "File" and "Help". Below the menu is a toolbar with icons for "Persons" and "Help". A table titled "Personen" displays two rows of data:

Name	Age
Name 1	42
Name 2	66

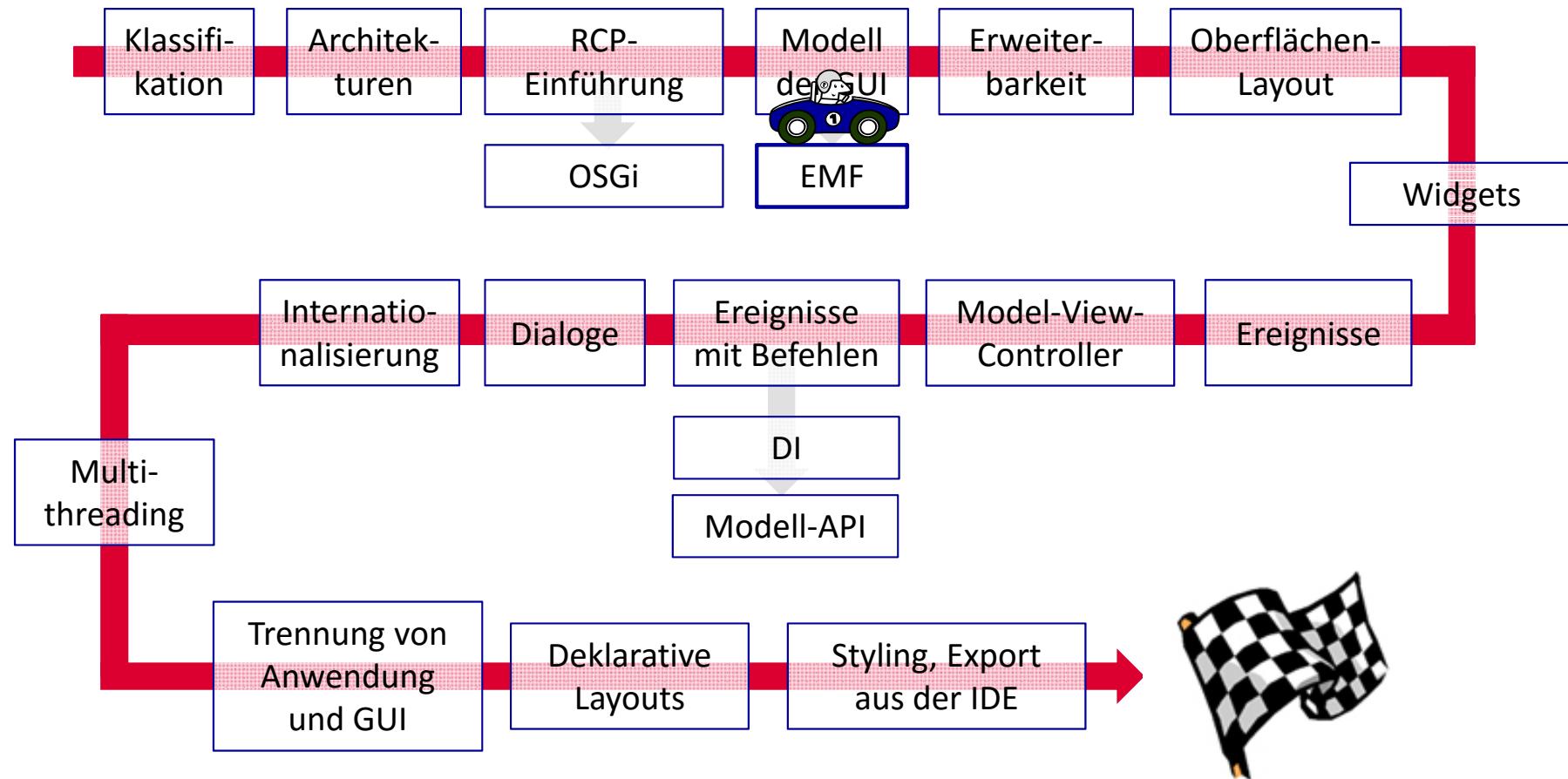
- Anmerkung:
 - ◆ Parts lassen sich programmatisch erzeugen.
 - ◆ Das Modell erleichtert es extrem, Parts aus anderen Plug-ins einzubinden.



- Die Parts werden bei Programmänderungen nicht immer sichtbar sein:
 - ◆ Die Eclipse RCP speichert in der Standardeinstellung die Positionen, Größen, Sichtbarkeiten usw. aller Oberflächenelemente.
 - ◆ Beim Wiederherstellen werden nur die vorher sichtbaren Elemente angezeigt.
 - ◆ Auswege:
 - Programm mit dem Kommandozeilenparameter **-clearPersistedState** starten (bei der Programmentwicklung ohnehin fast immer sinnvoll)
 - Ordner mit den Informationen löschen. Er heißt normalerweise **runtime-*<Projektname>*.product**
- Das Modell lässt sich „life“ ändern: Interaktiver Editor mit **ALT SHIFT F9** (auch innerhalb der IDE → nicht im Workbench-Fenster **To Be Rendered** auf **false** setzen!), das Plug-in „Model Spy“ muss vorhanden sein.
- Weitere Teile des Modells werden etwas später vorgestellt (z.B. Zugriff über eine API).

Eclipse Modeling Framework (EMF)

Einführung





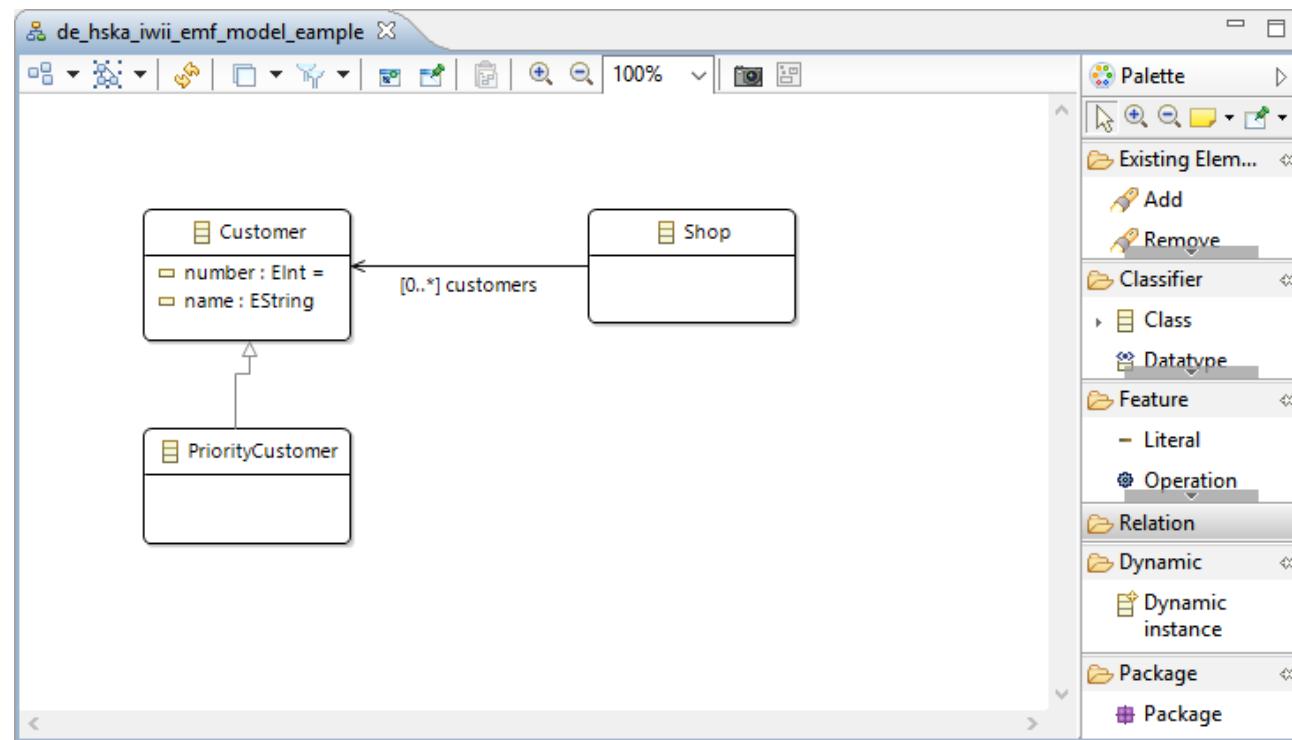
- Was verbirgt sich eigentlich hinter dem Modellansatz bei RCP?
- Wieso funktioniert der Life-Editor?
- EMF
 - ◆ ist ein Java-Framework,
 - ◆ unterstützt Anwendungen, die ein strukturiertes Modell besitzen,
 - ◆ kann Code aus Modellen generieren,
 - ◆ bietet eine sehr eingeschränkte Teilmenge dessen, was mit UML ausgedrückt werden kann → dadurch aber recht einfach verwendbar und
 - ◆ besitzt eine effiziente Teilelementierung des OMG MOF (genannt Ecore), sehr ähnlich zu „Essential MOF“ (EMOF).
- Beschreibungsmöglichkeit für Modelle:
 - ◆ XMI (XML Metadata Interchange), kann XMI aus Modellierungswerkzeugen einlesen → in der Vorlesung vorgestellt, weil RCP dieses Format verwendet
 - ◆ Annotierte Java-Klassen (keine „echten“ Java-Annotationen), stattdessen Annotationen im Kommentar

Eclipse Modeling Framework (EMF)



Einführung

- Zum Nachvollziehen der EMF-Beispiele sind der Editor der „Ecore Tools“ (<http://www.eclipse.org/ecoretools/download.html>) sowie das „EMF - Eclipse Modeling Framework SDK“ erforderlich.
- Dann kann ein Modell erstellt werden (Projekt **EMF-Beispiel**):



Eclipse Modeling Framework (EMF)

Einführung



- Das Diagramm wird im XMI-Format abgelegt:

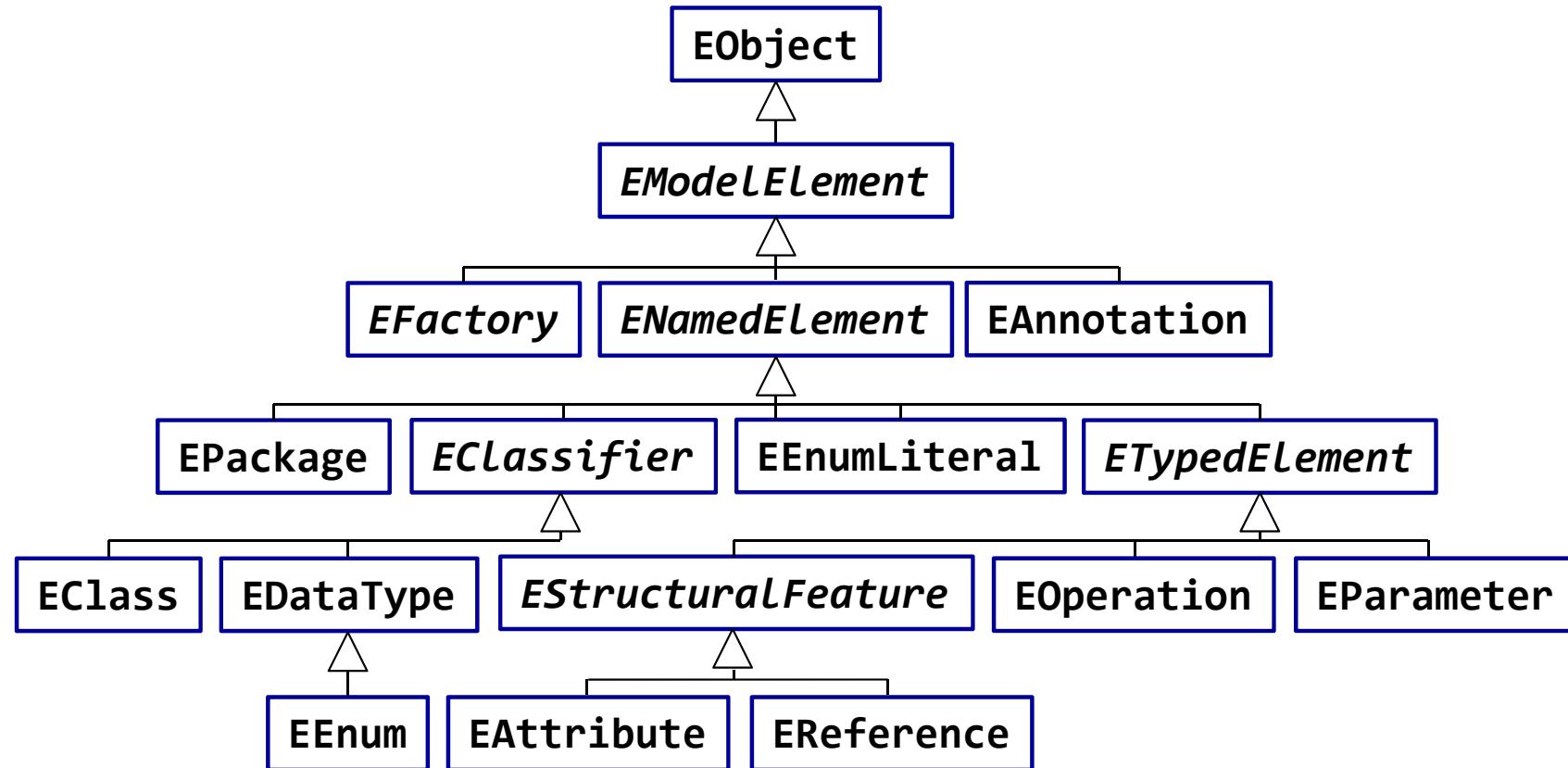
```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="de_hskai_ivii_emf_model_example" nsURI="http://www.eclipse.org/emf/2002/Ecore" nsPrefix="de_hskai_ivii_emf_model_example">
  <eClassifiers xsi:type="ecore:EClass" name="Customer">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="number" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Shop">
    <eStructuralFeatures xsi:type="ecore:EReference" name="customers" upperBound="-1" eType="#//Customer"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="PriorityCustomer" eSuperTypes="#//Customer"/>
</ecore:EPackage>
```

Eclipse Modeling Framework (EMF)



Einführung

- Modellelemente in EMF:



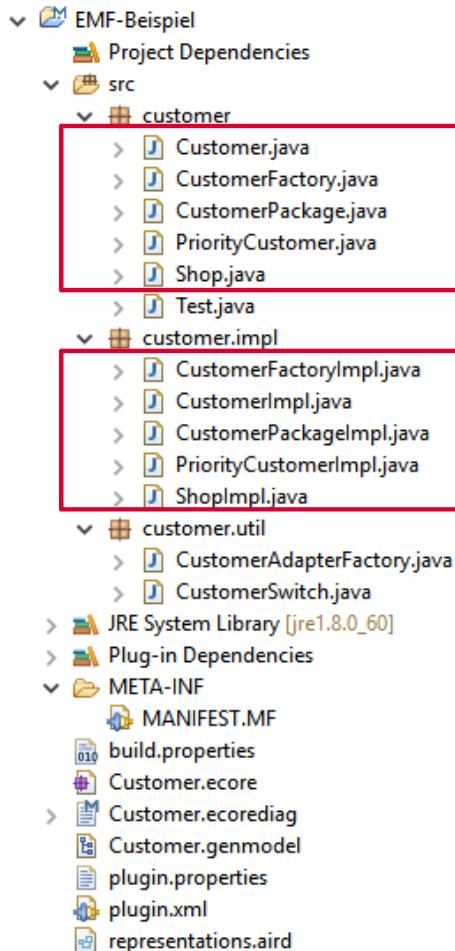
[[http://help.eclipse.org/mars/index.jsp?topic=%2Org.eclipse.emf.doc%2Fpreferences%2Foverview%2FEMF.html&cp=16_0_0](http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.emf.doc%2Fpreferences%2Foverview%2FEMF.html&cp=16_0_0)]

Eclipse Modeling Framework (EMF)



Einführung

- Aus der XMI-Repräsentation lassen sich Java-Klassen erzeugen:



Implementierungen
der Schnittstellen,
erben von **EObjectImpl**

Schnittstellen mit
Gettern und Settern,
erben von **EObject**



- An jedem **EObject** kann sich ein Beobachter registrieren.
- Die Setter-Methoden lösen bei Datenänderung Ereignisse aus:

```
/**  
 * <!-- begin-user-doc -->  
 * <!-- end-user-doc -->  
 * @generated  
 */  
public void setNumber(int newNumber) {  
    int oldNumber = number;  
    number = newNumber;  
    if (eNotificationRequired())  
        eNotify(new ENotificationImpl(this, Notification.SET, CustomerPackage.CUSTOMER__NUMBER, oldNumber, number));  
}
```

- Das gilt genauso für die Referenzen aus dem Diagramm: Werden Kunden zum Shop hinzugefügt oder aus dem Shop entfernt, dann werden Ereignisse ausgelöst.
- Der Entwickler arbeitet nur mit den Schnittstellen, nicht den konkreten Klassen.
- Die Objekte werden über Fabrikmethoden erzeugt.

Eclipse Modeling Framework (EMF)

Einführung



- Beispielcode mit Ereignisüberwachung:

```
Test.java
package customer;

import org.eclipse.emf.common.notify.Notification;

public class Test {

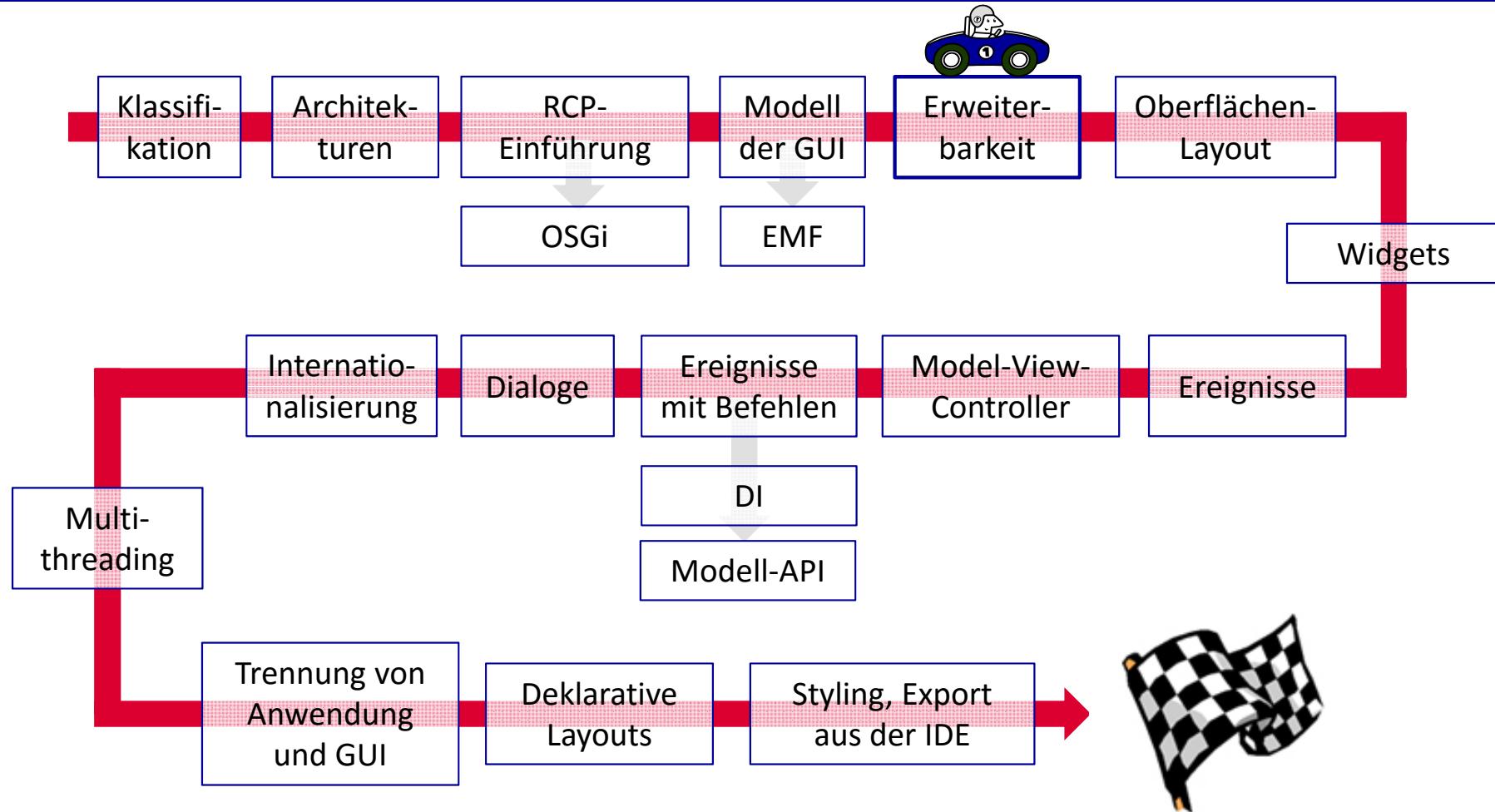
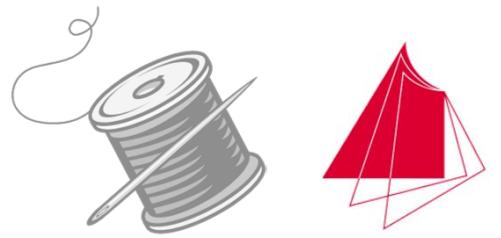
    /**
     * @param args
     */
    public static void main(String[] args) {
        CustomerFactory customerFactory = CustomerFactory.eINSTANCE;
        Shop shop = customerFactory.createShop();
        shop.eAdapters().add(new AdapterImpl() {
            @Override
            public void notifyChanged(Notification arg0) {
                System.out.println("notityChanged: " + arg0);
            }
        });
        Customer customer = customerFactory.createCustomer();
        shop.getCustomers().add(customer);
    }
}
```



- Was nützt dieses Wissen denn jetzt?
- Funktionsweise des Anwendungsmodells:
 - ◆ Das XMI-Modell ist eine Instanz eines Ecore-Modells.
 - ◆ Es wird beim Programmstart und damit zur Laufzeit eingelesen.
 - ◆ Die Eclipse-RCP registriert sich als Beobachter an den Modell-Objekten.
 - ◆ Damit werden Änderungen am Modell direkt überwacht und können sich auf die Darstellung auswirken → Live-Editor, zusätzliche Plug-ins!
 - ◆ Renderer „zeichnen“ das Modell mit Widgets, wobei normalerweise SWT zur Darstellung verwendet wird.
- Das Alles bleibt dem Entwickler normalerweise verborgen (zumindest in der Vorlesung nicht weiter interessant).

Erweiterbarkeit

Einführung





- Eine RCP-Anwendung „lebt“ davon, dass Plug-ins Dienste bereitstellen, die andere nutzen → siehe Einführung zu OSGi.
- Plug-ins sollen aber auch erweiterbar sein. Beispiel aus der Eclipse-IDE:

The screenshot shows the Eclipse IDE's Preferences dialog. The left sidebar lists various categories like JavaScript, Maven, Model Editor, etc., with 'WindowBuilder' expanded to show its sub-categories: Code Parsing, Swing, SWT, and XML. A red arrow points from the text 'WTP-Plug-in' to the 'Server' entry in the sidebar. Another red arrow points from the text 'WindowBuilder-Plug-in' to the 'WindowBuilder' entry in the sidebar. A third red arrow points from the text 'Diese Plug-ins erweitern die IDE an definierten Punkten.' to the main content area of the dialog, which contains various configuration options for the WindowBuilder plug-in.

Preferences

type filter text

JavaScript
Maven
Model Editor
Model Validation
Mylyn
ObjectAid
Oomph
Plug-in Development
Run/Debug
Server
Sirius
SWTBot Preferences
Team
Validation
Web
Web Services
WindowBuilder
Code Parsing
Swing
SWT
XML

WindowBuilder

Close and re-open any editors to see the effects of these preferences.
See '[Formatter](#)' to modify the Eclipse formatting preferences.

Editor layout: On separate notebook tabs (Source first)
Sync Delay (ms): 1000
Double-click on component tree to: Open editor at position of this widget

Associate WindowBuilder editor with automatically recognized Java GUI files
 Maximize editor on "Design" page activation
 Format source code (and reparse) on editor save
 Go to component definition in source on selection
 Automatically add to palette when using Choose Component
 Accept drop non-visual beans to design canvas
 Show debug information in console
 Show warning for incompatible Eclipse/WindowBuilder versions

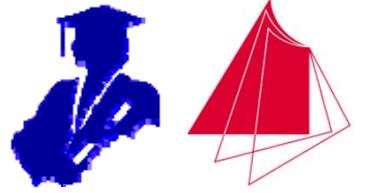
Restore Defaults Apply

OK Cancel

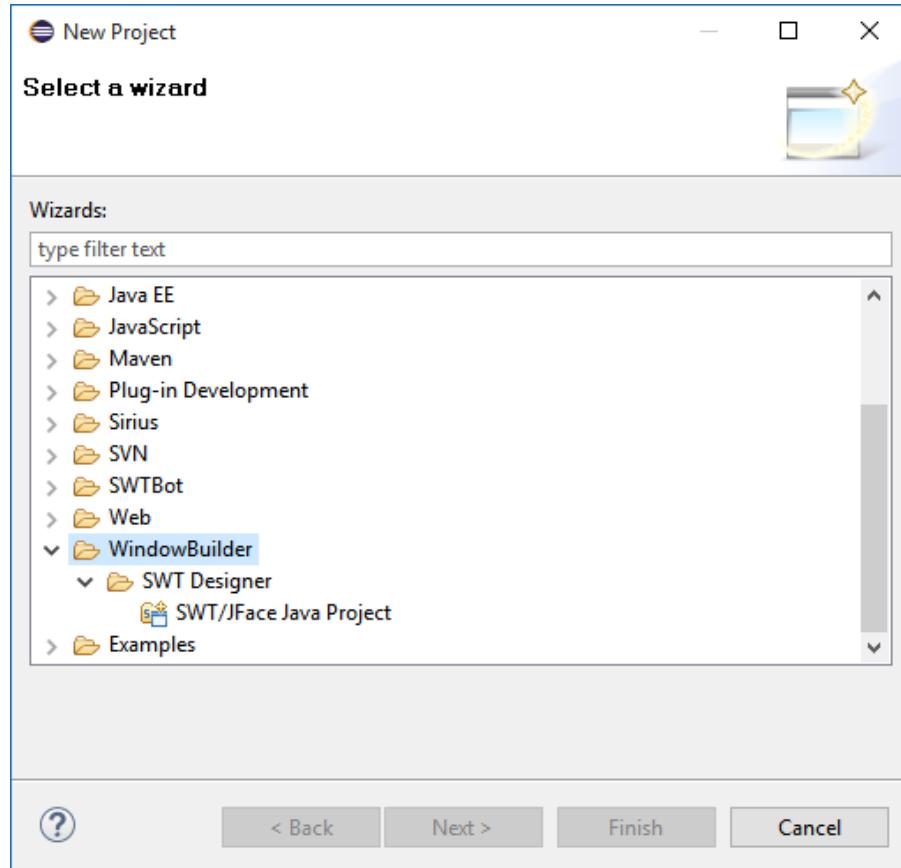
WTP-Plug-in

WindowBuilder-Plug-in

Diese Plug-ins erweitern die IDE an definierten Punkten.

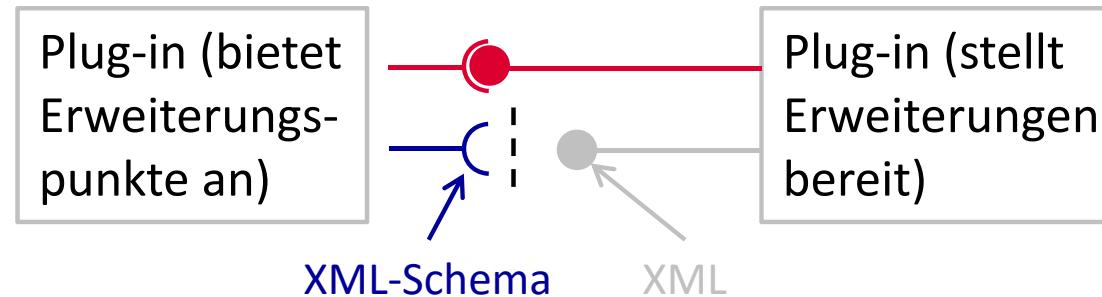


- Die Plug-ins haben sich an verschiedenen Stellen der IDE „eingenistet“:





- Die IDE und somit jede beliebige RCP-Anwendung ist ein Grundgerüst, das durch Erweiterungen ausgebaut werden kann.
- Dazu besitzt sie Erweiterungspunkte (extension points), an denen sich Erweiterungen (extensions) von Plug-ins „einklinken“ können.



- Die Erweiterung muss exakt zum Erweiterungspunkt passen.
- Wie soll das sichergestellt sein?
 - ◆ Der Erweiterungspunkt beschreibt seine Anforderungen an die Erweiterung durch einen Eintrag in seiner **plugin.xml**-Datei sowie in einem XML-Schema-Dokument.
 - ◆ Die Erweiterung wird in der **plugin.xml**-Datei des erweiternden Plug-ins beschrieben.



- Beschreibung des **Erweiterungspunktes** anhand des Anwendungsstarts in **plugin.xml** aus der RCP-Workbench:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
    <!-- Workbench extension points -->
    <extension-point id="org.eclipse.core.runtime.products"
        name="%productsName" schema="schema/products.exsd"/>
```

- **id**: eindeutige Identifikation innerhalb des Plug-ins, damit der Erweiterungspunkt von „außen“ angesprochen werden kann.
- **name**: symbolischer Name
- **schema**: Name der Schema-Datei, die die Anforderungen an die Erweiterung spezifiziert und Dokumentation enthält
- Der Aufbau der Schema-Datei ist länglich...
- Sowohl die Einträge in die **plugin.xml**-Datei als auch in die Schema-Datei werden durch Formulare vorgenommen (keine manuelle Arbeit erforderlich) → das Eclipse-Paket „Eclipse for RCP/Plug-in Developers“ enthält Wizards sowie die Ausfüllhilfen.



- Stark gekürzte Schema-Beschreibung des Erweiterungspunktes
org.eclipse.core.runtime.products als Startpunkt für eine Anwendung:

```
....  
<element name="extension">  
  <complexType>  
    <sequence>  
      <choice>  
        <element ref="product"/>    <!-- Für eine statische Erweiterung -->  
        <element ref="provider"/>   <!-- Für eine dynamische Erweiterung -->  
      </choice>  
    </sequence>  
    <attribute name="point" type="string" use="required"/>  
    <attribute name="id" type="string"/>  
  ....  
  → <element name="product">  
    <sequence>  
      <element ref="property" minOccurs="0" maxOccurs="unbounded"/>  
    </sequence>  
    <complexType>  
      <attribute name="application" type="string" use="required"/>  
      <attribute name="name" type="string" use="required"/>  
    ....
```



- Beschreibung der Erweiterung aus dessen **plugin.xml**-Datei (Verwendung des Erweiterungspunktes)

```
<extension
    id="product"
    point="org.eclipse.core.runtime.products">
    <product
        name="de.hska.iwii.erste4rcpproject"
        application="org.eclipse.e4.ui.workbench.swt.E4Application">
        <property
            name="appName"
            value="de.hska.iwii.erste4rcpproject">
        </property>
        <property
            name="clearPersistendState"
            value="true">
        </property>
        <property
            name="applicationCSS"
            value="platform:/plugin/de.hska.iwii.erste4rcpproject/css/default.css">
        </property>
    </product>
</extension>
```



- Oder viel einfacher: Erweiterung durch Formulareinträge beschrieben

The screenshot shows two instances of the Eclipse RCP Project Properties dialog, one on top of the other. Both dialogs are for the project "de.hska.iwii.erste4rcpproject".

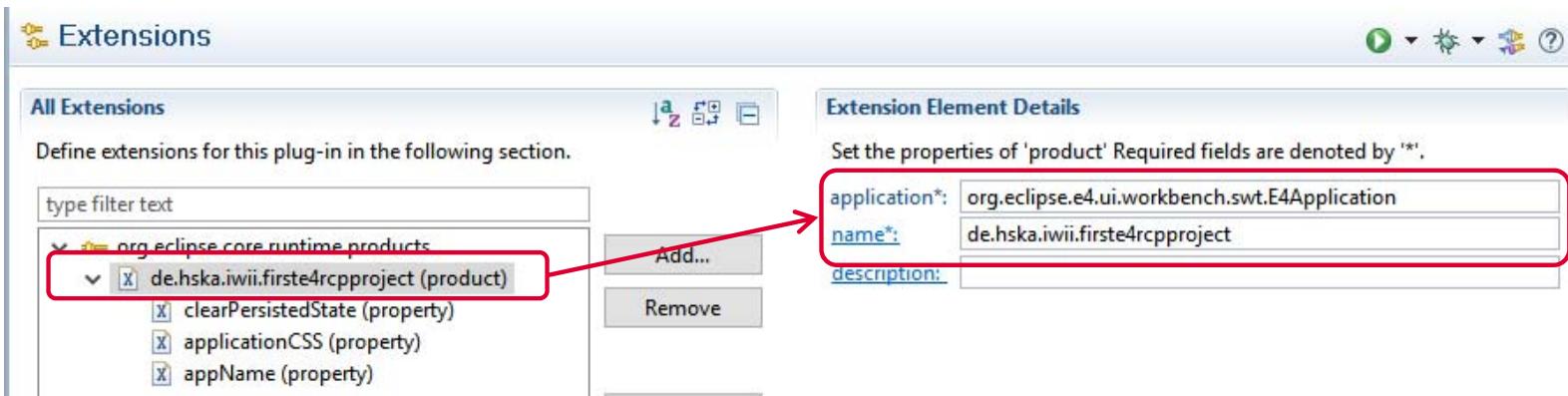
The top dialog is titled "Extensions" and shows the "Extension Details" section for a selected extension. The "ID" field is set to "product" and the "Name" field is empty. The left pane lists extensions under "org.eclipse.core.runtime.products": "de.hska.iwii.erste4rcpproject (product)" which contains "clearPersistedState (property)", "application (property)", and "appName (property)".

The bottom dialog is also titled "Extensions" and shows the "Extension Element Details" section for the "product" extension. It lists the properties: "application*" set to "org.eclipse.e4.ui.workbench.swt.E4Application", "name*" set to "de.hska.iwii.erste4rcpproject", and "description" left empty. The left pane lists the same extension structure as the top dialog.

Both dialogs have tabs at the bottom: Overview, Dependencies, Runtime, Extensions, Extension Points, Build, MANIFEST.MF, plugin.xml, and build.properties. The "Extensions" tab is selected in both cases.



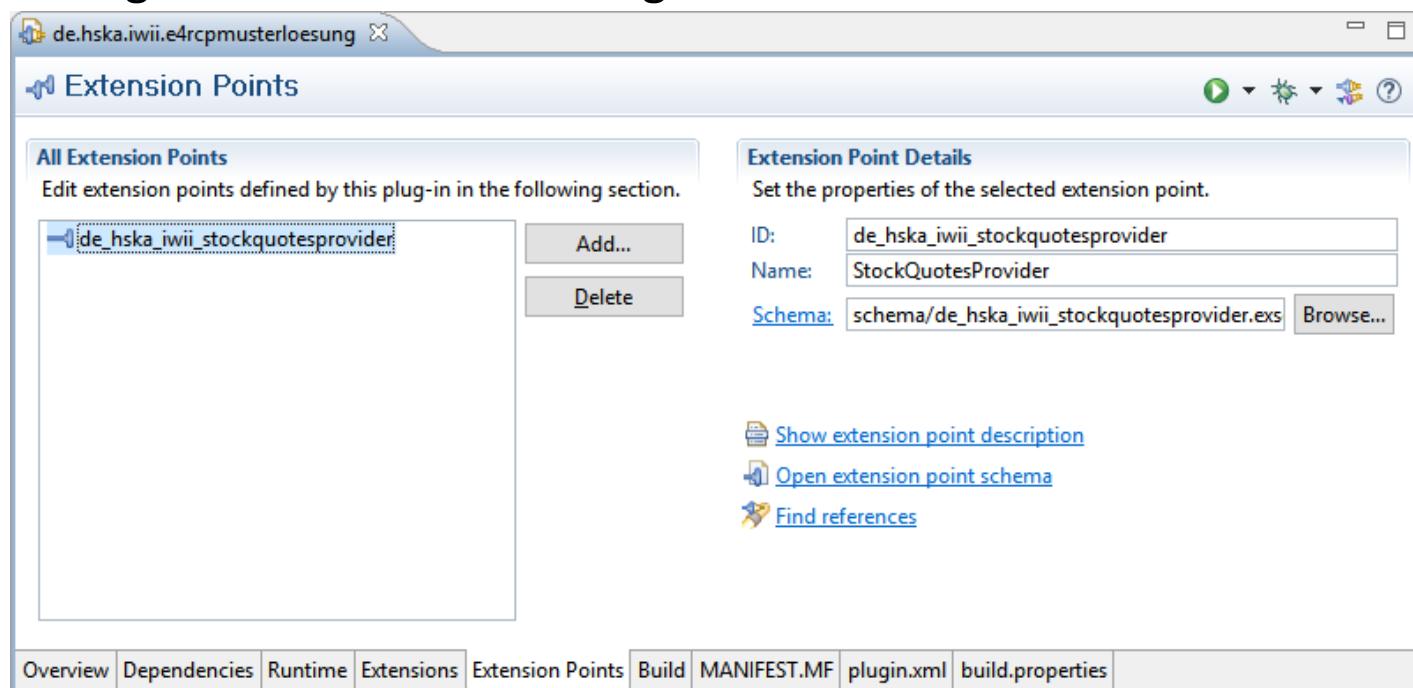
- Wo ist denn nun **main**? Die „Anwendung“ ist keine Anwendung. Sie ist eine Sammlung von Plug-ins.
- Das Framework bietet die Möglichkeit, über einen Erweiterungspunkt ein Anwendungs-Modell bereitzustellen:



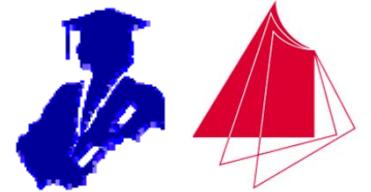
- Wie kommt dann die Oberfläche der Anwendung zustande?
 - ◆ Die Eigenschaft („property“) **applicationXMI** enthält den Namen der XML-Datei des Modells der Anwendung (hier nicht verwendet → Standard-Name).
 - ◆ Die Eigenschaft **applicationCSS** enthält den Namen der CSS-Datei, die das Aussehen der Anwendung bestimmt.



- Erzeugen eines eigenen Erweiterungspunktes:
 - ◆ Erweiterungspunkt mit **Add...** anlegen und ID, Name sowie den Namen der zu erzeugenden Schemadatei angeben:



- ◆ Diese Daten stehen in der Datei **plugin.xml**. Die Schemainformationen landen in einer separaten Datei (hier **de_hska_iwii_stockquotesprovider.exsd**).
- ◆ Die ID sollte nur Klein- und Großbuchstaben sowie den Unterstrich enthalten.



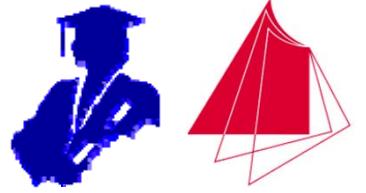
- Der Erweiterungspunkt dient als Beispiel für die Daten-Plug-ins der Bonusaufgabe:
 - de.hska.iwii.stockquotes.net.IStockQuotes** muss von jedem Plug-in implementiert werden.
 - Es dürfen beliebig viele Datenquellen-Plug-ins angegeben werden. Mindestens eines ist erforderlich.

The screenshot shows a software interface for editing XML schema definitions (XSD). The title bar reads "de_hska_iwii_stockquotesprovider.exsd". The main window is titled "StockQuotesProvider".

The left pane, titled "Extension Point Elements", contains a tree view of elements under "StockQuotesProvider". It includes nodes for "extension", "Sequence", "point", "id", "name", "call", and "class".

The right pane, titled "Attribute Details", is configured for the "class" attribute of the "extension" element. The "Name" field is set to "class". The "Deprecated" field has "false" selected. The "Use" field is set to "required". The "Type" field is set to "java". The "Implements" field contains the fully qualified class name "de.hska.iwii.stockquotes.net.IStockQuotes".

At the bottom of the interface, there are tabs for "Overview", "Definition", and "Source".



- Erläuterung:
 - ◆ Das Element **call** nimmt die Referenz auf ein Daten-Plug-in auf.
 - ◆ Es kommt min. einmal vor.
 - ◆ Das Attribut **class** in **call** bekommt beim Hinzufügen einer Erweiterung den Namen der Plug-in-Klasse angegeben, die die geforderte Schnittstelle implementiert.
 - ◆ **implements** beinhaltet die zu implementierende Schnittstelle des Plug-ins.



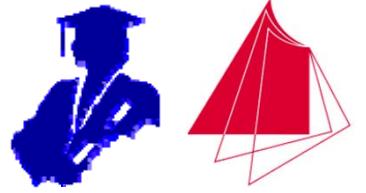
- Und wie kann ein Plug-in auf Erweiterungen an seinen eigenen Erweiterungspunkten zugreifen?

```
// Registry für die am Plug-in angemeldeten Erweiterungen holen  
IExtensionRegistry registry = Platform.getExtensionRegistry();  
  
// Referenz auf den eigenen Erweiterungspunkt mit dem angegebenen Namen  
// holen. Name des Erweiterungspunktes: Anwendungs-ID.EP.Erweiterungspunkt-ID  
IExtensionPoint point = registrygetExtensionPoint(  
    "de.hska.iwii.e4rcpmusterloesung.de_hska_iwii_stockquotesprovider");  
if (point != null) {  
    // Alle Erweiterungen auslesen.  
    IExtension[] extensions = point.getExtensions();  
    for (int i = 0; i < extensions.length; i++) {  
        // Elemente call auslesen  
        IConfigurationElement[] callElements =  
            extensions[ i ].getConfigurationElements();  
        for (IConfigurationElement callElement: callElements) {  
            IStockQuotes plugin = getPluginObject(callElement);  
            // Plug-in irgendwo speichern  
        }  
    }  
}
```



- Es fehlt noch die Methode, die das Plug-in lädt und das referenzierte Objekt erzeugt:

```
private IStockQuotes getPluginObject(IConfigurationElement callElement) {  
    // Objekt erzeugen, dessen Klassenname im Attribut "class"  
    // gespeichert ist  
    try {  
        Object service = callElement.createExecutableExtension("class");  
        if (service instanceof IStockQuotes) {  
            return (IStockQuotes) service;  
        }  
        return null;  
    }  
    catch (CoreException ex) {  
        // Fehlerbehandlung  
    }  
}  
}
```



- Ein Plug-in, das Erweiterungspunkte anbietet, kann sich als Beobachter daran registrieren: Es wird informiert, wenn sich Erweiterungen an- und abmelden.
- Wichtig:
 - ◆ Die Verknüpfung zwischen beiden übernimmt die Plattform.
 - ◆ Sie stellt auch sicher, dass Bedingungen (z.B. die Kardinalität der Erweiterungen) eingehalten werden → nicht mehr als eine Anwendung.

Erweiterbarkeit

Erweiterungen und Erweiterungspunkte



- Wie lassen sich die ganzen Erweiterungspunkte finden?

The screenshot shows the Eclipse Help interface for the Platform Plug-in Developer Guide. The left sidebar contains a tree view of various developer guides, with 'Extension Points Reference' selected. The main content area is titled 'Platform Extension Points' and describes how they can be used to extend platform infrastructure. It lists extension points under three categories: 'Platform runtime', 'Workspace', and 'Platform Text', each followed by a list of URLs.

Platform runtime

- [org.eclipse.core.contenttype.contentTypes](#)
- [org.eclipse.core.runtime.adapters](#)
- [org.eclipse.core.runtime.applications](#)
- [org.eclipse.core.runtime.contentTypes](#)
- [org.eclipse.core.runtime.preferences](#)
- [org.eclipse.core.runtime.products](#)
- [org.eclipse.equinox.preferences.preferences](#)

Workspace

- [org.eclipse.core.filesystem.filesystems](#)
- [org.eclipse.core.resources.builders](#)
- [org.eclipse.core.resources.fileModificationValidator](#)
- [org.eclipse.core.resources.filterMatchers](#)
- [org.eclipse.core.resources.markers](#)
- [org.eclipse.core.resources.modelProviders](#)
- [org.eclipse.core.resources.moveDeleteHook](#)
- [org.eclipse.core.resources.natures](#)
- [org.eclipse.core.resources.refreshProviders](#)
- [org.eclipse.core.resources.teamHook](#)
- [org.eclipse.core.resources.variableResolvers](#)

Platform Text

- [org.eclipse.core.filebuffers.annotationModelCreation](#)
- [org.eclipse.core.filebuffers.documentCreation](#)
- [org.eclipse.core.filebuffers.documentSetup](#)
- [org.eclipse.ui.editors.annotationTypes](#)
- [org.eclipse.ui.editors.documentProviders](#)
- [org.eclipse.ui.editors.markerAnnotationSpecification](#)

Erweiterbarkeit

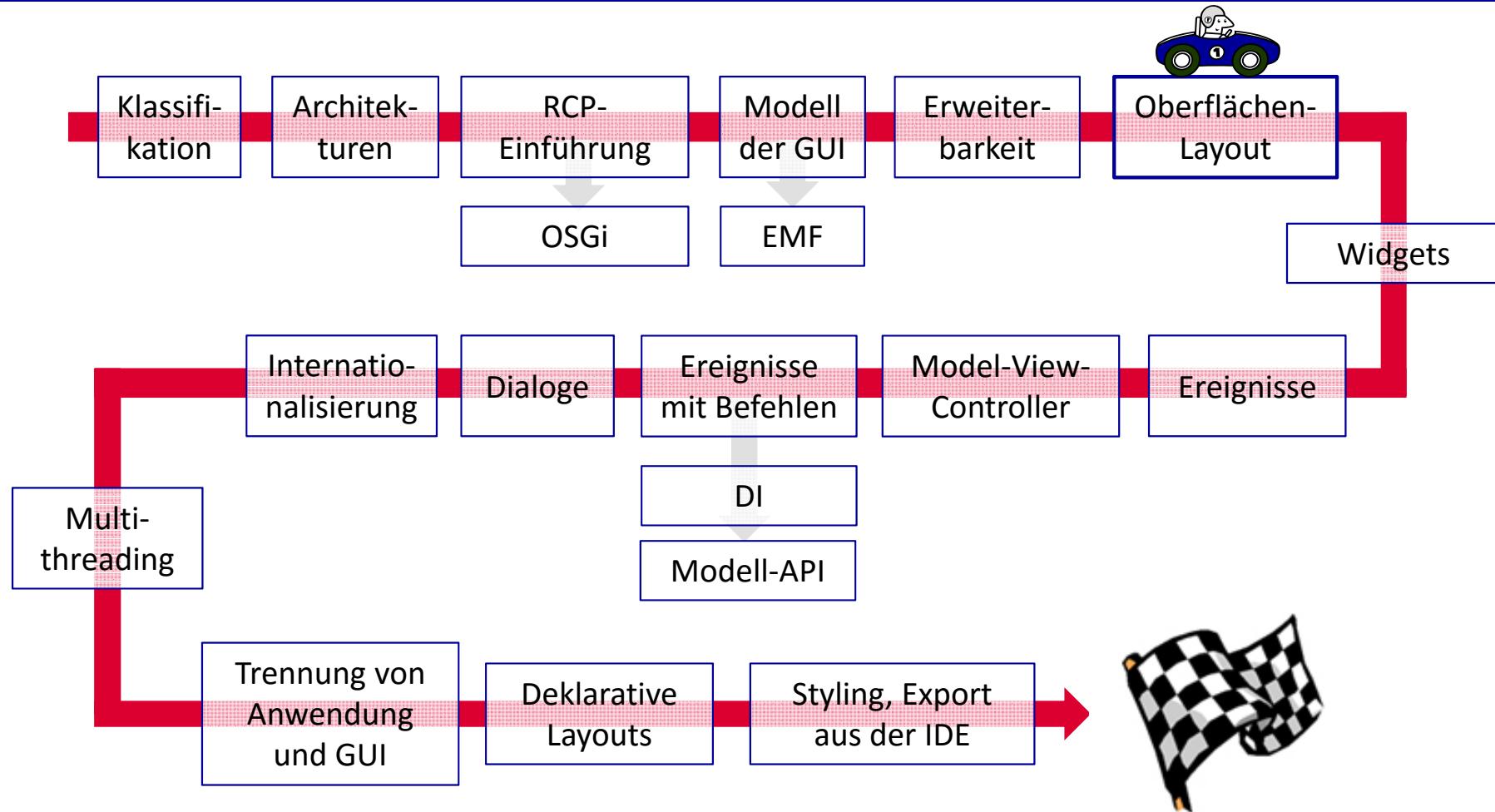
Erweiterungen und Erweiterungspunkte



- Mit Eclipse 4 werden wesentlich weniger Erweiterungen benutzt als noch in Eclipse 3.
- Das Anwendungs-Modell übernimmt einen großen Teil der Aufgaben.

Oberflächenlayout

Übersicht



Oberflächenlayout

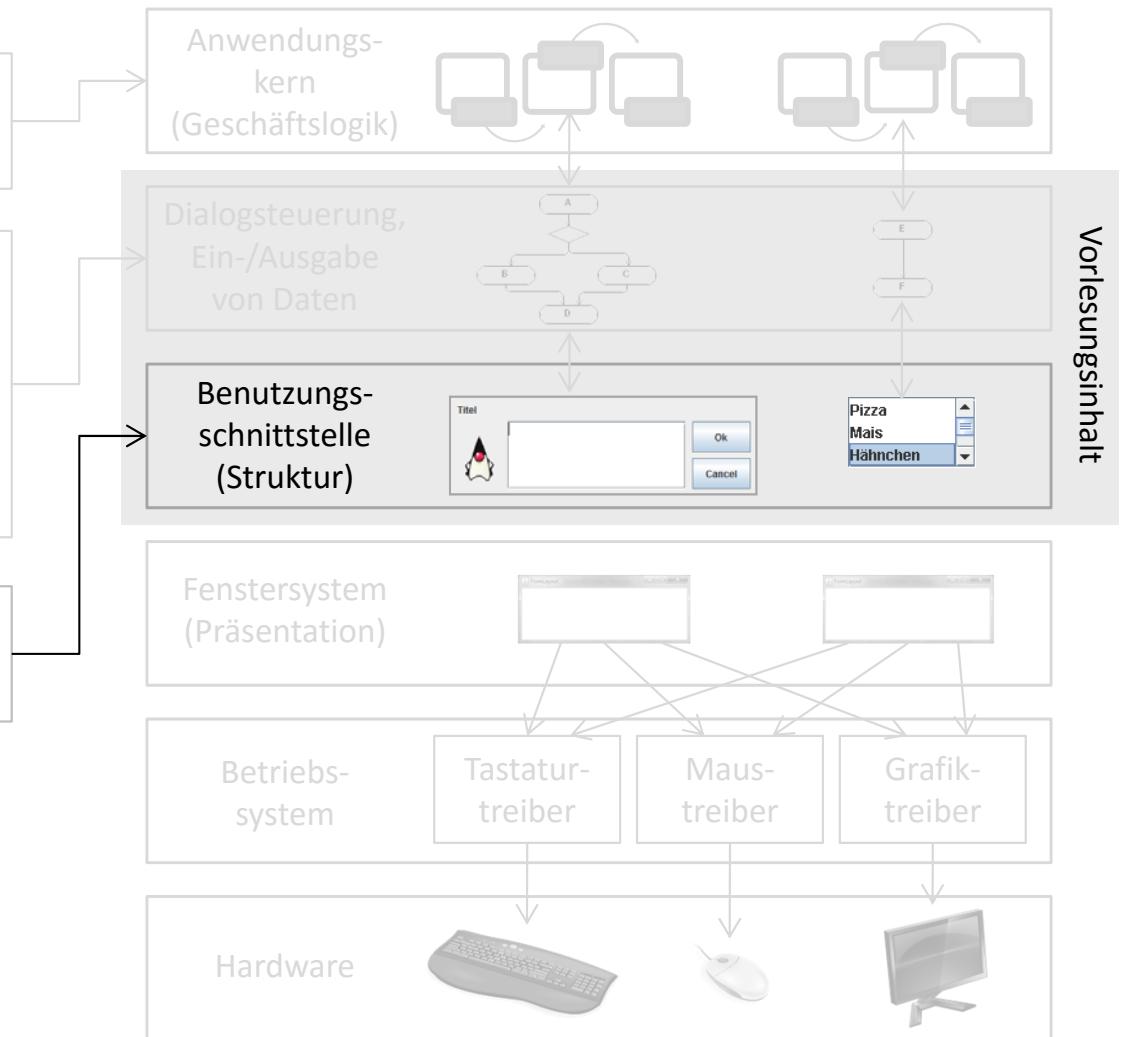
Übersicht



die eigentliche Funktionalität,
von der Dialogsteuerung aufgerufen

kontrolliert Aktionen des Benutzers auf
Plausibilität, steuert die Benutzungs-
schnittstelle und die Geschäftslogik,
synchronisiert Daten in der Oberfläche
und Geschäftslogik

verantwortlich für die Darstellung und
Verwaltung des Inhalts in den Fenstern



Oberflächenlayout

Eine erste SWT-Anwendung



- Source-Code ([FirstSWTApplication.java](#)) einer „nackten“ SWT-Anwendung ohne RCP:

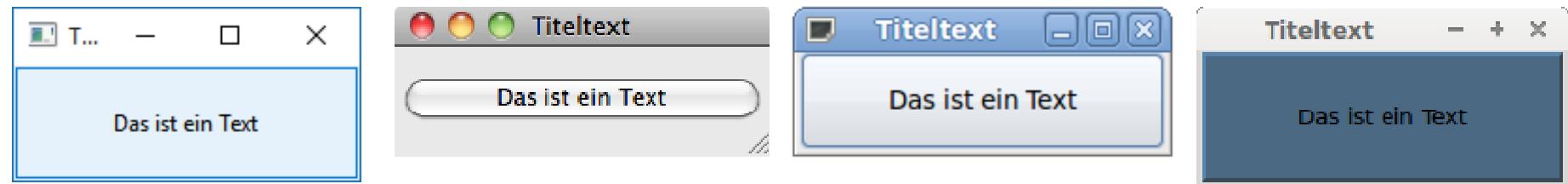
```
public class FirstSWTApplication {  
  
    private Display display;  
    private Shell shell;  
  
    public FirstSWTApplication(Display display) {  
        this.display = display;  
        shell = new Shell(display);  
        shell.setText("Titeltext");  
        shell.setSize(200, 100);  
    }  
  
    private void createGUI() {  
        Button button = new Button(shell, SWT.PUSH);  
        button.setText("Das ist ein Text");  
        button.setBounds(shell.getClientArea());  
        shell.pack();  
    }  
}
```

Oberflächenlayout

Eine erste SWT-Anwendung



```
public void run() {  
    createGUI();  
    shell.open();  
    while (!shell.isDisposed()) {  
        if (!display.readAndDispatch())  
            display.sleep();  
    }  
    display.dispose();  
}  
  
public static void main(String[] args) {  
    new FirstSWTApplication(Display.getDefault()).run();  
}  
}
```





- Erste Erklärungen:
 - ◆ **Display**: repräsentiert das Fenstersystem, in dem das Programm läuft
 - ◆ **Shell**: „Top-Level“-Fenster des Fenstersystems
 - ◆ **new Button**: Fügt eine Taste in das Fenster ein → Erklärungen zur Positionierung folgen ...
SWT.PUSH: Das ist eine normale Taste (Eigenschaften werden über Konstanten eingetragen ...).
 - ◆ **button.setText**: Beschriftung der Taste eintragen.
 - ◆ **button.setBounds**: Die Taste wird so groß wie der Inhaltsbereich des Fensters.
 - ◆ **shell.pack**: Layout-Berechnung starten, Fenstergröße anhand der Größe der Widgets ermitteln → Erklärung kommt später.
 - ◆ **shell.setText**: Titelzeile des Fensters eintragen.
 - ◆ **shell.setSize**: Fenstergröße festlegen.

Oberflächenlayout

Eine erste SWT-Anwendung



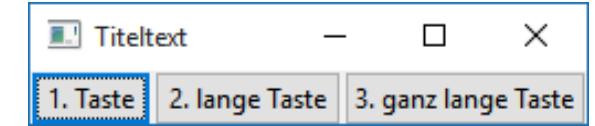
- Der Rest der Anwendung ist die Ereignisschleife →
 - ◆ Wenn Ereignisse vom Betriebssystem vorliegen, dann werden sie in der Anwendung behandelt.
 - ◆ Wenn keine Ereignisse vorliegen, dann legt sich die Anwendung schlafen.
 - ◆ Wenn das Fenster frei gegebene wird (Aufruf von **dispose**), dann beendet sich die Anwendung.



Problem 1: Unterschiedliche Plattformen

Zeichensätze und Widget-Designs unterscheiden sich.

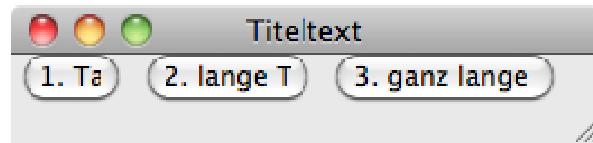
Windows XP, 7, 10:



Ubuntu 13.10 mit Unity, Linux Mint 17:



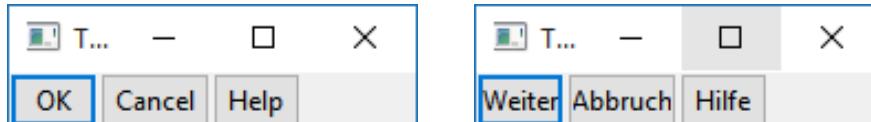
Mac OS X (10.6):





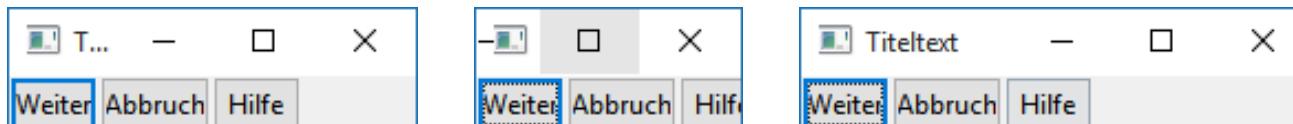
Problem 2: Internationalisierung

- Nachträgliche Übersetzung (Internationalisierung) der Texte in andere Sprachen → Lösung dazu kommt später.
- Ausgaben (Sprachvarianten unter Windows 10):



Problem 3: Interaktive Größenänderungen der Dialoge

- Der Anwender ändert die Fenstergröße interaktiv → der Inhalt passt sich nicht automatisch an (Platzverschwendungen oder „Verschwinden“ von Inhalten):

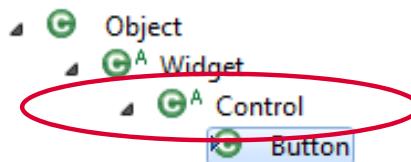




- Konsequenzen:
 - ◆ Plattformunabhängige Programmierung ist mit absoluten Layouts nicht sinnvoll möglich (unterschiedliche Zeichensätze oder Zeichengrößen).
 - ◆ Portierung einer Anwendung in mehrere Sprachen erfordert bei absoluten Layouts manuelle Nacharbeiten → nicht praktikabel.
 - ◆ Niemals ohne Layoutmanager Benutzeroberflächen erstellen!
 - ◆ Auch andere Toolkits arbeiten mit Layoutmanagern: Qt, GTK, Swing, JavaFX, AWT, WPF, ...
- Gute Einführung in die Layoutmanager:
<http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html>



- Frage: Woher kennt ein Layoutmanager die Größen der einzelnen Widgets?
- Lösung:
 - ◆ Die Widgets kennen die Größen. Der Layoutmanager fragt die Widgets ab.
 - ◆ Jedes SWT-Widget erbt von der Basisklasse **Control**, die wiederum von **Widget** erbt.



- ◆ **Control** besitzt die Methode **computeSize**, die die bevorzugte Größe des Widgets berechnet und zurückgibt.
- ◆ Mit **getSize** bzw. **setSize** kann die aktuelle Größe des Widgets ausgelesen bzw. überschrieben werden.
- Die folgenden Beispiele stammen aus dem Projekt **de.hska.iwii.layoute4rcpproject**.



Platzierung der Widgets nacheinander in einheitlicher Größe

- Die Abstände zwischen den Widgets, die äußeren Ränder sowie die Ausrichtung können angegeben werden.
- Alle Widgets erhalten dieselbe Größe.



- Ausrichtung:

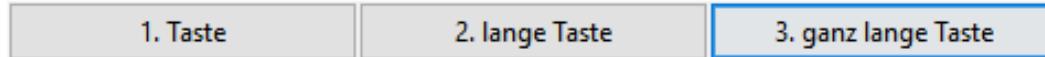
Konstante	Beschreibung
<code>SWT.HORIZONTAL</code>	Die Widgets werden waagerecht angeordnet.
<code>SWT.VERTICAL</code>	Die Widgets werden senkrecht angeordnet.

Oberflächenlayout

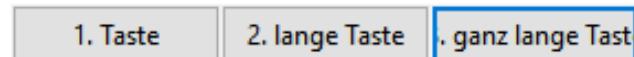


FillLayout

- Beispiel:



horizontale Ausrichtung



manuell verkleinert



vertikale Ausrichtung



FillLayout

- Programmcode (**FillLayoutPart**):

```
...
container.setLayout(new FillLayout(SWT.VERTICAL));
Button button = new Button(container, SWT.PUSH);
button.setText("1. Taste");
button = new Button(container, SWT.PUSH);
button.setText("2. lange Taste");
button = new Button(container, SWT.PUSH);
button.setText("3. ganz lange Taste");
...
```

Einige Eigenschaften von FillLayout

Öffentliches Attribut	Beschreibung
marginWidth , marginHeight	linker und rechter bzw. oberer und unterer Rand
spacing	Abstände zwischen den Widgets



Platzierung der Widgets in bevorzugter oder erzwungener Größe

- Die Abstände zwischen den Widgets, die äußeren Ränder sowie die Ausrichtung können angegeben werden.
- Alle Widgets erhalten ihre bevorzugte oder eine erzwungene Größe.
- Ein automatischer „Zeilenumbruch“ ist möglich.

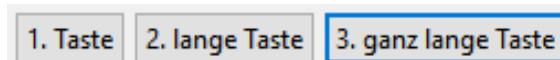




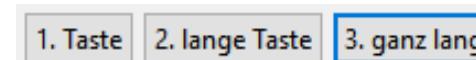
- Ausrichtung:

Konstante	Beschreibung
SWT.HORIZONTAL	Die Widgets werden waagerecht angeordnet.
SWT.VERTICAL	Die Widgets werden senkrecht angeordnet.

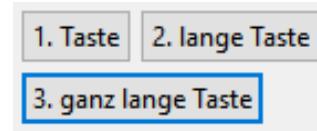
- Beispiel:



horizontale Ausrichtung



manuell verkleinert
(ohne „Zeilenumbruch“)



manuell verkleinert
(mit „Zeilenumbruch“)



vertikale Ausrichtung



- Programmcode (**RowLayoutPart**):

```
...
RowLayout layout = new RowLayout(SWT.HORIZONTAL);
layout.wrap = false; // kein "Zeilenumbruch"
container.setLayout(layout);

Button button = new Button(container, SWT.PUSH);
button.setText("1. Taste");
button = new Button(container, SWT.PUSH);
button.setText("2. lange Taste");
button = new Button(container, SWT.PUSH);
button.setText("3. ganz lange Taste");
...

```



- Programmcode mit der JFace-Klasse **RowLayoutFactory**:

```
...
// erst Tasten wie oben erzeugen
RowLayoutFactory.swtDefaults().type(SWT.HORIZONTAL)
    .wrap(false).applyTo(container);
```



Einige Eigenschaften von RowLayout

Öffentliches Attribut	Beschreibung
center	true : Die Widgets werden bei unterschiedlichen Breiten/Höhen zueinander zentriert
justify	true : Die Widgets werden wie im Blocksatz angeordnet. false : Die Widgets werden am Ursprung ausgerichtet.
marginTop , marginLeft , marginBottom , marginRight	oberer/linker/unterer/rechter Rand
pack	true : bevorzugte Größen der Widgets verwenden false : Alle Widgets erhalten die Maße der größten Komponente.
spacing	Abstand zwischen den Widgets
wrap	true : „Zeilenumbruch“ zulassen



Oberflächenlayout

RowLayout

- Weiteres Konzept für viele Layout-Manager: Einem Widget werden Layout-Eigenschaften übergeben (Aufruf der Methode **setLayoutData**).
- **RowLayout**:
 - ◆ Die Klasse **RowData** nimmt die Layout-Eigenschaften auf.
 - ◆ **RowData** beinhaltet Breite und Höhe des Widgets (überschreibt dessen bevorzugte Größe).
- Beispiel (eine Taste soll eine feste Breite und ihre bevorzugte Höhe erhalten):

```
Button button = new Button(container, SWT.PUSH);
button.setText("1. Taste");
RowData layoutData = new RowData(200, SWT.DEFAULT);
button.setLayoutData(layoutData);
```

feste Breite

bevorzugte Höhe





Einige Eigenschaften von RowData

Öffentliches Attribut	Beschreibung
width, height	neue Breite/Höhe (überschreibt bevorzugte Breite/Höhe)
exclude	Das Widget soll nicht im Layout erscheinen.

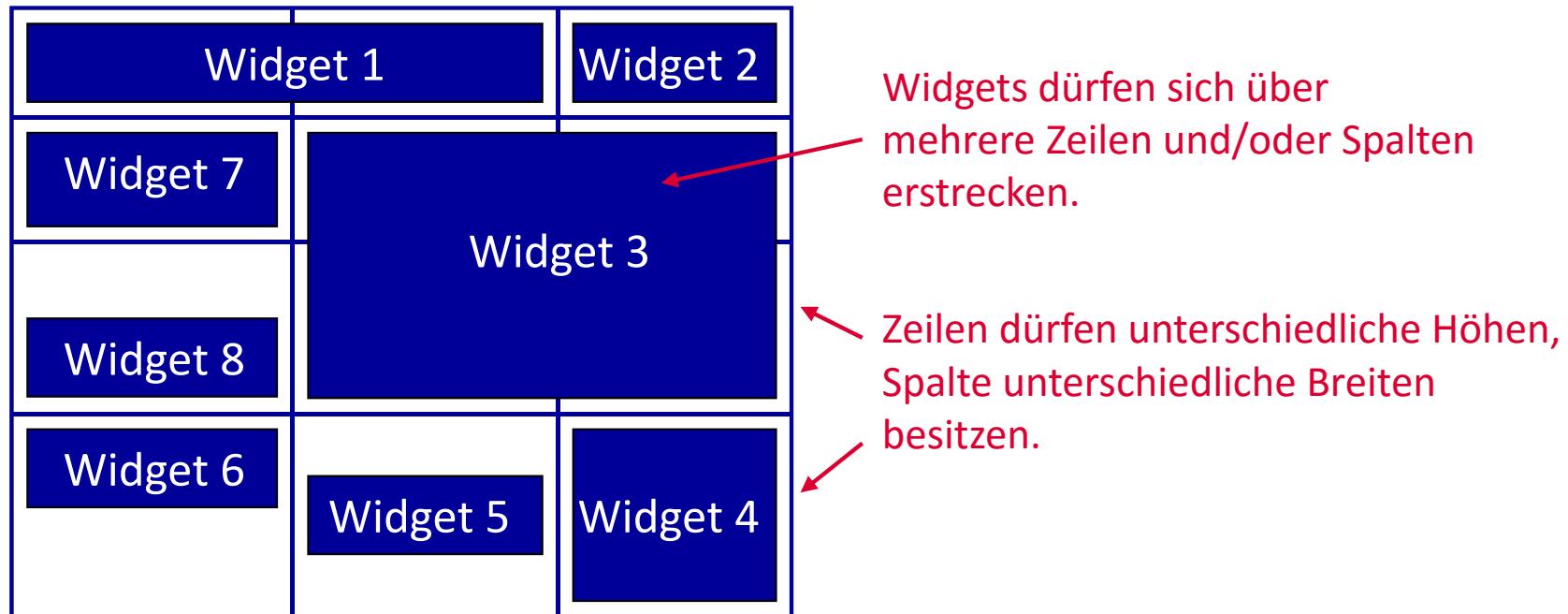
In JFace erlaubt die Klasse **RowLayoutFactory** das einfachere Erzeugen des Layouts.



Oberflächenlayout

GridLayout

- Das **GridLayout** ordnet alle Widgets in einem tabellarischen Raster an.
- Die Zellen können sich über mehrere Zeilen und Spalten erstrecken.
- Die Widgets können über ihre bevorzugte Größe hinaus vergrößert werden, um ihre jeweilige Zelle zu füllen.
- Es kann eine einheitliche Spaltenbreite erzwungen werden.





Einige Eigenschaften von GridLayout

Öffentliches Attribut	Beschreibung
horizontalSpacing , verticalSpacing	horizontaler bzw. vertikaler Abstand zwischen den Zellen
makeColumnsEqualWidth	identische Spaltenbreite erzwingen
marginTop , marginLeft , marginBottom , marginRight	Größe der Einzelränder um das Layout
marginHeight , marginWidth	Größe des oberen und unteren bzw. des linken und rechten Randes
numColumns	Anzahl Spalten im Layout



Oberflächenlayout

GridLayout

- Wie bei **RowLayout** können einem Widget zusätzliche Layout-Eigenschaften übergeben werden.
- **GridLayout:**
 - ◆ Die Klasse **GridData** nimmt die Layout-Eigenschaften auf.
 - ◆ **GridData** beeinflusst das Verhalten des Widgets im Layout.
- Beispiel (die OK-Taste soll sich über die volle Breite und Höhe der Zelle erstrecken, obwohl sie eigentlich kleiner sein möchte):

```
Button button = new Button(container, SWT.PUSH);
button.setText("OK");
data = new GridData(SWT.FILL, SWT.FILL, true, true);
button.setLayoutData(data);
```

Die Taste belegt die
volle Höhe der Zelle
und soll mitwachsen.

Die Taste belegt die volle
Breite der Zelle und soll
mitwachsen.



Einige Eigenschaften von GridData

Öffentliches Attribut	Beschreibung
<code>exclude</code>	<code>true</code> : Die Zelle im Layout ignorieren (nicht anzeigen).
<code>grabExcessHorizontalSpace</code> , <code>grabExcessVerticalSpace</code>	<code>true</code> : Die Zelle (Spalte, Zeile) soll den kompletten horizontalen/vertikalen Platz belegen, wenn das Vaterelement wächst.
<code>widthHint</code> , <code>heightHint</code>	explizit vorgegebene bevorzugte Breite/Höhe des Widgets
<code>horizontalAlignment</code> , <code>verticalAlignment</code>	horizontale/vertikale Ausrichtung des Widgets, wenn es kleiner als die Zelle ist: <code>SWT-BEGINNING</code> , <code>SWT-CENTER</code> , <code>SWT-END</code> , <code>SWT-FILL</code> Mit <code>SWT-FILL</code> wird das Widget auf die Breite der Zelle gestreckt oder gestaucht.
<code>horizontalIndent</code> , <code>verticalIndent</code>	linker bzw. oberer Einzug des Widgets



Öffentliches Attribut	Beschreibung
<code>horizontalSpan, verticalSpan</code>	Anzahl Spalten/Zeilen, die das Widget im Layout belegt
<code>minimumWidth, minimumHeight</code>	minimale Breite/Höhe des Widgets

Spaltenbreite:

- max. bevorzugte Breite des Widgets in der Spalte
- Wenn min. ein Widget `grabExcessHorizontalSpace = true` hat, wird die Spalte so breit wie möglich.

Die Zeilenhöhe wird analog berechnet.

Die Attribute lassen sich auch über den Konstruktor von `GridData` setzen (siehe API-Dokumentation).

In JFace erlauben die Klassen `GridLayoutFactory` und `GridDataFactory` das einfachere Erzeugen des Layouts.



Oberflächenlayout

GridLayout

- Programmcode (**GridLayoutPart**):

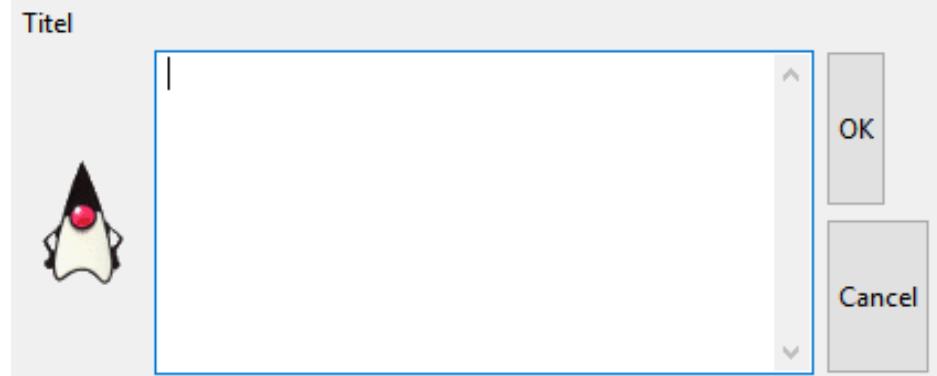
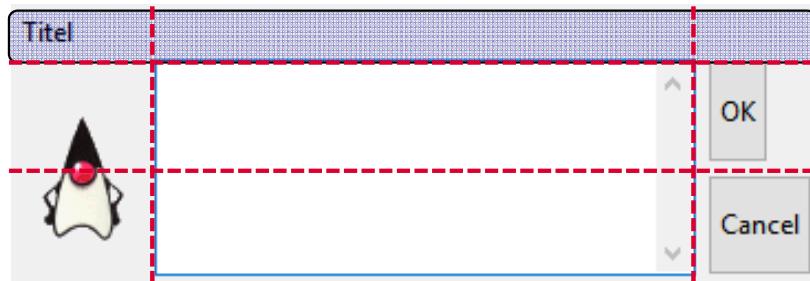
```
...
GridLayout layout = new GridLayout(3, false); // 3 Spalten
container.setLayout(layout);
```

```
Label label = new Label(container, SWT.BEGINNING);
label.setText("Titel");
GridData data = new GridData(SWT.FILL, SWT.DEFAULT, true, false, 3, 1);
label.setLayoutData(data);
```

füllen (h/v)

wachsen (h/v)

Spalten/Zeilen

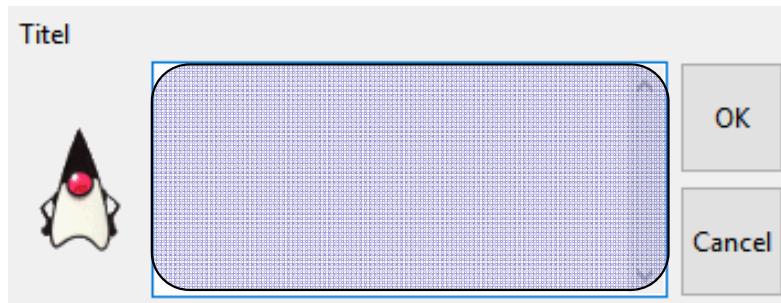




Oberflächenlayout

GridLayout

```
Text text = new Text(container, SWT.MULTI | SWT.WRAP | SWT.BORDER  
                      | SWT.H_SCROLL | SWT.V_SCROLL);  
data = new GridData(SWT.FILL, SWT.FILL, true, true, 1, 2);  
data.widthHint = 200;      // bevorzugte Breite 200 Pixel  
text.setLayoutData(data);
```

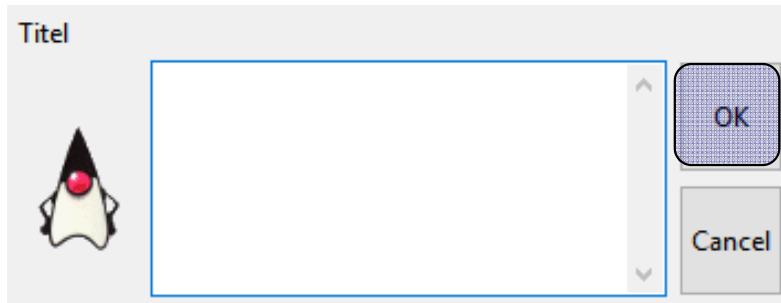




Oberflächenlayout

GridLayout

```
Button button = new Button(container, SWT.PUSH);
button.setText("OK");
data = new GridData(SWT.FILL, SWT.FILL, false, true);
button.setLayoutData(data);
```



Alternativ:

```
Button button = new Button(shell, SWT.PUSH);
button.setText("OK");
data = new GridData(SWT.DEFAULT, SWT.FILL, false, true);
button.setLayoutData(data);
```

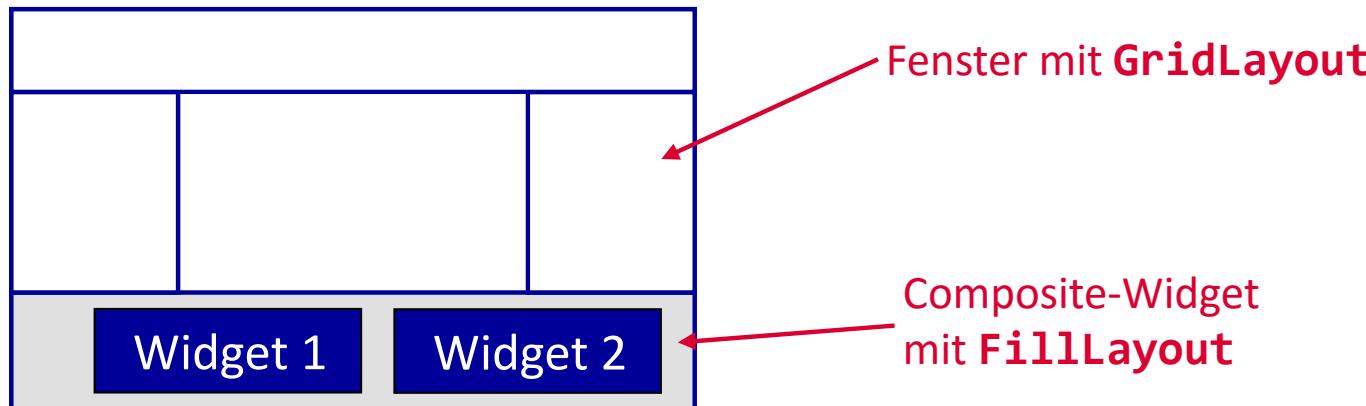




Oberflächenlayout

Verschachtelte Layouts

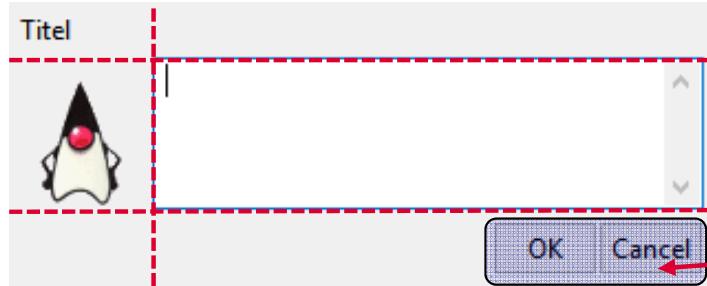
- Bisher wurde einem Fenster immer ein Layoutmanager zugeordnet. Alle Widgets waren direkt im Fenster platziert.
- Dieser Ansatz ist für komplizierte Layouts nicht flexibel genug.
- Ergänzung: Als Widgets können auch andere Container eingesetzt werden, die ihrerseits wiederum einen eigenen Layoutmanager besitzen und Widgets aufnehmen können → hierarchische Schachtelung.



- **Composite:**
 - ◆ Container für andere Widgets
 - ◆ Hier: keine eigene Darstellung



- Beispiel: ein Dialog mit zwei nebeneinander angeordneten, identisch breiten Tasten (**NestedLayoutPart**)



Composite-Widget mit FillLayout
(der Rahmen zeigt die Ausmaße
des Composite-Objektes)

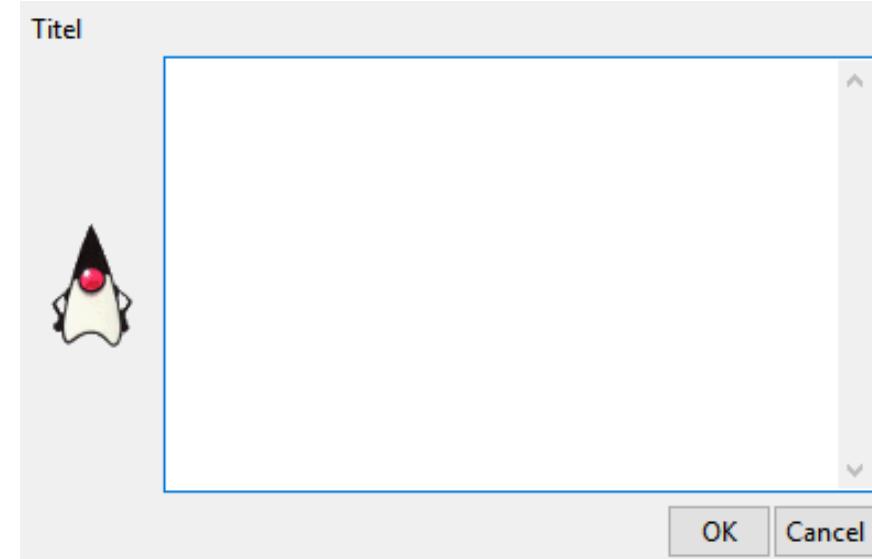
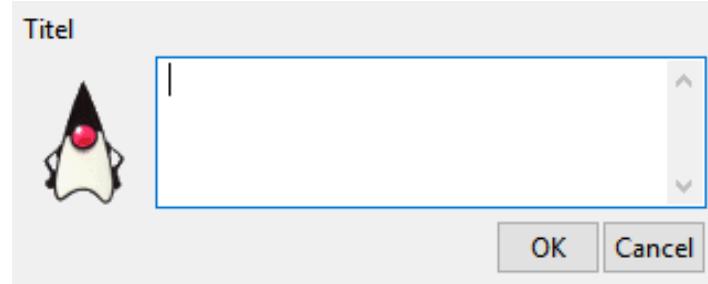
```
Composite buttonPanel = new Composite(container, SWT.NONE);
buttonPanel.setLayout(new FillLayout());
data = new GridData(SWT.END, SWT.DEFAULT, false, false, 2, 1);
buttonPanel.setLayoutData(data);
new Button(buttonPanel, SWT.PUSH).setText("OK");
new Button(buttonPanel, SWT.PUSH).setText("Cancel");
```

Oberflächenlayout

Verschachtelte Layouts



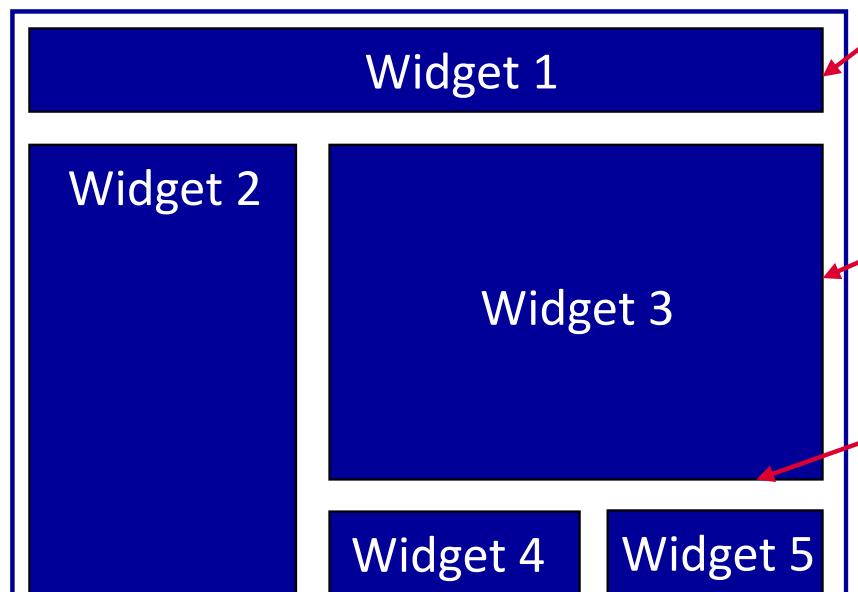
- Verhalten bei Größenänderungen und Aussehen unter Windows:





Flexible Gestaltung aufwändiger Dialoge

- **FormLayout** ist der flexibelste und komplexeste Standardlayoutmanager im SWT.
- **FormLayout** platziert die Widgets relativ zueinander.



Widget 1 wird mit seinem linken und rechten Rand am Fenster ausgerichtet.

Widget 3 wird mit seinem rechten Rand immer am Rand des Fenster ausgerichtet.

Widget 3 wird mit seinem unteren Rand am oberen Rand von Widget 5 ausgerichtet.

Widget 5 wird mit seinem unteren Rand immer am Fenster ausgerichtet.



Einige Eigenschaften von FormLayout

Öffentliches Attribut	Beschreibung
spacing	Abstand zwischen den Zellen
marginTop, marginLeft, marginBottom, marginRight	Größe der Einzelränder um das Layout
marginHeight, marginWidth	Größe des oberen und unteren bzw. des linken und rechten Randes



- Wie bei **GridLayout** können einem Widget zusätzliche Layout-Eigenschaften übergeben werden.
- **FormLayout**:
 - ◆ Die Klasse **FormData** nimmt für die vier Seiten die Platzierungsbeziehungen auf (Klasse **FormAttachment**).

Öffentliches Attribut	Beschreibung
top, left, bottom, right	Platzierungsregeln für den oberen/linken/unteren/rechten Rand
width, height	manuell vorgegebene bevorzugte Breite/Höhe des Widgets



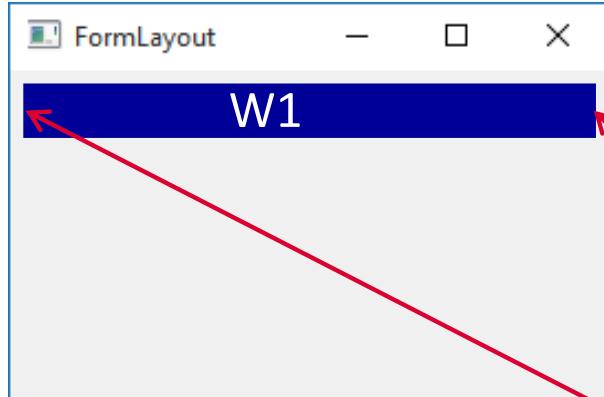
- ◆ **FormAttachment** beschreibt eine Platzierung relativ zu einem anderen Widget oder zum Vaterobjekt.

Öffentliches Attribut	Beschreibung
alignment	Seite des Zielwidgets, an dem ausgerichtet werden soll. Für die obere oder untere Seite: SWT.TOP , SWT.CENTER , SWT.BOTTOM Für die linke oder rechte Seite: SWT.LEFT , SWT.CENTER , SWT.RIGHT
control	Zielwidget, an dem ausgerichtet werden soll
numerator , denominator (Standardwert 100)	Gilt nur bei Ausrichtung am Vaterobjekt. Die Position einer eigenen Seite berechnet sich wie folgt: $pos = fract * size + offset$ $fract = numerator/denominator$ $size = \text{Breite bzw. Höhe des Vaterobjektes}$
offset	absoluter Abstand in Pixeln vom Zielwidget



Beispiele zur Erläuterung

- Widget **W1** links und rechts am Fenster ausrichten (wächst in der Breite mit):



```
FormData data = new FormData();

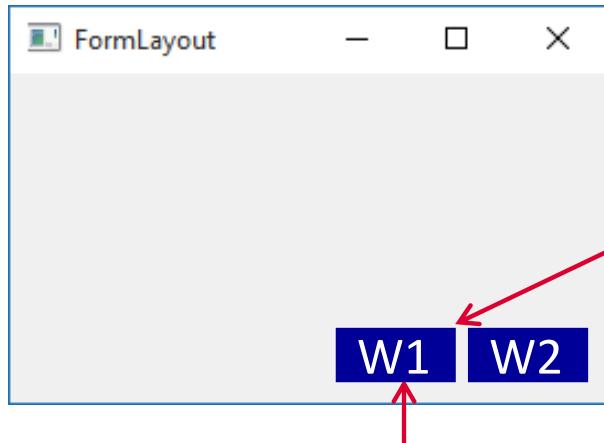
// Der rechte Rand sitzt bei 100% Breite
// des Fensters (100/100). Der Offset
// ist 0.
data.right = new FormAttachment(100);

// Der linker Rand sitzt bei 0% Breite
// des Fensters (0/100). Der Offset
// ist 0.
data.left = new FormAttachment(0);

w1.setLayoutData(data);
```



- Widget **W2** rechts an **W1** ausrichten:



```
FormData data = new FormData();

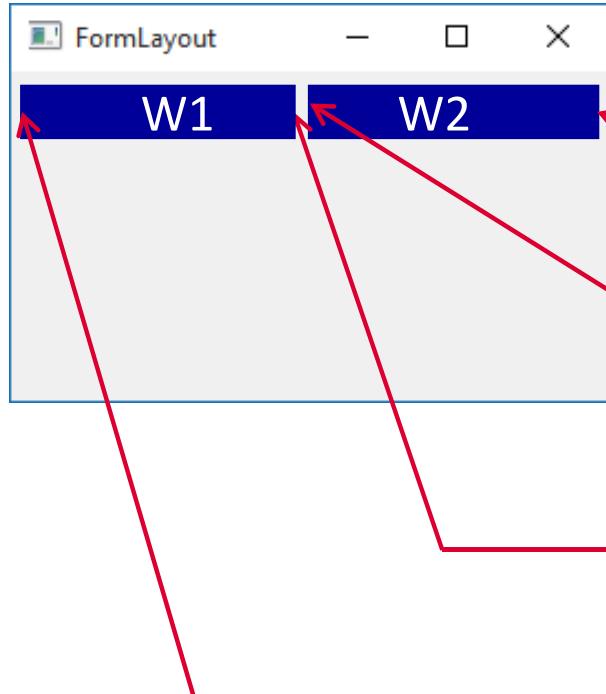
// Der rechte Rand schließt direkt an
// den linken Rand von W1 an. Der Abstand
// wird durch das Layout vorgegeben.
data.right = new FormAttachment(w2);

// Der untere Rand sitzt bei 100% Höhe
// des Fensters (100/100). Der Offset
// ist 0.
data.bottom = new FormAttachment(100);

w1.setLayoutData(data);
```



- **W1** und **W2** belegen jeweils 50% der Breite des Fensters und haben in der Mitte einen zusätzlichen Abstand von 4 Pixeln.



```
FormData w1Data = new FormData();
FormData w2Data = new FormData();

// Der rechte Rand von W2 liegt am Fenster.
w2Data.right = new FormAttachment(100);

// Der linke Rand von W2 liegt mit 4 Pixeln
// Abstand rechts von W1.
w2Data.left = new FormAttachment(w1, 4);

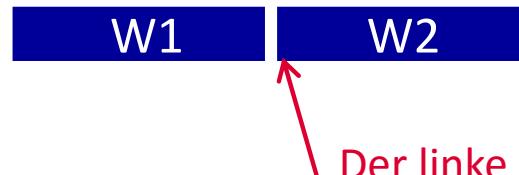
// Der rechte Rand von W1 liegt bei 50% der
// Breite (minus 2 Pixel).
w1Data.right = new FormAttachment(50, -2);

// Der linke Rand von W1 liegt am Fenster.
w1Data.left = new FormAttachment(0);

w1.setLayoutData(w1Data);
```

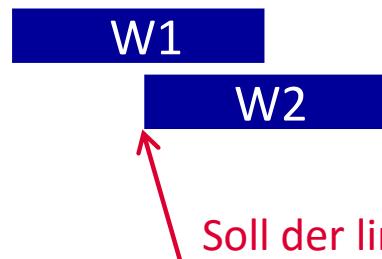


- Die Angabe des **alignment** fehlt in den Beispielen, weil hier immer die eindeutig benachbarten Ränder genommen werden.
 - Rechter Rand von **W1** hat den linken von **W2** als Nachbarn → wird automatisch erkannt und muss daher nicht angegeben werden.



Der linke Rand von **W2** wird an dem rechten Rand von **W1** ausgerichtet.

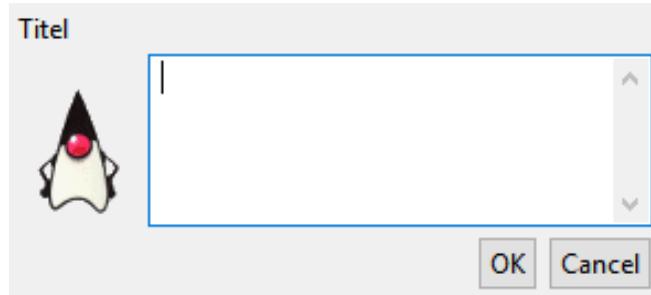
- Wenn es nicht eindeutig ist, muss der Rand spezifiziert werden:



Soll der linke Rand von **W2** an dem rechten oder linken Rand von **W1** ausgerichtet werden?



- Beispiel (**FormLayoutPart**) :

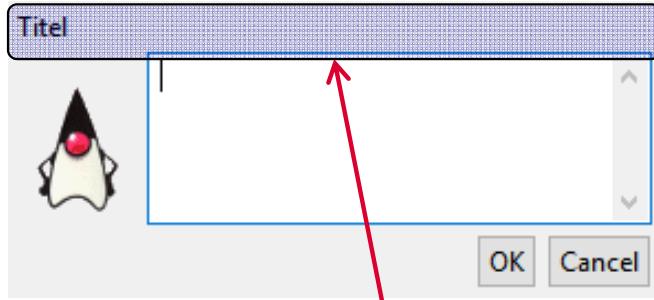


Anmerkung: Für Dialoge mit den Standardtasten gibt es spezielle Dialog-Klassen.

```
FormLayout layout = new FormLayout();
layout.marginHeight = 4; // oberer und unterer Rand
layout.marginWidth = 4; // linker und rechter Rand
layout.spacing      = 4; // Abstände zwischen den Komponenten
container.setLayout(layout);
```

Oberflächenlayout

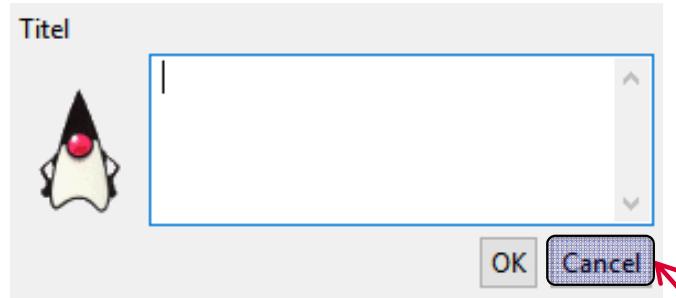
FormLayout



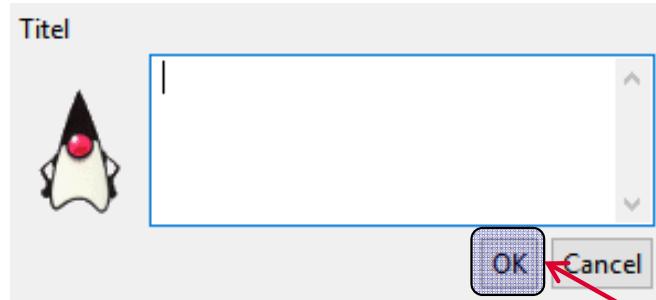
```
Label label = new Label(container, SWT.BEGINNING | SWT.BORDER);
label.setText("Titel");
FormData data = new FormData();
data.left = new FormAttachment(0); // linker Rand bei 0% Fensterbreite
data.right = new FormAttachment(100); // rechter Rand bei 100% Fensterbr.
label.setLayoutData(data);
```

Oberflächenlayout

FormLayout



```
Button cancelButton = new Button(container, SWT.PUSH);
cancelButton.setText("Cancel");
data = new FormData();
data.right  = new FormAttachment(100); // rechter Rand bei 100% Fensterbreite
data.bottom = new FormAttachment(100); // unterer Rand bei 100% Fensterhöhe
cancelButton.setLayoutData(data);
```

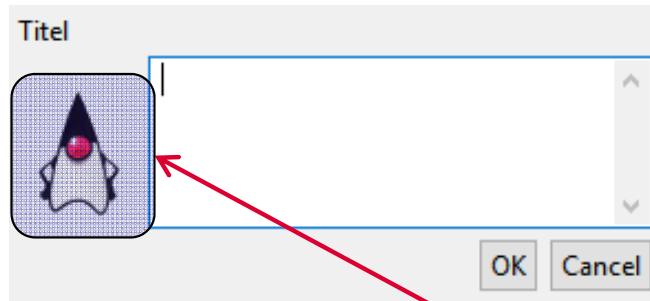


```
Button okButton = new Button(container, SWT.PUSH);
okButton.setText("OK");
data = new FormData();
data.right = new FormAttachment(cancelButton); // rechten Rand an der
                                                // Cancel-Taste ausrichten
data.bottom = new FormAttachment(100);          // unterer Rand bei 100%
                                                // Fensterbreite
okButton.setLayoutData(data);
```

Identische Tastenbreiten → **FillLayout** für die Tasten verwenden!

Oberflächenlayout

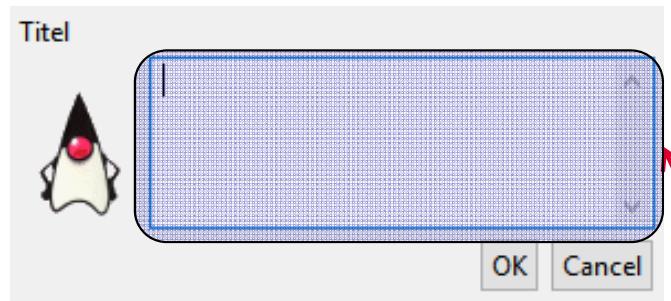
FormLayout



```
Label dukeLabel = new Label(container, SWT.CENTER | SWT.BORDER);
dukeLabel.setImage(imageRegistry.get("duke"));
data = new FormData();
data.left   = new FormAttachment(0);                      // linker Rand bei 0%
                                                       // Fensterbreite
data.top    = new FormAttachment(label);                  // oberen Rand am
                                                       // Titel-Label ausrichten
data.bottom = new FormAttachment(cancelButton);          // unteren Rand an der
                                                       // Cancel-Taste ausrichten
dukeLabel.setLayoutData(data);
```

Oberflächenlayout

FormLayout



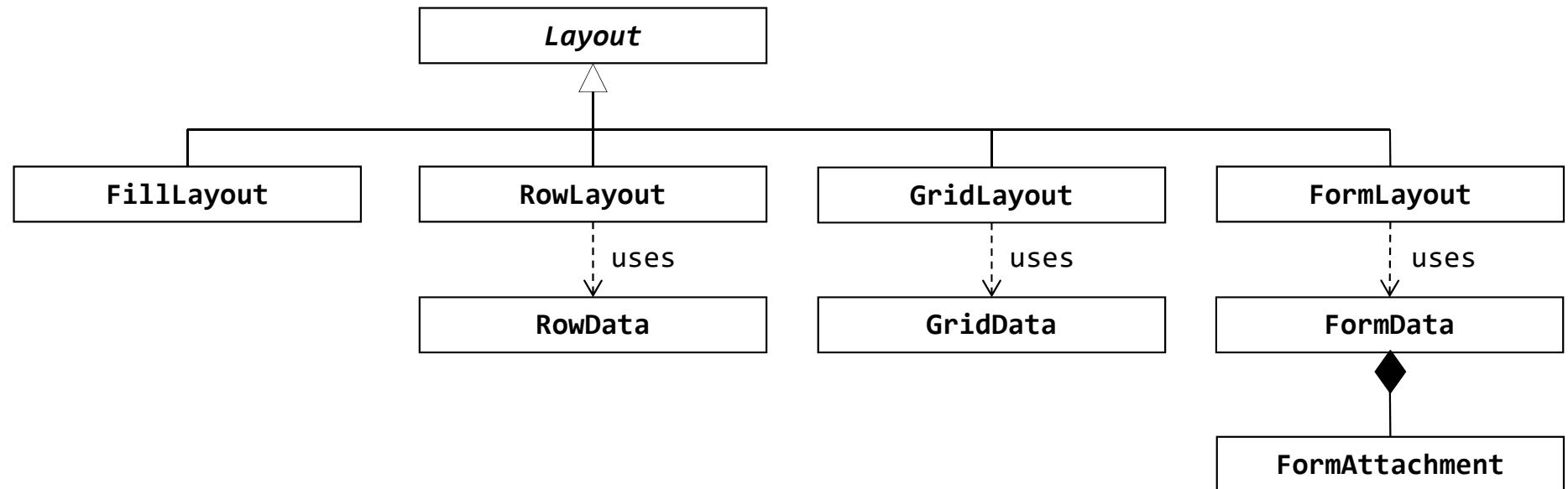
```
Text text = new Text(container, SWT.MULTI | SWT.WRAP | SWT.BORDER  
                     | SWT.H_SCROLL | SWT.V_SCROLL);  
  
data = new FormData();  
data.left   = new FormAttachment(dukeLabel);      // linken Rand am  
             // Duke-Label ausrichten  
data.top    = new FormAttachment(label);          // oberen Rand am  
             // Titel-Label ausrichten  
data.right  = new FormAttachment(100);            // rechter Rand ist bei  
             // 100 %Fensterbreite  
data.bottom = new FormAttachment(cancelButton); // unteren Rand an der  
             // Cancel-Taste ausrichten  
  
text.setLayoutData(data);
```



- Anmerkung zum Beispiel:
 - ◆ Dialoge (mit Tasten) sollten von einer der vordefinierten Dialogklassen erben, weil dann die Tasten
 - automatisch platziert
 - und anhand der Look&Feel-Vorgaben der Plattform ausgerichtet werden.
 - ◆ Einige Dialogklassen werden später vorgestellt.



- Übersicht über die hier vorgestellten Layouts:





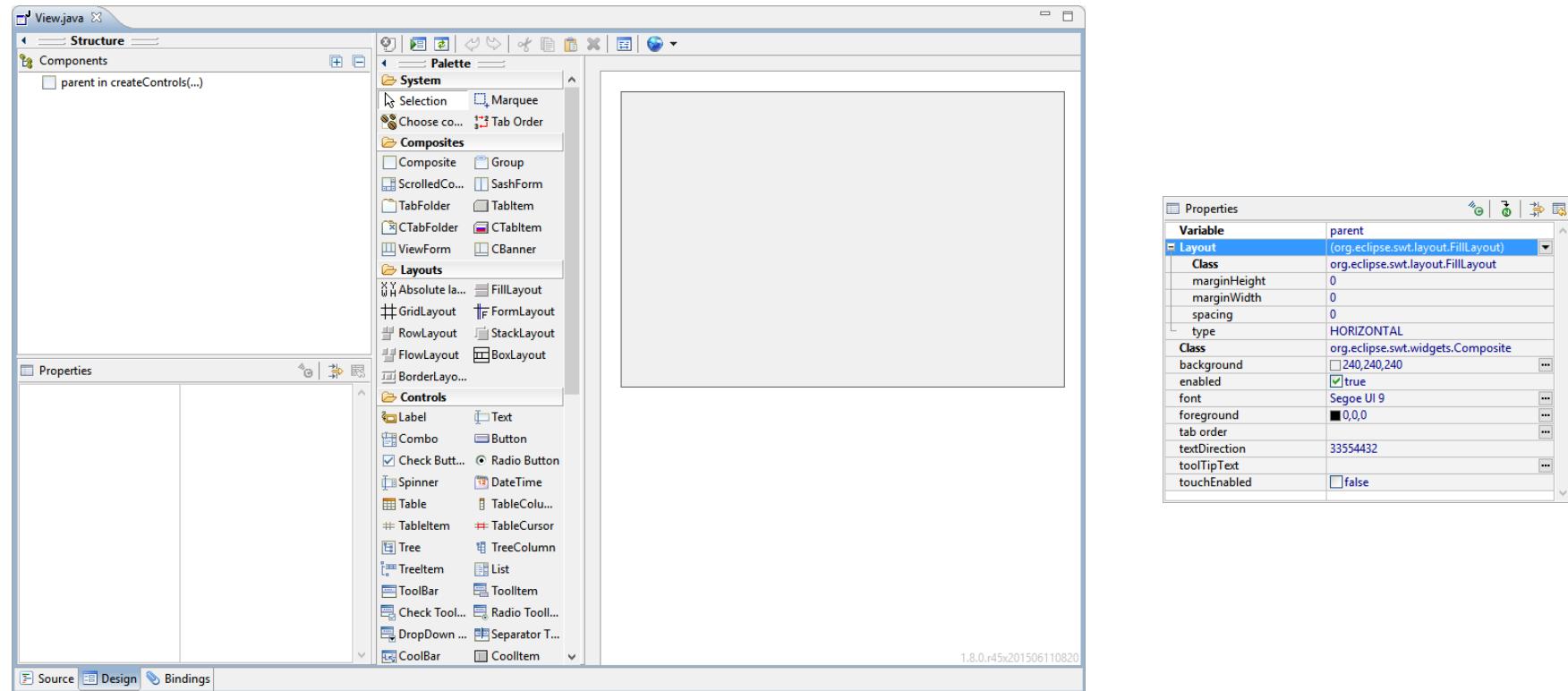
- Die manuelle Anwendung eines Layouts kann mühsam sein.
- Daher: Blick auf die interaktive Konstruktion einer Oberfläche in einer IDE.
- Die beste Unterstützung bietet der WindowBuilder Pro (Google, frei verfügbar, direkt nachinstallierbar).
- Hier können die Widgets visuell den Zellen zugeordnet bzw. Widgets untereinander ausgerichtet werden.
- Ein separater Editor erlaubt die Änderung von Attributen eines Widgets oder der Layout-Daten.

Oberflächenlayout

Interaktive Werkzeuge

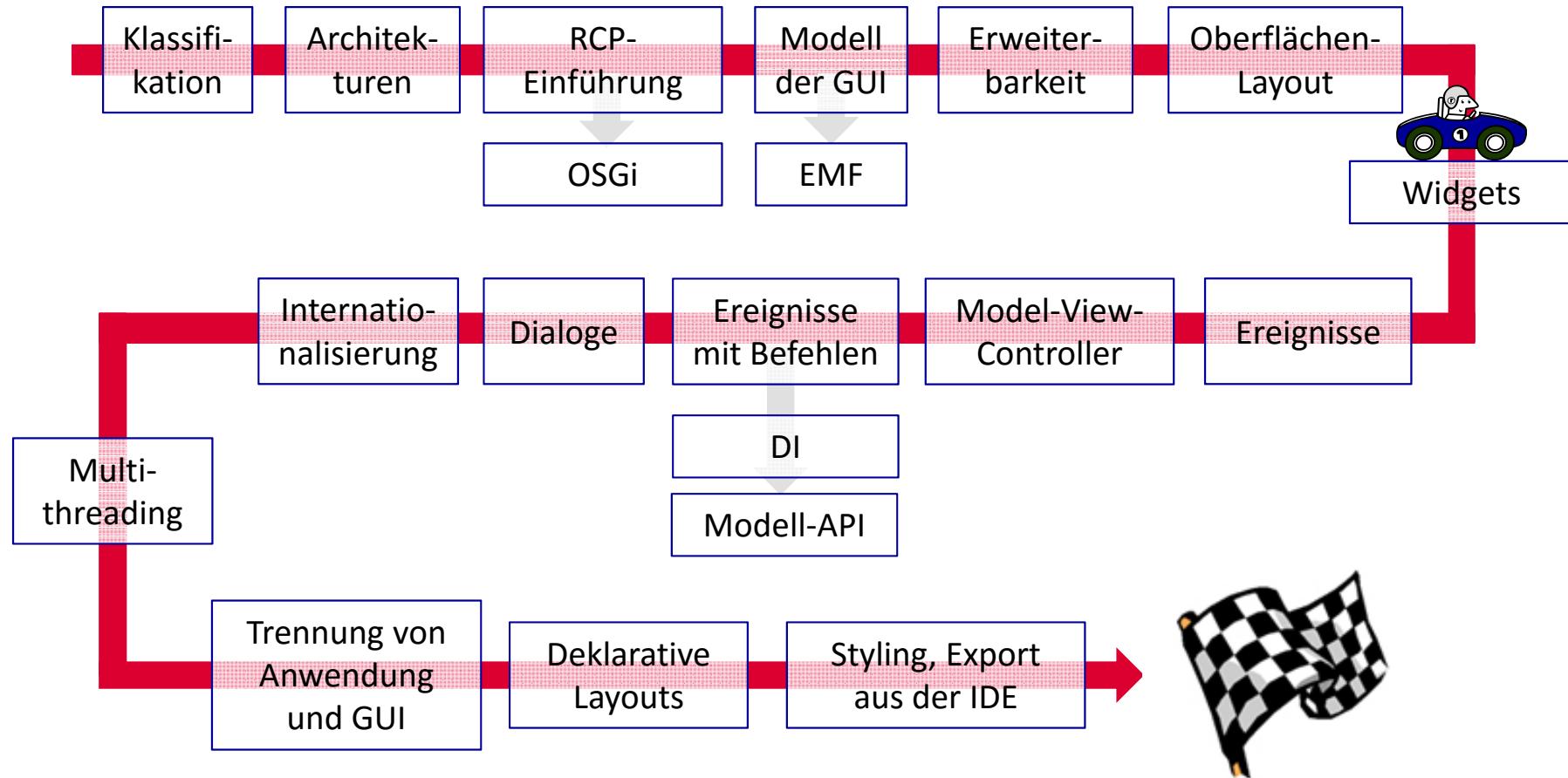


- Ansicht des WindowBuilder-Plug-ins für Eclipse:



Widgets

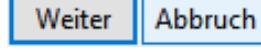
Übersicht



Widgets

Übersicht



Name	Verhalten	Verwendung
Button (SWT.PUSH) 	Kann gedrückt werden. Der Selektionszustand bleibt nicht erhalten.	Auslösen einer Aktion.
Button (SWT.FLAT) 	wie SWT.PUSH , eventuell andere Rahmendarstellung	wie SWT.PUSH
Button (SWT.ARROW SWT.LEFT) und alle anderen Richtungen 	wie SWT.PUSH , andere Darstellung	wie SWT.PUSH
Button (SWT.CHECK) 	Kann selektiert und deselektiert werden. Der Selektionszustand bleibt bis zur Änderung erhalten.	Wahl einer Belegung ohne Beeinflussung anderer Wahlmöglichkeiten.

Widgets

Übersicht

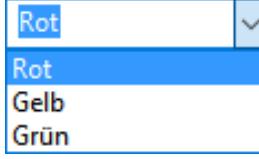
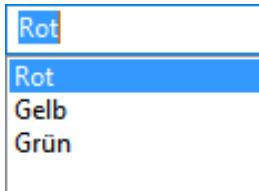


Name	Verhalten	Verwendung
Button (SWT.RADIO) 	Kann selektiert werden. Der Selektionszustand bleibt erhalten, bis eine andere Taste derselben Gruppe gewählt wird.	Wahl einer Belegung aus einer Menge von Werten. Die Anzahl der Werte sollte gering sein.
Button (SWT.TOGGLE) 	Kann dauerhaft selektiert werden.	Wahl einer Belegung. Häufig zusammen mit der Auslösung einer Aktion
Label 	Keines.	Bezeichnung und/oder Bild (z.B. für ein anderes Widget)
Text 	Freitexteingabefeld mit Cursorsteuerung und Selektion, keine variable Textattribute.	Eingabe eines ein- oder mehrzeiligen unformatierten, Textes, Passworteingabe

Widgets

Übersicht

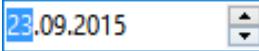
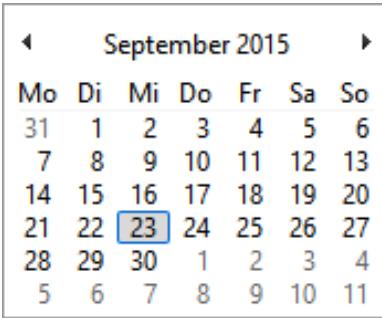


Name	Verhalten	Verwendung
StyledText 	Freitexteingabefeld mit Cursorsteuerung und Selektion, variable Textattribute.	Eingabe eines mehrzeiligen formatierten, Textes
Combo (SWT.DROP_DOWN) 	Zeigt den ausgewählten Eintrag und eine Liste weiterer möglicher Einträge an. Die Auswahlliste ist normalerweise verborgen.	Wahl eines Eintrags aus einer Liste oder freie Eingabe.
Combo (SWT.SIMPLE) 	Zeigt den ausgewählten Eintrag und eine Liste weiterer möglicher Einträge an. Die Auswahlliste ist immer angezeigt.	Wahl einer Belegung. Häufig zusammen mit der Auslösung einer Aktion
Slider 	Kann horizontal oder vertikal verschoben werden und an best. Markierungen einrasten.	Auswahl eines ganzzahligen Wertes aus einem Zahlenbereich.

Widgets

Übersicht



Name	Verhalten	Verwendung
DateTime (SWT.DATE) 	Manuelle Eingabe oder Auswahl durch Tasten	Anzeige und Auswahl eines Datums.
DateTime (SWT.TIME) 	Manuelle Eingabe oder Auswahl durch Tasten	Anzeige und Auswahl einer Uhrzeit.
DateTime (SWT.CALENDAR) 	Auswahl in einem Kalender	Anzeige und Auswahl eines Datums.

Weitere Datums- und Zeitfunktionen sind über Konstanten wählbar.



Name	Verhalten	Verwendung
Group 	Zeigt normalerweise mehrere Radio-Tasten an, um eine Belegung auszuwählen. Group ist ein Composite -Element mit eigenem Aussehen und Layout.	Wahl eines Eintrags aus einer Menge von Einträgen.
ScrollBar 	Wird mit <code>getVerticalBar()</code> bzw. <code>getHorizontalBar()</code> aus einer Komponente ausgelesen	Verschieben des zu großen Inhaltes einer Komponente (z.B. Text), wird über den Stil der Komponente eingeschaltet
ProgressBar (SWT.SMOOTH) 	passiv, wird programmgesteuert verändert, kontinuierliche Anzeige	Fortschrittsanzeige für eine länger laufende Operation für eine bekannte Anzahl Schritte
ProgressBar (SWT.INDETERMINATE) 	passiv, wird programmgesteuert verändert	Fortschrittsanzeige für eine länger laufende Operation für eine unbekannte Anzahl Schritte

Widgets

Übersicht



Name	Verhalten	Verwendung								
Table  <table border="1"><thead><tr><th>Auswahl</th><th>Spalte 1</th></tr></thead><tbody><tr><td><input type="checkbox"/> Zeilenwert 0</td><td>Zeilenwert 1</td></tr><tr><td><input type="checkbox"/> Zeilenwert 3</td><td>Zeilenwert 4</td></tr><tr><td><input type="checkbox"/> Zeilenwert 6</td><td>Zeilenwert 7</td></tr></tbody></table>	Auswahl	Spalte 1	<input type="checkbox"/> Zeilenwert 0	Zeilenwert 1	<input type="checkbox"/> Zeilenwert 3	Zeilenwert 4	<input type="checkbox"/> Zeilenwert 6	Zeilenwert 7	Zeigt Daten tabellarisch an, erlaubt die Angabe von Selektionskriterien.	Tabellarische Darstellung und Eingabe
Auswahl	Spalte 1									
<input type="checkbox"/> Zeilenwert 0	Zeilenwert 1									
<input type="checkbox"/> Zeilenwert 3	Zeilenwert 4									
<input type="checkbox"/> Zeilenwert 6	Zeilenwert 7									
Tree  <ul style="list-style-type: none">▼ Knoten 0<ul style="list-style-type: none">➢ Knoten 0 - 0➢ Knoten 0 - 1➢ Knoten 0 - 2➢ Knoten 0 - 3➢ Knoten 0 - 4	Baum, bei dem sich die einzelnen Knoten auf- und zuklappen lassen	Auswahl und Navigation in einer baumartigen Datenstruktur (z.B. Dateisystem, XML)								
List  <table border="1"><tbody><tr><td>Eintrag 0</td></tr><tr><td>Eintrag 1</td></tr><tr><td>Eintrag 2</td></tr><tr><td>Eintrag 3</td></tr><tr><td>Eintrag 4</td></tr></tbody></table>	Eintrag 0	Eintrag 1	Eintrag 2	Eintrag 3	Eintrag 4	scrollbare Auswahl aus einer vorgegebenen Anzahl Möglichkeiten	einfache oder mehrfache Auswahl von Elementen aus einer Menge			
Eintrag 0										
Eintrag 1										
Eintrag 2										
Eintrag 3										
Eintrag 4										



Name	Verhalten	Verwendung
Browser 	Mit Ereignisverwaltung kann ein installierter Browser eingebettet und zur Navigation verwendet werden.	HTML-Hilfe, Besuch von Web-Seiten
Spinner 	Eingabe durch manuelles Eintippen oder durch Auswahl über die Pfeiltasten	Eingabe eines numerischen Wertes
Link 	Zeigt einen Text an, innerhalb dessen der auswählbare Teil mit <A>Text markiert ist.	wie eine Taste (Button)
Scale 	Kann horizontal oder vertikal verschoben werden und an best. Markierungen einrasten.	Auswahl eines ganzzahligen Wertes aus einem Zahlenbereich.



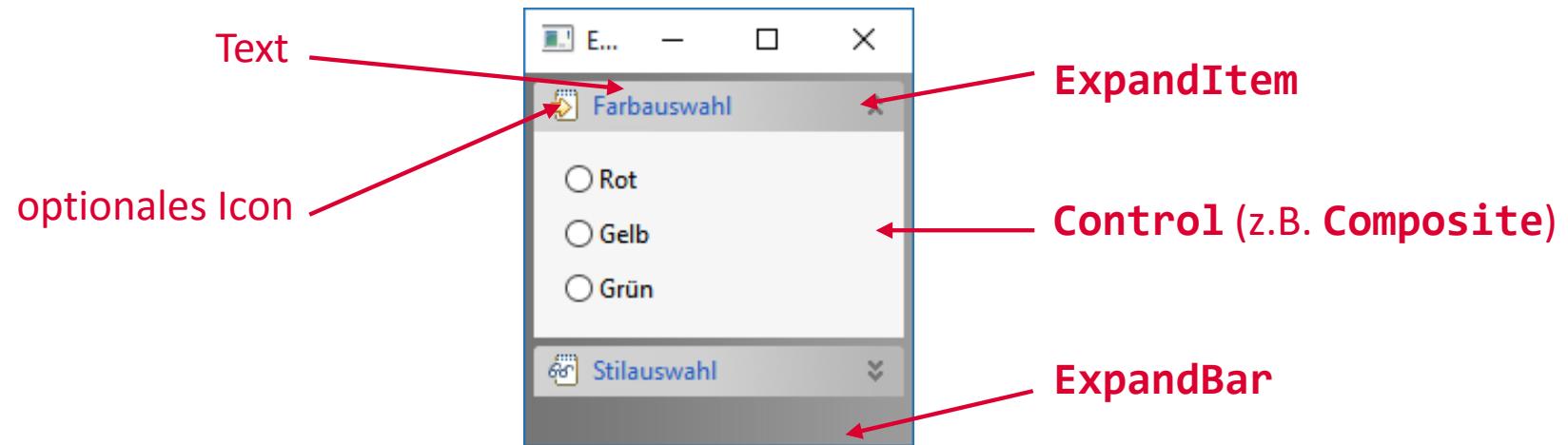
Daneben existieren weitere Widgets im Paket **org.eclipse.swt.custom**:

- **CCombo** statt **Combo**: passt in eine Tabellenzelle
- **CLabel** statt **Label**: gleichzeitige Darstellung von Bild und Text, Farbverlauf in Hintergrund, Hintergrundbild, automatische Textkürzung
- **CTabFolder/CTabItem** statt **TabFolder/TabItem**: unterschiedliches Aussehen, optionale Schließtaste, Farbverlauf in Hintergrund, Hintergrundbild
- **ScrolledComposite**: **Composite**, das seinen Inhalt scrollen kann
- **SashForm**: Anordnung zweier Widgets neben- oder untereinander, die Größe lässt sich durch einen Trennstrich einstellen
- **ControlEditor**: erlaubt die Eingabe in ein Widget, das ansonsten keine interaktive Eingabe erlaubt (siehe Beispiel **VirtualEditableTableSWTApplication** anhand des speziellen Editors **TableEditor** für Tabellen)



Aufgaben und Eigenschaften

- Eine **ExpandBar** kann mehrere Widgets platzsparend vertikal anordnen.





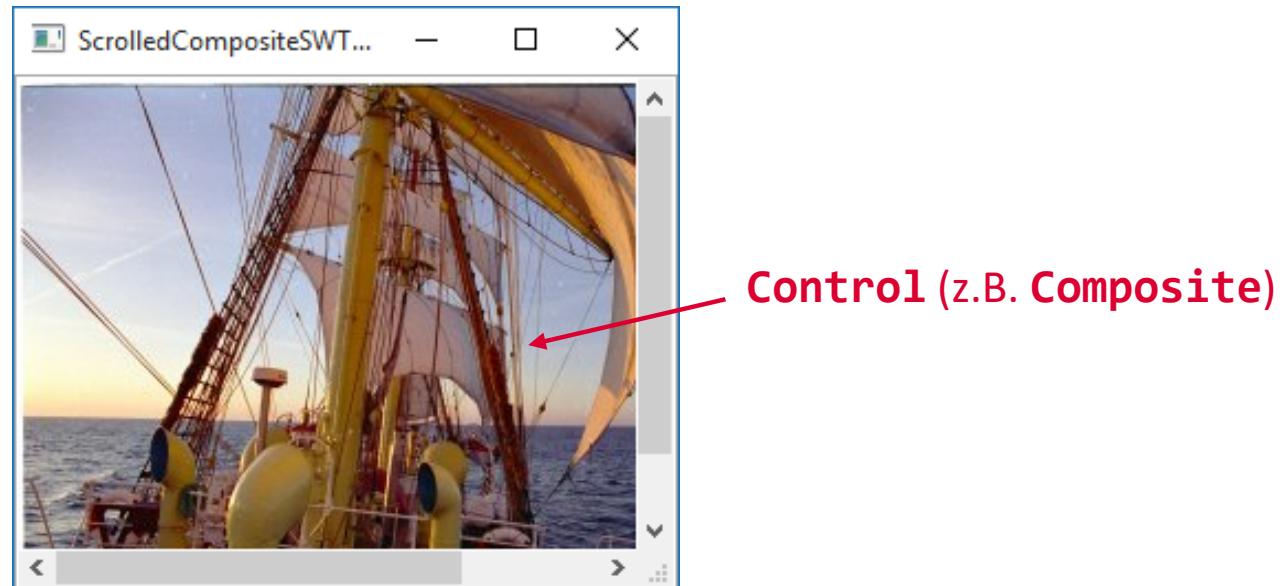
- Verwendung anhand eines Beispiels:

```
// FillLayout ermöglicht es dem ExpandBar, sich den vollen  
// Platz in der Höhe zu nehmen.  
parent.setLayout(new FillLayout());  
  
ExpandBar bar = new ExpandBar(parent, SWT.V_SCROLL);  
  
// 1. Tab erzeugen  
Composite composite = new Composite(bar, SWT.NONE);  
// composite belegt den Inhalt des ersten Tab.  
// Er wird hier gefüllt.  
// ...  
  
// Den Inhalt über ein ExpandItem zum ExpandBar hinzufügen.  
ExpandItem expandItem = new ExpandItem(bar, SWT.NONE);  
expandItem.setText("Farbauswahl");  
expandItem.setImage(image1);  
// Größe manuell setzen, sonst hat der Eintrag die Höhe 0!  
expandItem.setHeight(composite.computeSize(SWT.DEFAULT, SWT.DEFAULT).y);  
expandItem.setControl(composite);
```



Aufgaben und Eigenschaften

- Ein **ScrolledComposite** kann eine Komponente scrollen, falls diese zu groß für einen bestimmten Platz ist.
- Hinweis: Ein **ScrolledComposite** kann die beiden Scrollbalken nach Bedarf ein- und ausschalten.





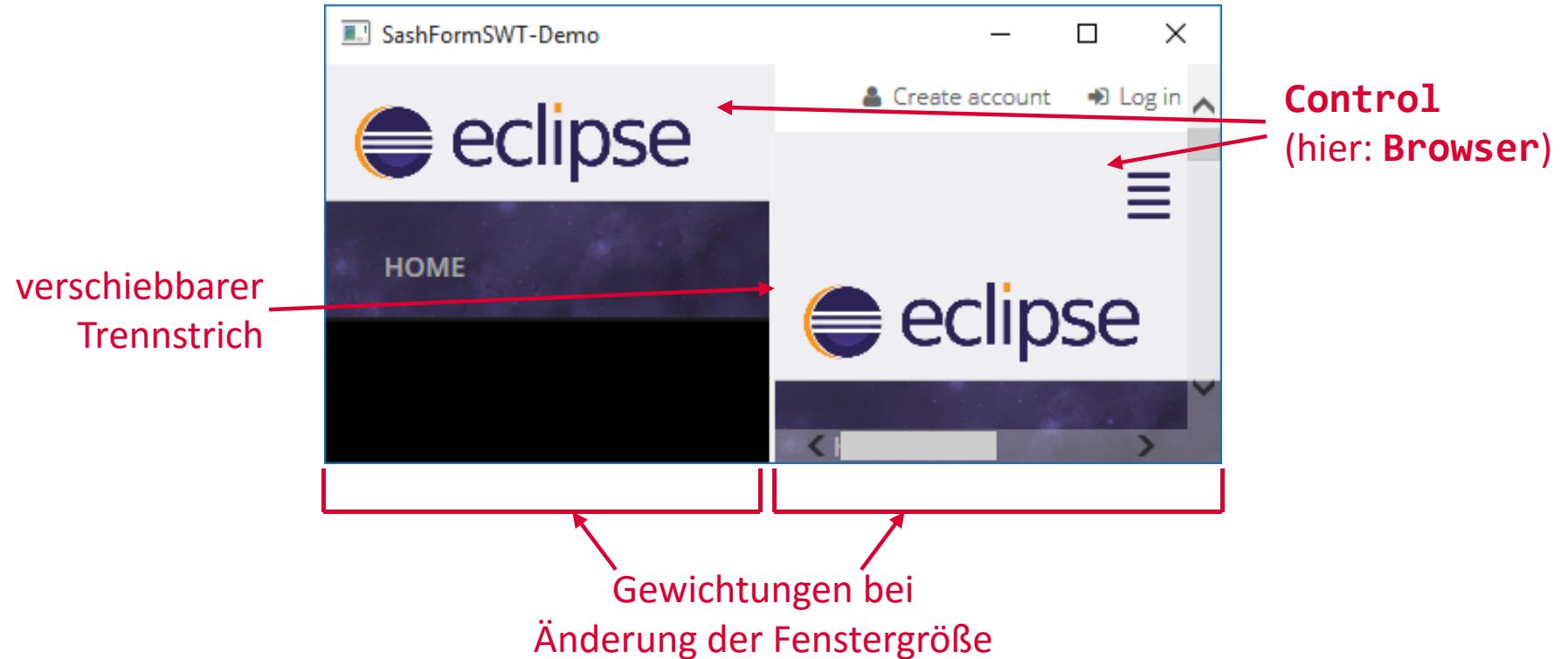
- Quellcode-Ausschnitt:

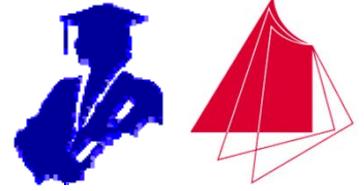
```
parent.setLayout(new FillLayout());  
  
ScrolledComposite scroller = new ScrolledComposite(parent,  
        SWT.H_SCROLL | SWT.V_SCROLL | SWT.BORDER);  
Composite contents = new Composite(scroller, SWT.NONE);  
scroller.setContent(contents);  
contents.setLayout(new FillLayout());  
  
Label label = new Label(contents, SWT.NONE);  
label.setImage(image);  
  
// Größe des Labels manuell berechnen  
contents.setSize(contents.computeSize(SWT.DEFAULT, SWT.DEFAULT));
```



Aufgaben und Eigenschaften

- Eine **SashForm** kann mehrere Widgets horizontal oder vertikal anordnen.
- Der Platz für die Widgets kann durch den Teiler interaktiv eingestellt werden.





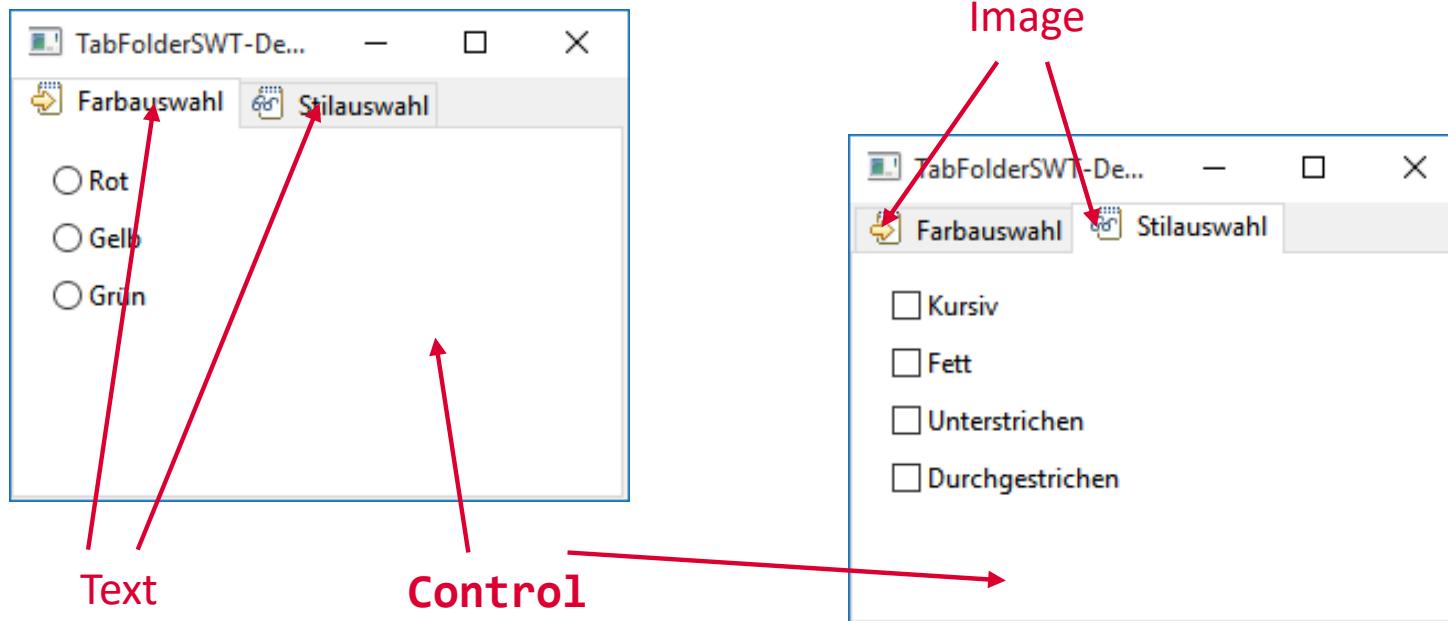
- Quellcode-Ausschnitt:

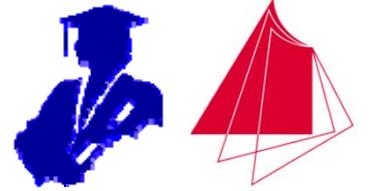
```
parent.setLayout(new FillLayout());  
  
SashForm form = new SashForm(parent, SWT.HORIZONTAL);  
  
// 1. Eintrag  
Browser browser = new Browser(form, SWT.NONE);  
browser.setUrl("http://www.heise.de");  
browser.setSize(150, 100);  
  
// 2. Eintrag  
browser = new Browser(form, SWT.NONE);  
browser.setUrl("http://www.heise.de");  
browser.setSize(150, 100);
```



Aufgaben und Eigenschaften

- Ein **TabFolder** (bzw. **CTabFolder**) kann beliebig viele Widgets hintereinander anordnen.
- Es kann nur ein Widget zur Zeit sichtbar sein.
- Ein **CTabFolder** kann auch Schließtasten in den einzelnen Einträgen besitzen.





- Quellcode-Ausschnitt:

```
parent.setLayout(new FillLayout());  
  
TabFolder folder = new TabFolder(parent, SWT.TOP);  
  
// 1. TabItem erzeugen  
Composite composite = new Composite(folder, SWT.NONE);  
// Composite mit Inhalt füllen  
// ...  
  
TabItem tabItem = new TabItem(folder, SWT.NONE);  
tabItem.setText("Farbauswahl");  
tabItem.setImage(image1);  
tabItem.setControl(composite);  
  
// 2. TabItem erzeugen
```

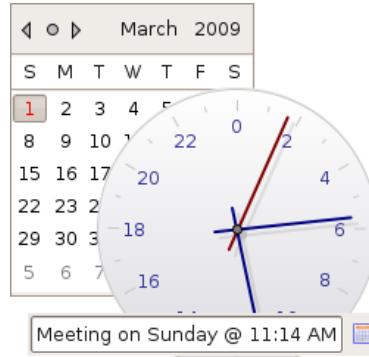
- **CTabFolder** ist mächtiger und unterstützt neben der Schließtaste und weiteren Attributen auch sehr gut Auswahlereignisse.
- Unvollständige Übersicht auch unter: <http://www.eclipse.org/swt/widgets/>

Widgets

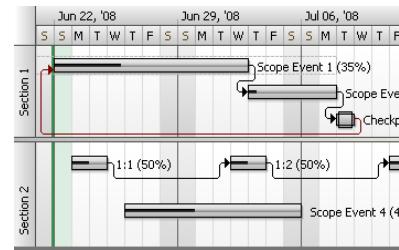
Nebula-Projekt



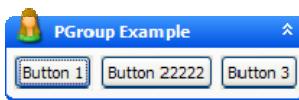
- Weitere Komponenten kostenlos im Nebula-Projekt unter
<http://www.eclipse.org/nebula/> (Auswahl):



	One	Two	Three	Four	Five	Six	Seven
1	One	Two	Three	Four	Five	Six	Seven
2	Two	Two	Three	Four	Four	Six	Seven
3	Three	Three	Three	Four	Five	Six	Seven
4	Four	Four	Four	Four	Five	Six	Seven
5	Five	Five	Five	Five	Five	Six	Seven
6	Six	Six	Six	Six	Six	Six	Seven
7	Seven						

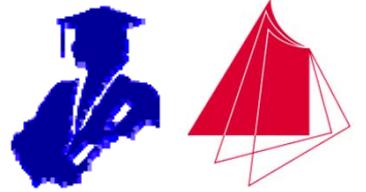


	First Column	Column Grouping	The Column #2	Click me
1	Item #...0000	Test data	asd fjas;dfljk	
2	should be 11	first tree		
3	recordree	this is a readonly checkbox (if		
4	first tree	first tree		
5	first tree	first tree		
6	Item #55	This cell spans over many columns, use setColumnSpan to achieve this		
7	Item #3	Test data	asd fjas;dfljk	
8	Item #4	Test data	asd fjas;dfljk	
9	Item #5	Test data	asd fjas;dfljk	

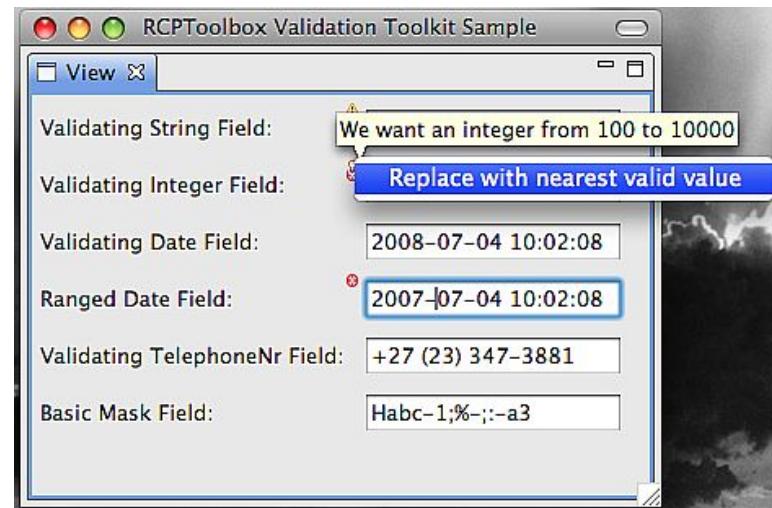
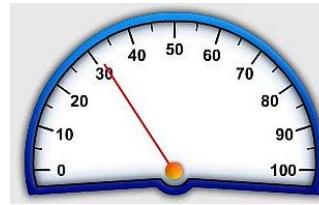
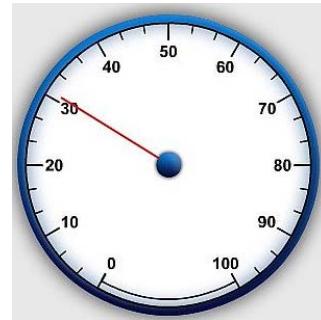
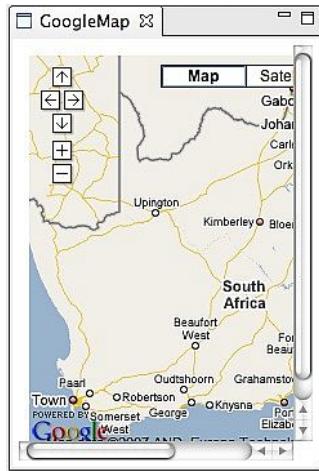


Widgets

RCPToolbox



- Als RCPToolbox unter: http://www.richclientgui.com/detail.php?product_id=1



- Einfache Widgets im Projekt OPAL: <http://code.google.com/a/eclipselabs.org/p/opal/>



- Java kann Ressourcen des Fenstersystems (Fenster, Farben, Zeichensätze, Tasten, ...) nicht automatisch freigeben.
- Zur manuellen Freigabe ist der Aufruf von **dispose()** erforderlich.
- Danach darf nicht mehr auf das Objekt zugegriffen werden.
- Wie sieht das in der Praxis aus?
 - ◆ **Regel 1:** Wer ein Element erzeugt, gibt es auch mit **dispose()** wieder frei.

Beispiel:

```
Color color = new Color(display, 0, 0, 255);
...
color.dispose();
```

Wird ein existierendes (vordefiniertes) Element verwendet, dann darf es nicht freigeben werden:

```
Color color = display.getSystemColor(SWT.COLOR_RED);
```



- ◆ **Regel 2:** Wird das Vaterelement gelöscht, dann werden automatisch auch die Kindelemente entfernt.

Dialogelemente bilden eine Hierarchie: Das Vaterelement wird immer dem Konstruktor des Kindelementes übergeben. Beispiel:

```
Shell shell = new Shell(display);
Button button = new Button(shell, SWT.PUSH);
```

Die Taste (**button**) ist ein Kindelement des Fensters (**shell**).

Es reicht aus, die Shell freizugeben:

```
shell.dispose(); // Gibt auch button wieder frei.
```



- JFace-Ansatz:
 - ◆ Ein zentrales oder mehrere Registry-Objekte für Bilder, Zeichensätze und Farben der Anwendung.
 - ◆ Die Bilder usw. werden bei ihrer ersten Verwendung erzeugt. In der Registry befinden sich lediglich Beschreibungen (Deskriptoren) der Bilder.
- Ablauf bei JFace für Bilder:
 - ◆ Es kann mehr als eine Registry für Bilder angelegt werden (aufgabenabhängig).
 - ◆ Für alle möglicherweise benötigten Bilder werden Deskriptoren (oder auch die Bilder selbst) in der Registry abgelegt.
 - ◆ Beim Auslesen eines Bildes wird anhand des Deskriptors das Bild geladen.



- Ablauf:
 - ◆ Registrierungstabelle zentral deklarieren:

```
private ImageRegistry imageRegistry;
```
 - ◆ Ressourcen durch Deskriptoren beschreiben:

```
private void createImages() {  
    imageRegistry = new ImageRegistry(Display.getDefault());  
    imageRegistry.put("fighting-duke",  
        ImageDescriptor.createFromFile(this.getClass(),  
            "/resources/duke-fight.gif"));  
}
```
 - ◆ Ressourcen verwenden:

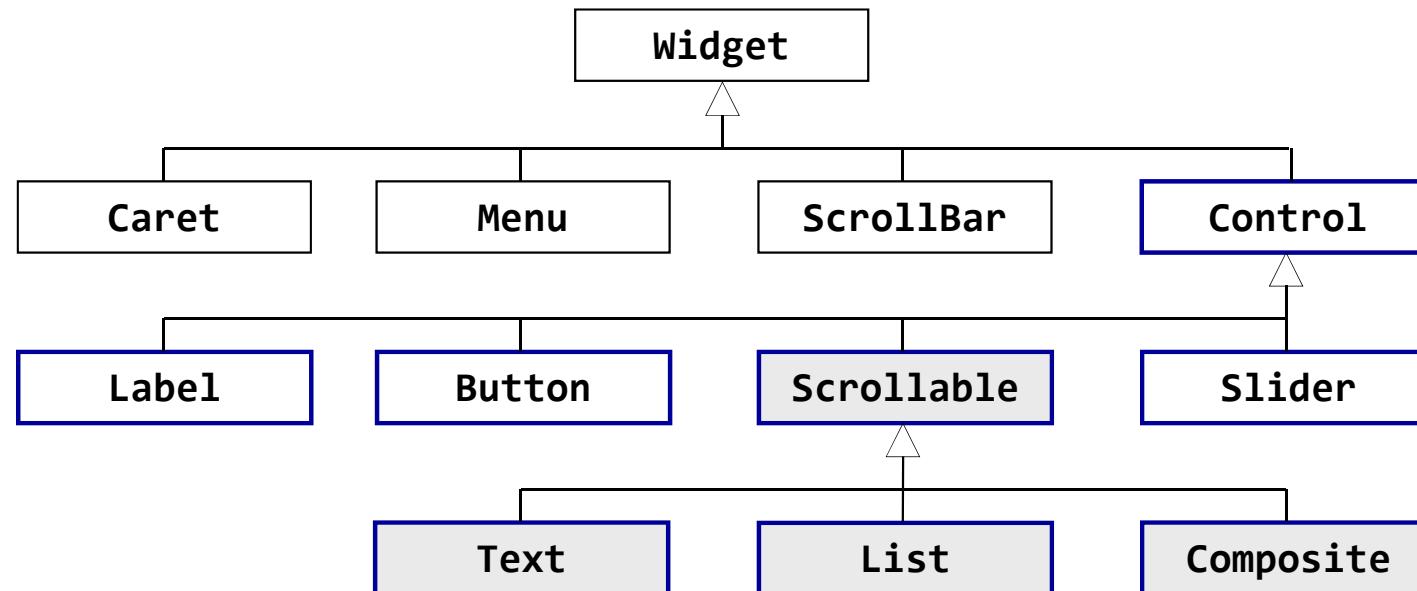
```
button.setImage(imageRegistry.get("fighting-duke"));
```

Widgets



Klassenhierarchie

- Kleiner Ausschnitt aus der Klassenhierarchie



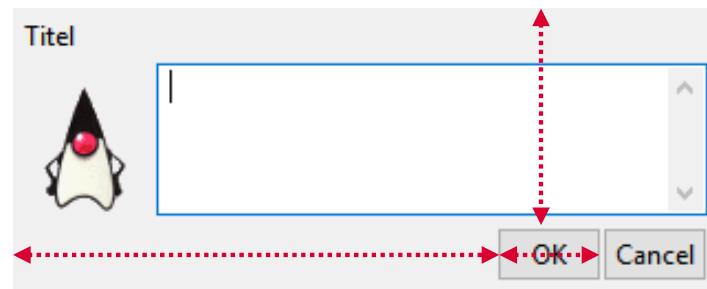


- **Widget:**
 - ◆ Basisklasse für alle Klassen, die Benutzerinteraktion zulassen
 - ◆ Verwaltet den disposed-Zustand (Widget wurde wieder freigegeben)
 - ◆ Erlaubt die Speicherung anwendungsspezifischer Informationen am Widget:
 - **Object getData(String key)**: Wert zu einem Schlüssel auslesen
 - **void setData(String key, Object value)**: Wert mit seinem Schlüssel speichern
 - ◆ Untypisierte Ereignisse (Ereignisse werden gleich genauer behandelt):
 - **DisposeEvent**: Die Komponente wurde freigegeben.
 - **Event**: Untypisiertes Ereignis, kann alle möglichen Ereignisse beinhalten (siehe **Control**)



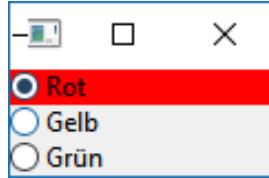
- **Control:**

- ◆ Klasse für die Widgets, die ein Gegenstück auf Betriebssystemebene besitzen
- ◆ Verwaltet
 - aktuelle und bevorzugte Größe
 - Position des Widgets in Bezug auf seinen Container, beim Fenster Position auf dem Bildschirm:

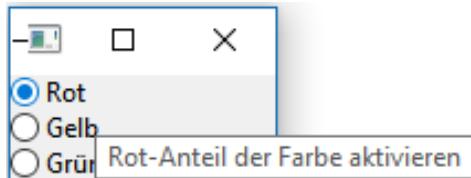




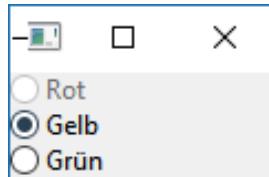
- ◆ Vorder- und Hintergrundfarbe (**setForeground**, **setBackground**) sowie Hintergrundbild (**setBackgroundImage**)



- ◆ ToolTip-Text (**setToolTipText**)

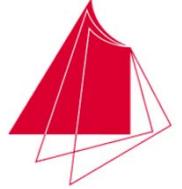


- ◆ Aktivierungszustand (**setEnabled**)





- ◆ Sichtbarkeit (**setVisible**)
- ◆ Menü bzw. Popup-Menü (**setMenu**)
- ◆ Zeichensatz (**setFont**)
- ◆ Layout-Daten (**setLayoutData**)
- ◆ Mauszeigertyp, Tastaturfocus
- ◆ Drag- und Drop von Daten
- ◆ Referenz auf das Vater-Element
- ◆ Typisierte Ereignisse (Ereignisse werden gleich genauer behandelt), Auswahl aus den Ereignissen:
 - **ControlEvent**: Die Komponente wurde verschoben oder in ihrer Größe verändert.
 - **DragDetectEvent**: Der Inhalt der Komponente soll per „Drag and Drop“ verschoben oder kopiert werden.
 - **FocusEvent**: Die Komponente hat den Tastaturfokus erhalten oder verloren.



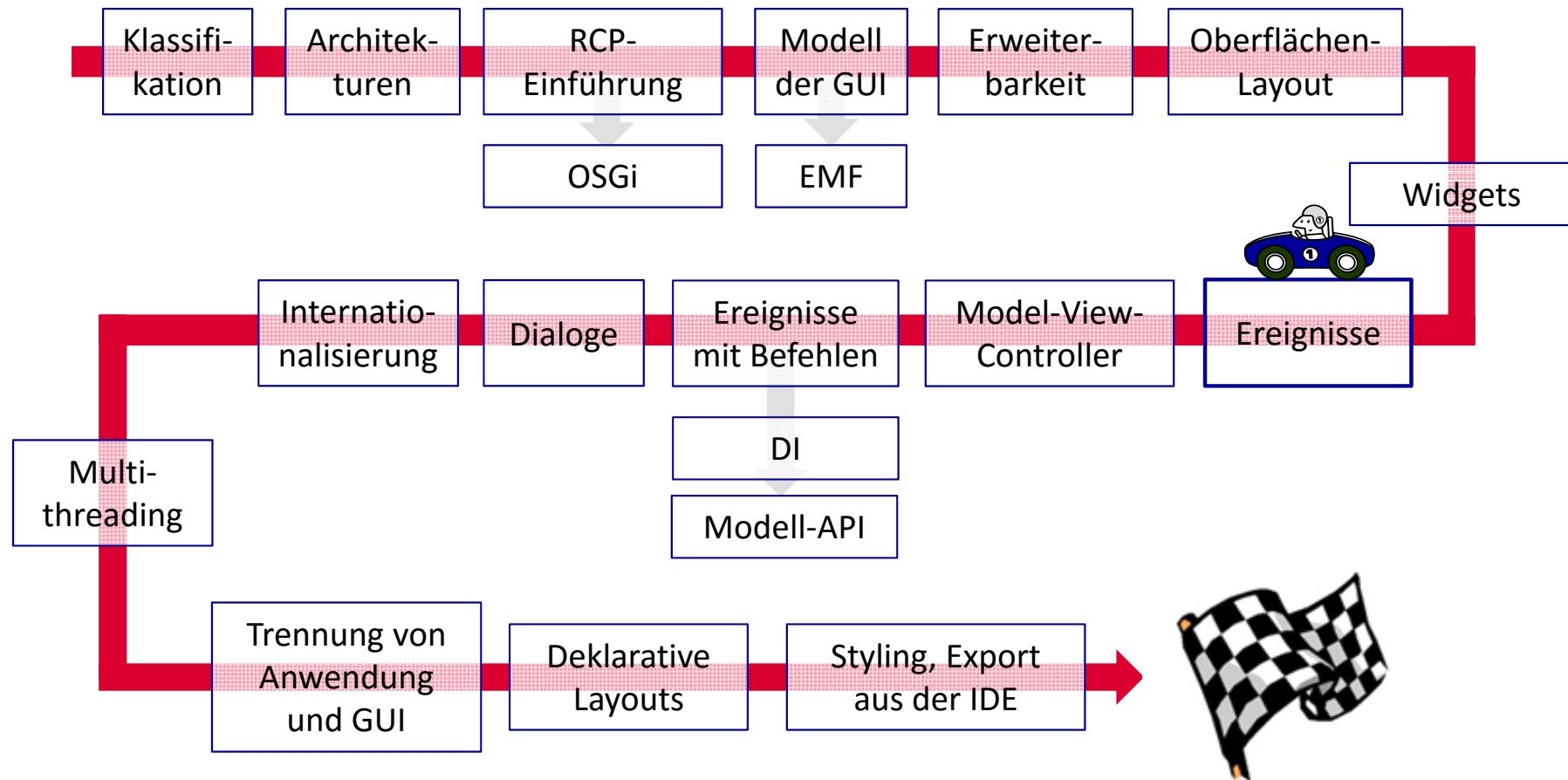
- **HelpEvent**: Es wurde eine Hilfe für die Komponente angefordert.
- **KeyEvent**: Es wurde eine Taste der Tastatur gedrückt oder losgelassen.
- **MenuDetectEvent**: Es wurde ein Popup-Menü für die Komponente angefordert.
- **MouseEvent**: Es wurde eine Maustaste über der Komponente gedrückt.
- **MouseMoveEvent**: Der Mauszeiger wurde über der Komponente bewegt.
- **PaintEvent**: Die Komponente muss neu gezeichnet werden.
- **TraverseEvent**: Es wurde eine Taste gedrückt, um den Fokus auf eine andere Komponente zu setzen.
- ◆ Von **Control** abgeleitete Klassen bieten häufig noch weitere Ereignisse.



- **Scrollable:**
 - ◆ Basisklasse für alle Widgets, die Scrollbalken besitzen können
 - ◆ verwaltet den zur Verfügung stehenden Platz
 - ◆ erlaubt das Auslesen der erzeugten Scrollbalken:
 - **ScrollBar getHorizontalBar()**: horizontalen Scrollbalken auslesen
 - **ScrollBar getVerticalBar()**: vertikalen Scrollbalken auslesen
- **Composite:**
 - ◆ Basisklasse für alle Widgets, die weitere Widgets als Kindelemente besitzen können
 - ◆ verwaltet die Kindelemente und den Layoutmanager, um sie zu platzieren
 - ◆ verwaltet die Reihenfolge, in der die Kindelemente beim Druck auf die Tabulatortaste besucht werden

Grundlagen der Ereignisbehandlung

Beobachter-Muster

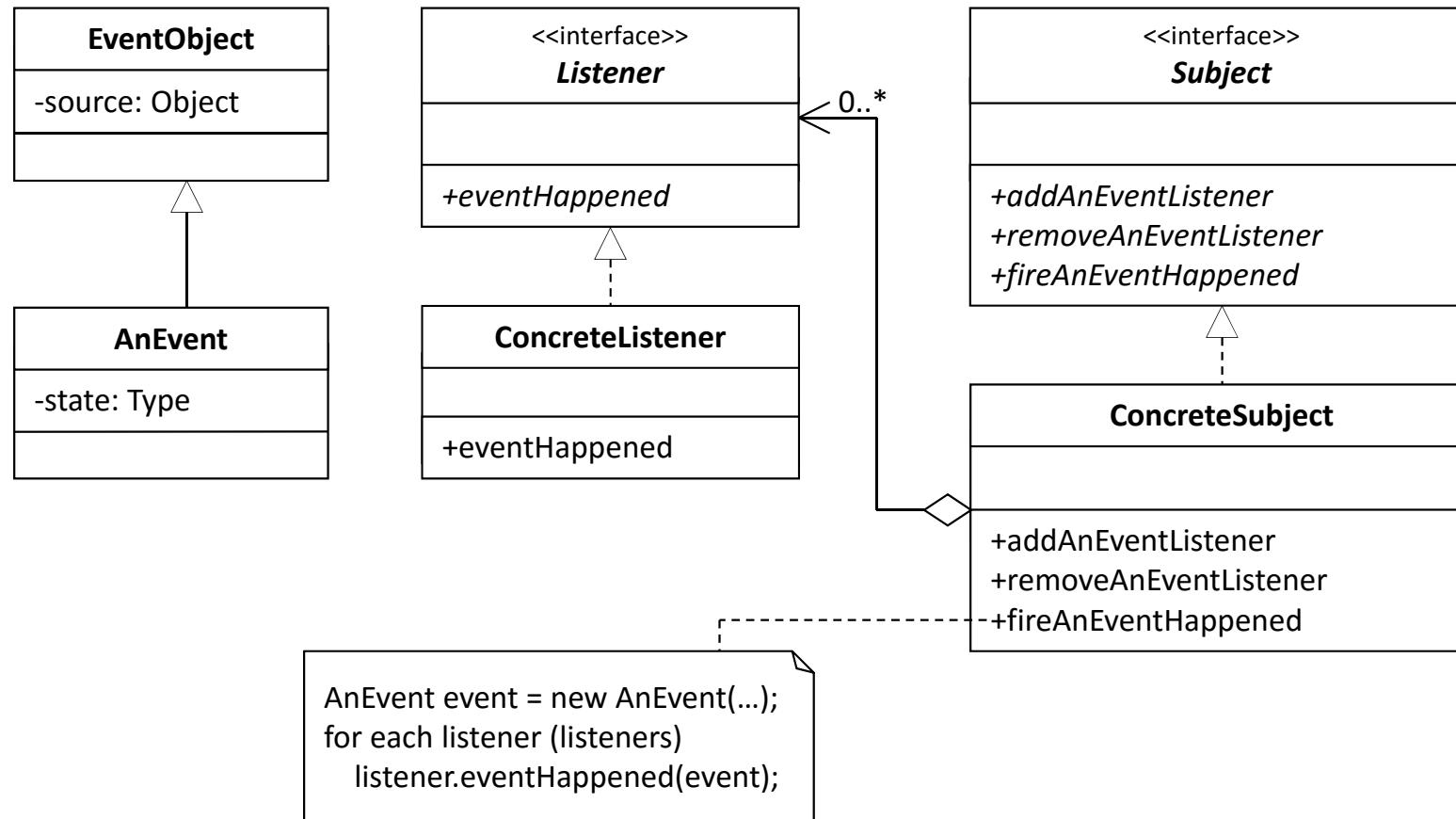


Grundlagen der Ereignisbehandlung



Beobachter-Muster

- Die einfache Ereignisbehandlung basiert auf einem modifizierten Beobachter-Entwurfsmuster (Listener = Beobachter):

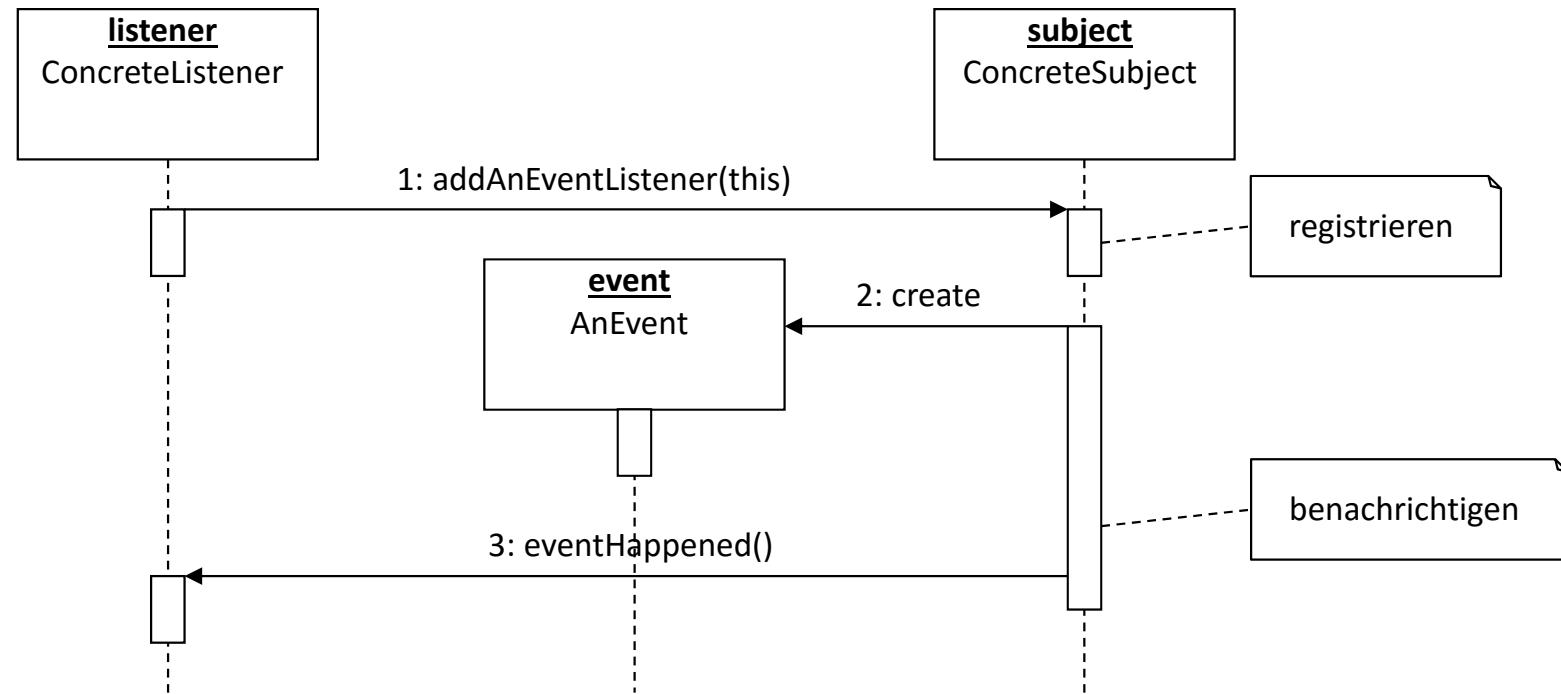


Grundlagen der Ereignisbehandlung



Beobachter-Muster

- Ablauf der Ereignisweitergabe (Delegation):

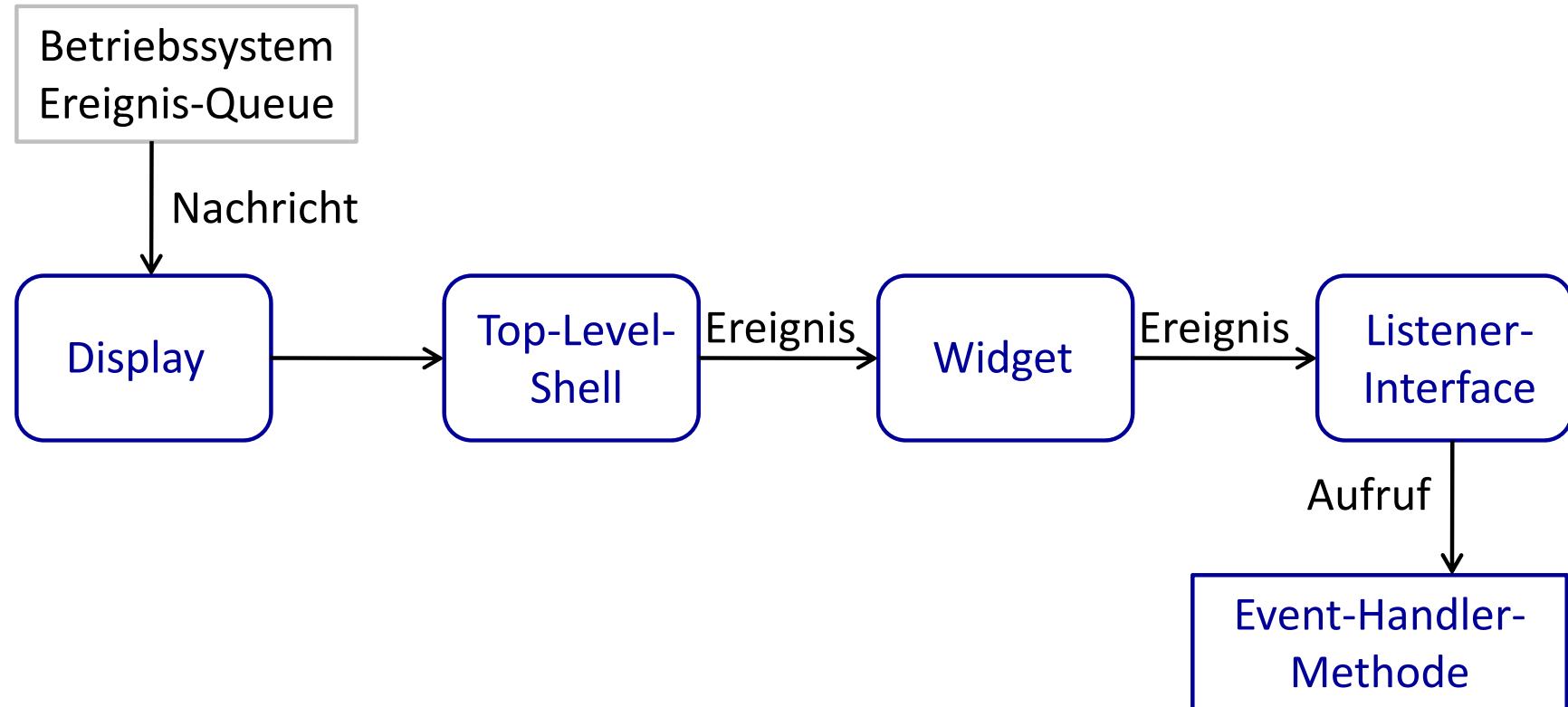


- Das Model wird gelegentlich auch „publisher-subscribe model“ genannt.

Grundlagen der Ereignisbehandlung



Beobachter-Muster: Ablauf in SWT





- In SWT werden sehr viele unterschiedliche Arten von Ereignissen unterstützt.
- Häufigstes Ereignis: Das **SelectionEvent** soll hier näher betrachtet werden.

SelectionEvent

- Eine Taste (**Button**) löst ein sogenanntes Selection-Event aus, nachdem die Taste gedrückt und wieder losgelassen wurde.
- Um diese Ereignisse abzufangen, müssen immer die folgenden Schritte durchlaufen werden:
 1. Definition der Beobachter-Klassen
 2. Erzeugen einer Instanz der Beobachter-Klasse
 3. Registrierung der Beobachter-Instanz an einer Komponente.
- Namensaufbau am Beispiel des Auswahl-Ereignisses (**Selection**):
 - ◆ Ereignisklasse: **SelectionEvent**
 - ◆ Beobachter (Interface): **SelectionListener**
 - ◆ Adapter (Klasse mit leeren Methodenrumpfen, wenn der Beobachter mehr als eine Methode besitzt): **SelectionAdapter**

Grundlagen der Ereignisbehandlung



Beobachter-Muster: SelectionEvent

- Beispiel (Implementierung des Beobachters als anonyme innere Klasse):

```
Button okButton = new Button(parent, SWT.PUSH);
okButton.setText("Weiter");
okButton.addSelectionListener(new SelectionListener() {

    // die wichtige Methode
    @Override
    public void widgetSelected(SelectionEvent event) {
        System.out.println("widgetSelected: " + event.widget);
    }

    // wird auf einigen Plattformen nie erzeugt (z.B.
    // Doppelklick auf einen Listeneintrag)
    @Override
    public void widgetDefaultSelected(SelectionEvent event) {
    }
});
```

Grundlagen der Ereignisbehandlung



Beobachter-Muster: SelectionEvent

- Beispiel (Implementierung des Beobachters durch Vererbung von der Adapter-Klasse):

```
Button okButton = new Button(shell, SWT.PUSH);
okButton.setText("Weiter");
okButton.addSelectionListener(new SelectionAdapter() {

    // die wichtige Methode
    @Override
    public void widgetSelected(SelectionEvent event) {
        System.out.println("widgetSelected: " + event.widget);
    }
});
```

Grundlagen der Ereignisbehandlung



Beobachter-Muster: SelectionEvent

- Unterscheidungsvarianten, wenn mehr als eine Komponente ein Ereignis auslösen kann und immer derselbe Beobachter verwendet wird:
 1. An jeder Komponente kann ein Wert (z.B. ein **String**) gespeichert werden, der später in **widgetSelected** wieder ausgelesen wird.
 2. Es können die Referenzen auf die Widgets aus dem Ereignis-Objekt verglichen werden.
- Quelltext (Projekt [de.hska.iwii.evente4rcpprojekt](#)):

```
final Button okButton = new Button(shell, SWT.PUSH);
okButton.setText("Weiter");
okButton.setData("source", "ok");
okButton.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent event) {
        // Variante 1
        boolean wasOk = event.widget.getData("source").equals("ok");
        // Variante 2
        boolean wasOk = event.widget == okButton;
    }
});
```



1. Eine zentrale Beobachterklasse empfängt die Ereignisse

- Definition der Beobachter-Klasse:
 - ◆ Eine Klasse, die **SelectionEvent**s empfangen möchte, muss von **SelectionAdapter** erben. Die folgende Methode muss implementiert werden:
- Beispiel (hier Beobachter als anonyme innere Klasse umgesetzt):

```
SelectionListener listener = new SelectionAdapter() {  
    @Override  
    public void widgetSelected(SelectionEvent event) {  
        ...  
    }  
});  
  
okButton.addSelectionListener(listener);  
cancelButton.addSelectionListener(listener);
```

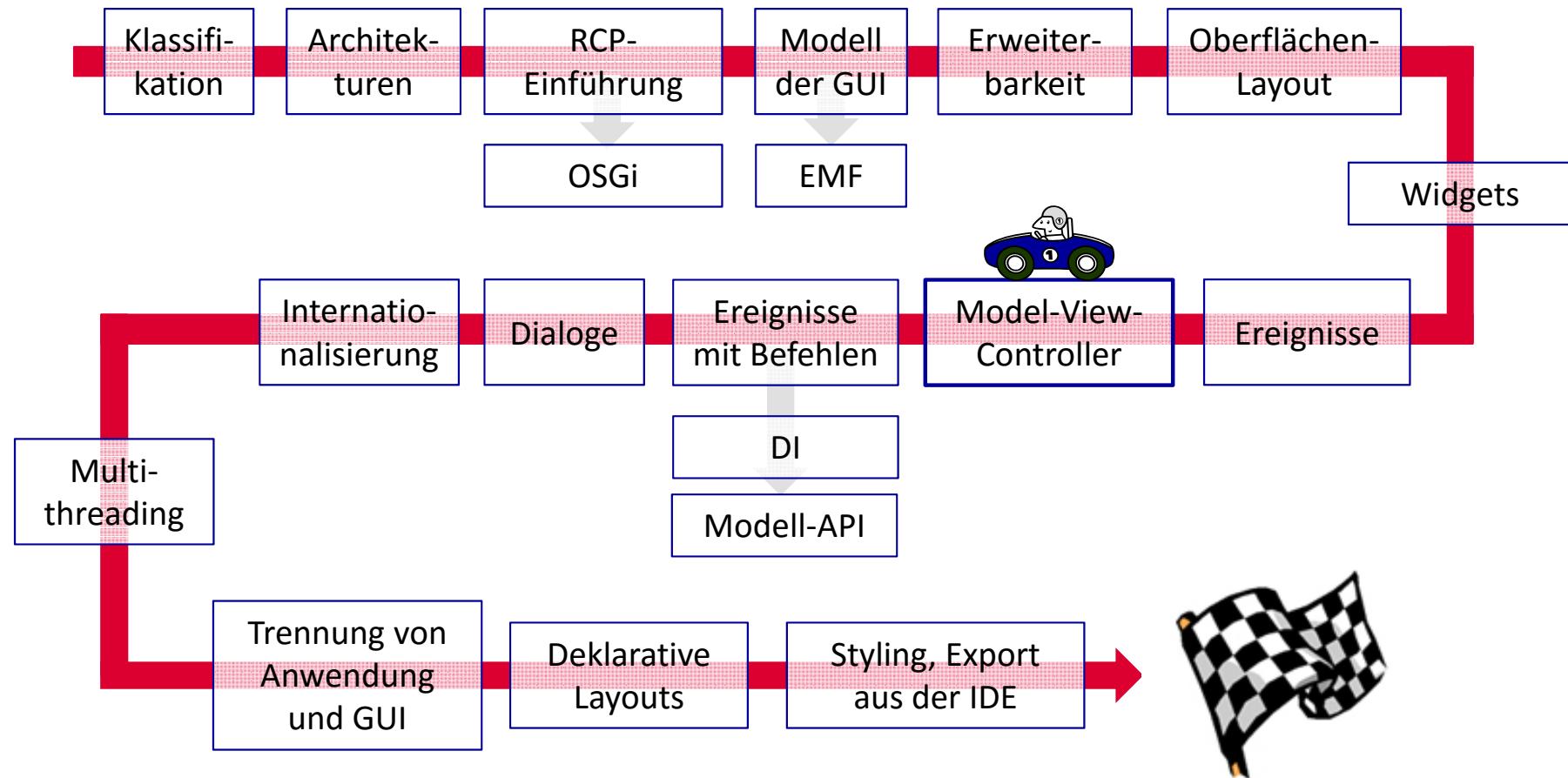
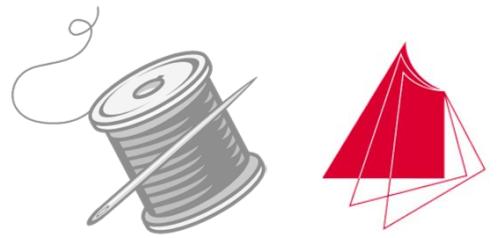


2. Für jede Ereignisquelle existiert eine eigene Beobachter-Klasse

- Unterschied zu Variante 1: Es wird für jedes Widget, das ein Ereignis auslösen kann, eine eigene Beobachter-Klasse geschrieben.

Model-View-Controller

Einführung





- Wie werden die von Widgets dargestellten Daten verwaltet?
- Wie stellen sich Widgets dar?
- Wie werden Ansicht und Daten (Modell) getrennt?

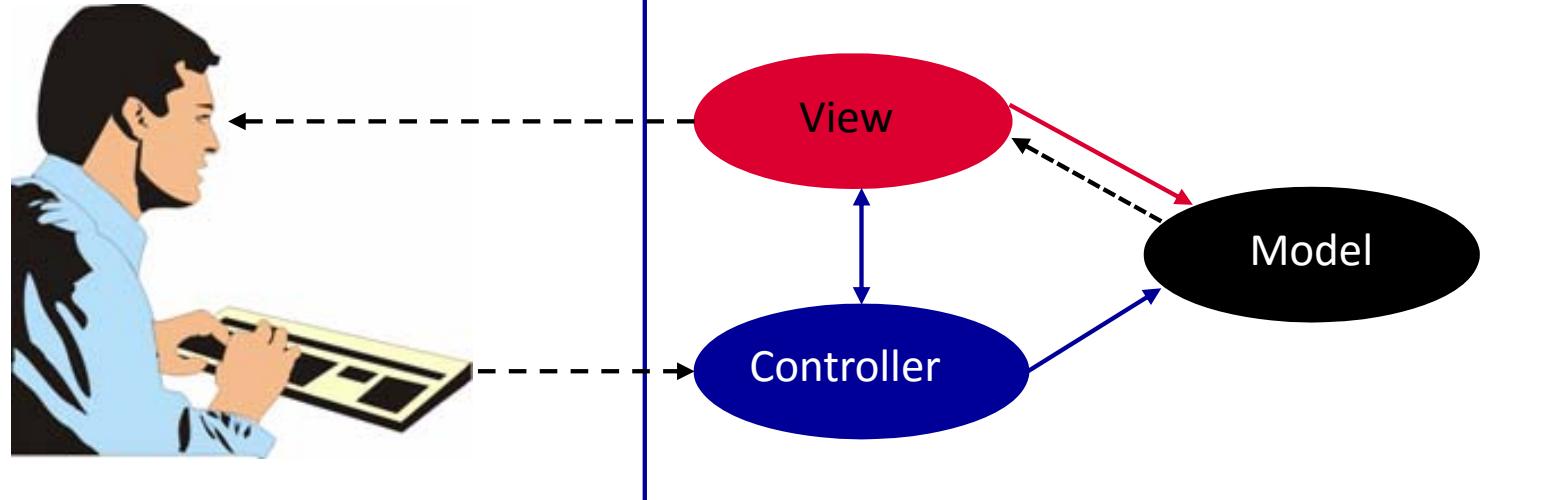
- MVC wurde zuerst in Smalltalk Ende der 80'er des vorigen Jahrhunderts eingesetzt:
 - ◆ Spezielle Form des Beobachter-Musters (Observer)
 - ◆ Model: Zustandsinformation der Komponente
 - ◆ View: Beobachter des Zustands, um diesen darzustellen
 - ◆ Controller: Legt das Verhalten des Widgets auf Benutzereingaben fest.

Model-View-Controller



Einführung

- Allgemeiner Aufbau des MVC:



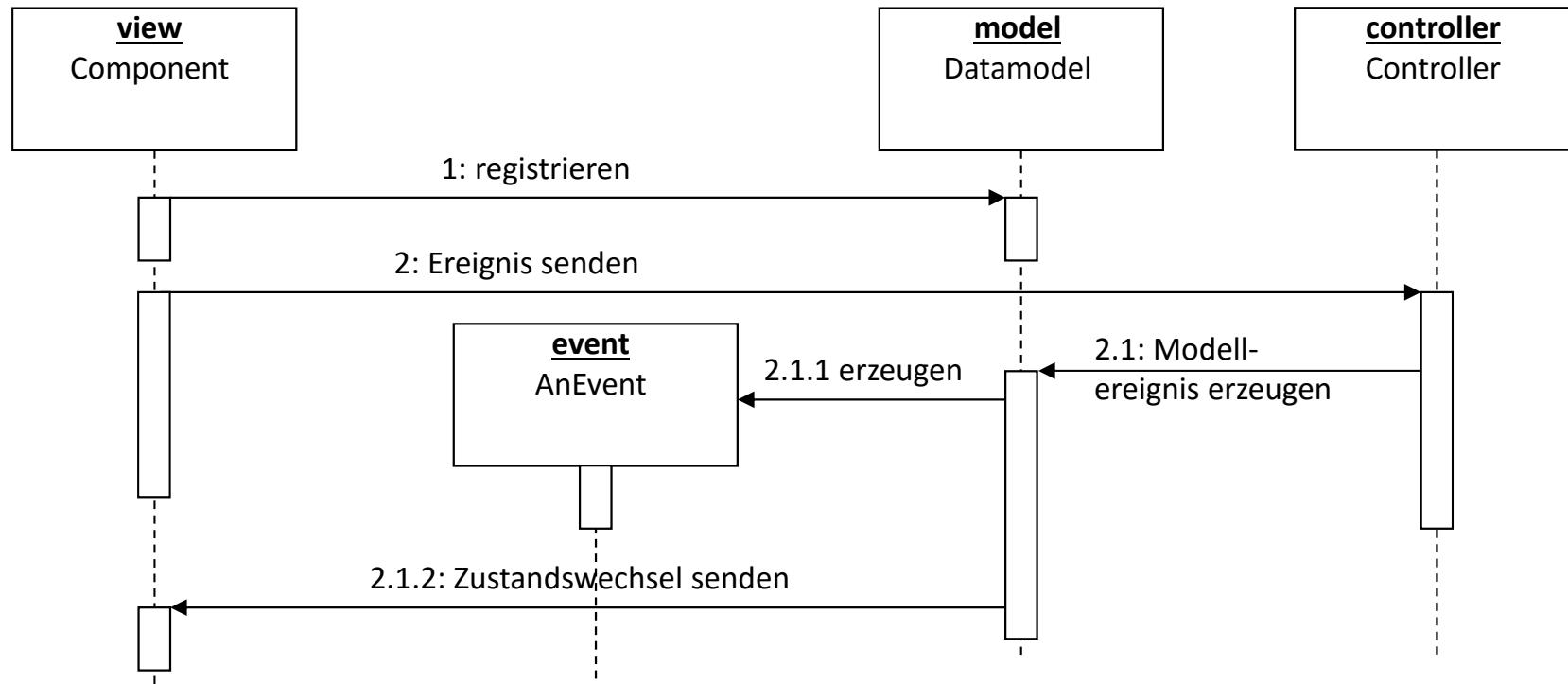
- Der Controller verändert das Modell bei Benutzereingaben.
- Der Controller informiert die Sicht bei Benutzereingabe, die nicht das Modell verändern (z.B. Verschieben eines Textcursors).
- > Das Modell informiert die Sicht, wenn sich die Daten geändert haben.
- Die Sicht holt sich die Daten aus dem Modell und stellt sie dar.
- Wichtig: Das Modell kennt weder Controller noch Sicht.

Model-View-Controller

Einführung



- Diagramm mit Pseudomethoden zur MVC-Kommunikation:





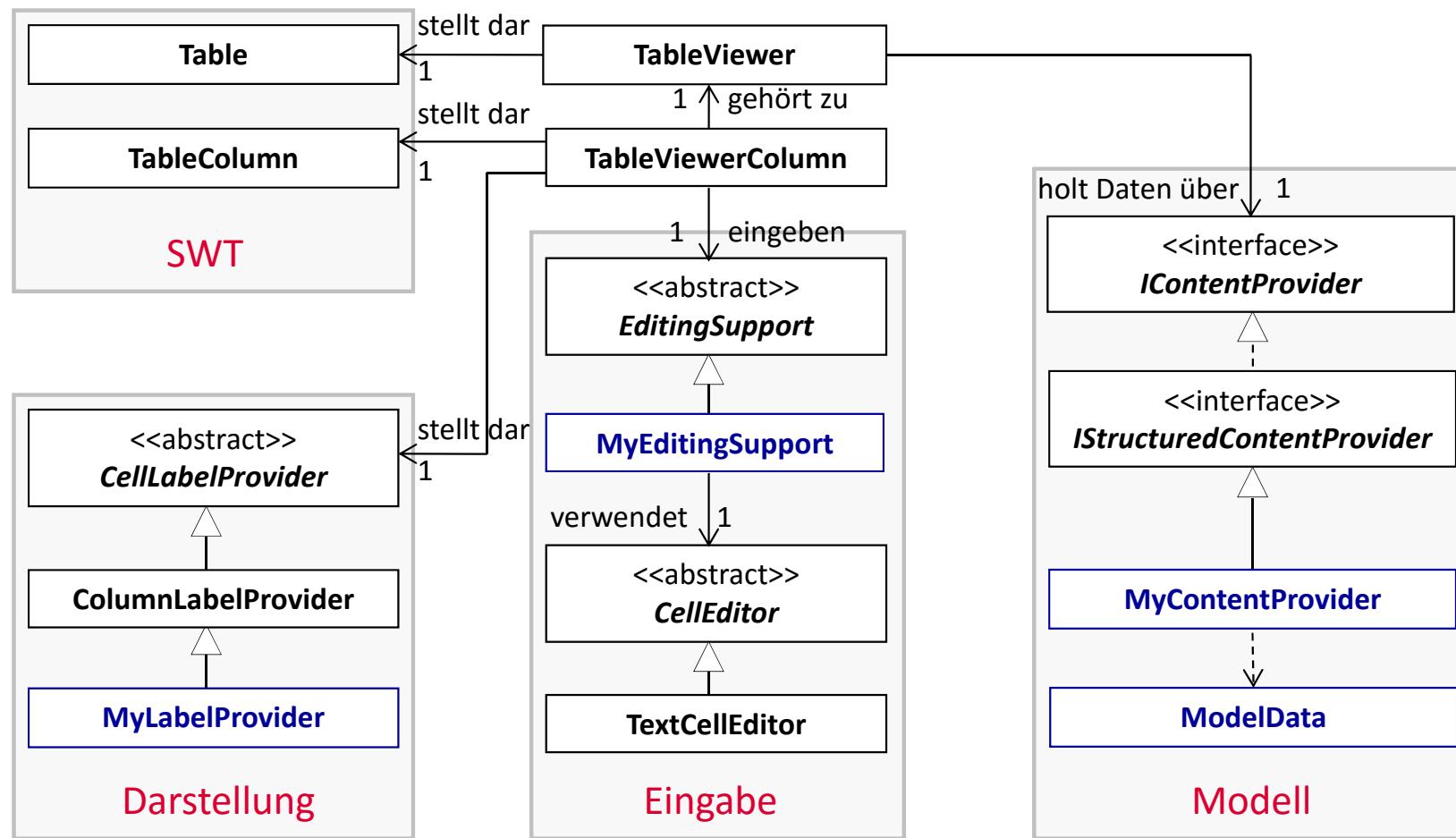
- JFace bietet für einige SWT-Widgets einen sauberen MVC-Ansatz:
 - ◆ **Tree: TreeViewer**
 - ◆ **Table: TableViewer**
 - ◆ **List: ListViewer**
 - ◆ **Combo: ComboViewer**
- Die Klassen sind in **org.eclipse.jface.viewers** zu finden.

Model-View-Controller



Beispiel JFace-Tabelle

Stark vereinfachtes Klassendiagramm für den **TableViewer** (blaue Schrift: eigene Klassen der Anwendung):





- Verwendung des Modells:
 - ◆ **ModelData** ist eine fiktive Anwendungsklasse, die die Modelldaten beinhaltet:
 - beispielsweise als **ArrayList** mit Objekten
 - Jeder Eintrag der **ArrayList** entspricht einer Zeile der Tabelle:
 - Filter: Nicht alle Einträge müssen angezeigt werden.
 - Sortierung: Die Reihenfolge in der Ansicht kann anhand von Kriterien vertauscht werden.
 - Attribute der Objekte werden in Zellen der Zeile dargestellt.
 - ◆ Beispiel: **ArrayList<Person>**

```
public class Person {  
    private String name;  
    private int age;  
    // ...  
}
```

Daten	
Name	Age
Name 1	42
Name 2	66
Name 3	33
Name 4	77

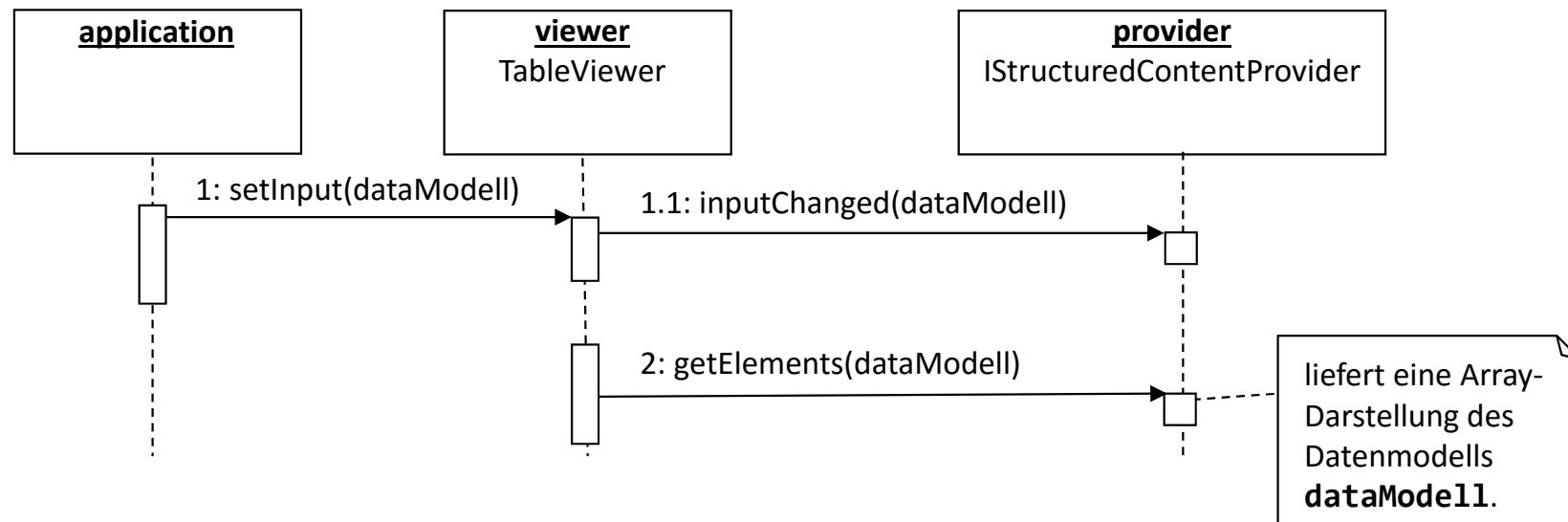
ArrayList

Model-View-Controller



Beispiel JFace-Tabelle

- Der **TableViewer** greift über einen **IStructuredContentProvider** auf die Modelldaten der Anwendung zu:
 - Dieser „versorgt“ den **TableViewer** mit den Modelldaten.
 - Der **IStructuredContentProvider** wandelt die Modelldaten in ein Array mit Objekten um.
 - Er wird vom **TableViewer** informiert, wenn neue Modelldaten vorliegen.

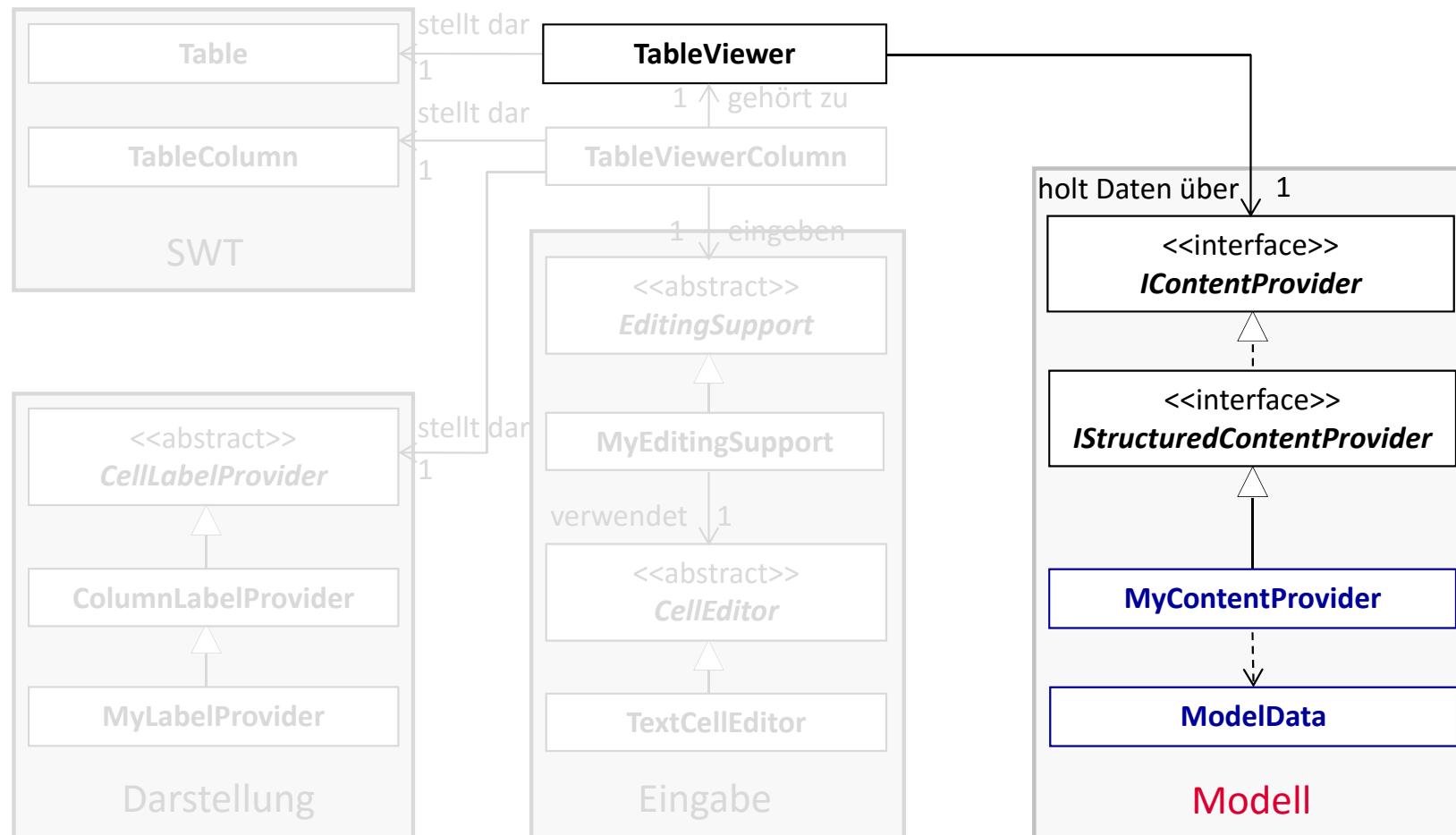


Model-View-Controller

Beispiel JFace-Tabelle



Beschaffung der Daten durch den **TableViewer**:





- Code bisher (Beispiel im RCP-Projekt zur Tabelle):

```
public class PersonContentProvider implements IStructuredContentProvider {  
    @Override  
    public Object[] getElements(Object inputElement) {  
        return ((List<Person>) inputElement).toArray();  
    }  
  
    @Override  
    public void dispose() {  
        // keine Ressourcen angelegt  
    }  
  
    @Override  
    public void inputChanged(Viewer viewer, Object oldInput,  
                            Object newInput) {  
        // interessiert uns hier nicht  
    }  
}
```



- Hauptanwendung (Ausschnitt):

```
public class TablePart {  
    private TableViewer tableViewer;  
    private ArrayList<Person> personModel = new ArrayList<>();  
  
    @Inject  
    public TablePart(Composite parent) {  
        // Modell füllen  
        personModel.add(new Person("Name 1", 42));  
        personModel.add(new Person("Name 2", 66));  
  
        Composite container = new Composite(parent, SWT.NONE);  
        container.setLayout(new FillLayout());  
  
        tableViewer = new TableViewer(container, SWT.FULL_SELECTION);  
        tableViewer.setContentProvider(new PersonContentProvider());  
        tableViewer.setInput(personModel);  
  
        // usw.  
    }  
}
```



- Fertig? Nein:



- Die Tabellenspalten müssen manuell erzeugt werden:

```
Table table = tableViewer.getTable();
TableViewerColumn column = new TableViewerColumn(tableViewer, SWT.LEFT);
column.getColumn().setText("Name");
column.getColumn().setWidth(200);
column = new TableViewerColumn(tableViewer, SWT.RIGHT);
column.getColumn().setText("Age");
column.getColumn().setWidth(80);

table.setHeaderVisible(true); // sichtbare Header
table.setLinesVisible(true); // sichtbare Linien
```



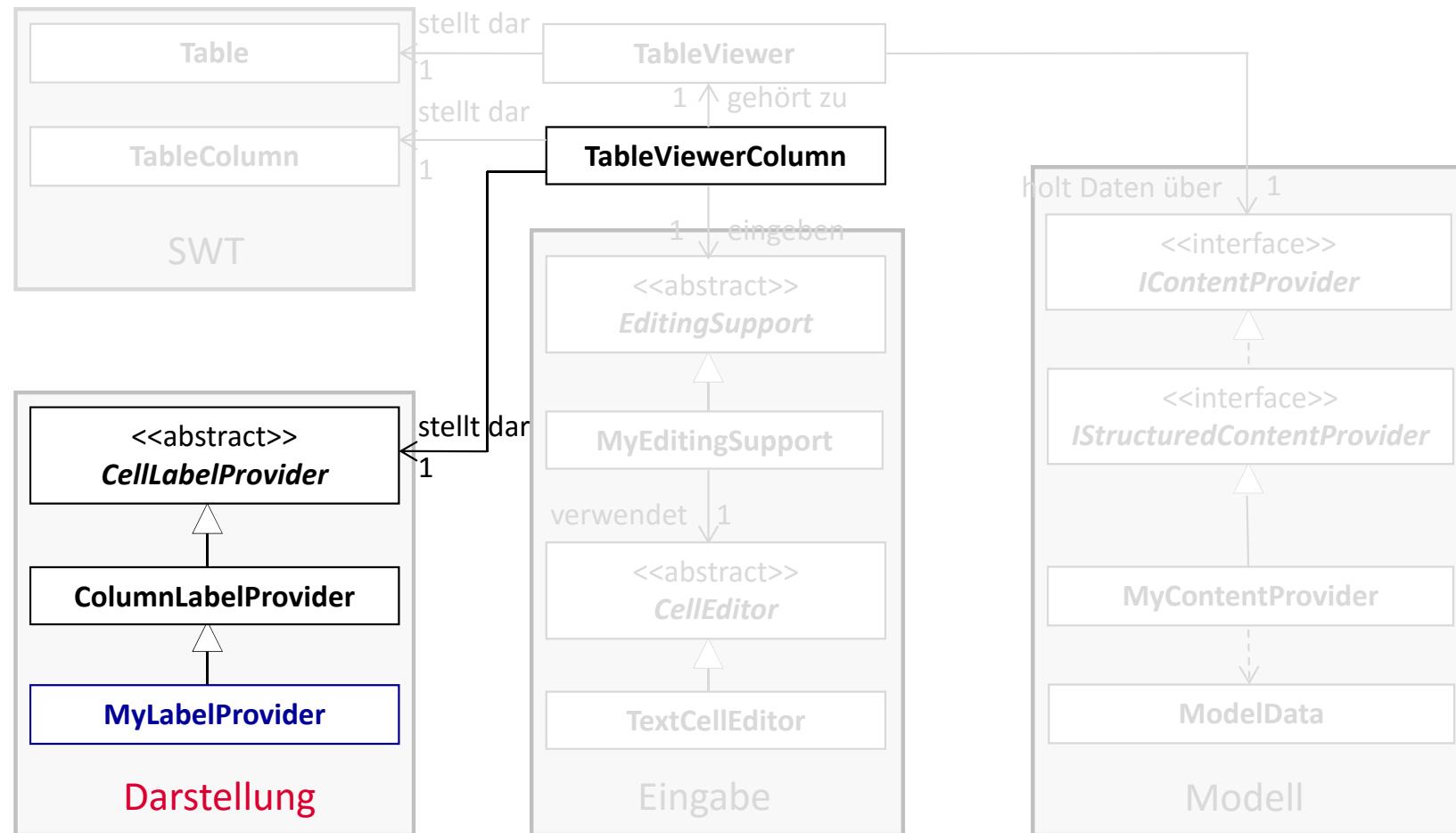
- Die darzustellenden Strings müssen von einer eigenen Klasse bereitgestellt werden, die von **ColumnLabelProvider** erbt:
 - **public String getText(Object element)**: **element** ist ein **Person**-Objekt.
 - **public Image getImage(Object element)**: falls auch ein Bild dargestellt werden soll. Ansonsten wird die Methode nicht überschrieben.
 - ◆ Es wird ein spezieller **ColumnLabelProvider** an jeder Spalte registriert.

Model-View-Controller

Beispiel JFace-Tabelle



Erstellung der darzustellenden Strings:



Model-View-Controller



Beispiel JFace-Tabelle

```
// Für den Namen einer Person
public class PersonNameLabelProvider extends ColumnLabelProvider {
    @Override
    public String getText(Object element) {
        return ((Person) element).getName();
    }
}
```

```
// Für das Alter einer Person
public class PersonAgeLabelProvider extends ColumnLabelProvider {
    @Override
    public String getText(Object element) {
        return String.valueOf(((Person) element).getAge());
    }
}
```

- Häufig sind hier anonyme innere Klassen eine gute Wahl.

Model-View-Controller



Beispiel JFace-Tabelle

```
// An den Spalten registrieren, hier exemplarisch für die Namensspalte:  
column.setLabelProvider(new PersonNameLabelProvider());  
// Und nicht vergessen: Dem TableViewer die Modelldaten übergeben.  
tableViewer.setInput(personModel);
```

- Und jetzt?

Daten	
Name	Age
Name 1	42
Name 2	66
Name 3	33
Name 4	77



- Dynamische Spaltenbreiten?

Daten	
Name	Age
Name 1	42
Name 2	66
Name 3	33
Name 4	77

Daten	
Name	Age
Name 1	42
Name 2	66
Name 3	33
Name 4	77

Daten
Name
Name 1
Name 2
Name 3
Name 4

- Spalten passen sich nicht der vorhandenen Breite an!
- Lösung: Es gibt den Layout-Manager **TableColumnLayout**, der die Spalten einer Tabelle automatisch anpasst:
 - ◆ Die Tabelle ist das alleinige Widget in einem **Composite**.
 - ◆ Dem **Composite** wird **TableColumnLayout** als Layout zugewiesen.
 - ◆ Die Breiten werden durch **ColumnPixelData** (absolut in Pixeln) oder **ColumnWeightData** (prozentual) den Spalten zugewiesen. Auch minimale Breiten können erzwungen werden.



- Beispiel erweitert die Personenanwendung:
 - ◆ Die Tabelle des **TableViewers** muss das alleinige Widget in einem Container sein.

```
Composite tableComp = new Composite(parent, SWT.NONE);
TableColumnLayout tcl = new TableColumnLayout();
tableComp.setLayout(tcl);
tableViewer = new TableViewer(tableComp, SWT.FULL_SELECTION);
```

- ◆ Dem Layout werden hier relative Spaltenbreiten zugewiesen. Eine Größenänderung ist erlaubt.

```
TableViewerColumn column = new TableViewerColumn(tableViewer,
                                                SWT.LEFT);
column.getColumn().setText("Name");
// 70% der Platzes, Größenänderung ist erlaubt
tcl.setColumnData(column.getColumn(), new ColumnWeightData(70, true));
```

Model-View-Controller

Beispiel JFace-Tabelle



- Jetzt stimmen die Spaltenbreiten:

Daten	
Name	Age
Name 1	42
Name 2	66
Name 3	33
Name 4	77

Daten	
Name	Age
Name 1	42
Name 2	66
Name 3	33
Name 4	77

Daten	
Name	...
Name 1	..
Name 2	..
Name 3	..
Name 4	..



- Sortierung und Filterung der Daten:
 - ◆ Ein Klick auf den Header einer Spalte stellt den Inhalt der Tabelle sortiert anhand des Inhaltes der Spalte dar (siehe Windows Explorer, Linux Datei-Manager, ...)
 - ◆ Ausblenden von Zeilen bei sehr großen Tabellen, um einen Überblick zu erhalten (siehe auch Explorer, um z.B. versteckte Dateien auszublenden)
- Ablauf beim Sortieren:

Daten	Name	Age
	Name 1	42
	Name 2	66
	Name 3	33
	Name 4	77

Klick auf Name-Header

Daten	Name	Age
	Name 4	77
	Name 3	33
	Name 2	66
	Name 1	42

Klick auf Age-Header

Daten	Name	Age
	Name 4	77
	Name 2	66
	Name 1	42
	Name 3	33



- Beispiel fortgeführt:

- ◆ Sorter erzeugen und am **TableViewer** registrieren:

```
final PersonSorter sorter = new PersonSorter();
tableViewer.setSorter(sorter)
```

- ◆ Als Beobachter an jeder Spalte registrieren, um das Sortieren zu starten:

```
column.getColumn().addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        // zu sortierende Spalte in den "Sortierer" eintragen
        sorter.doSort(PersonIndices.NAME_INDEX);
        // Tabelle zwingen, sich neu zu zeichnen
        tableViewer.refresh();
    }
});
```

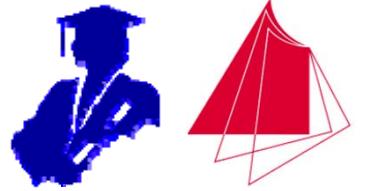


- Es fehlt noch der „Sortierer“:
 - ◆ Er sortiert gar nicht. Er vergleicht immer paarweise zwei Objekte → siehe **Collections.sort**, **Arrays.sort**.
 - ◆ Er bekommt eine weitere Methode, um die zu sortierende Spalte und die Sortierrichtung festzulegen.
- Klasse **PersonSorter**:

```
public class PersonSorter extends ViewerSorter {  
    // mögliche Sortiereihenfolgen  
    public enum Direction { ASCENDING, DESCENDING };  
  
    // aktuelle Spalte, anhand derer sortiert wurde  
    private int sortColumn;  
  
    // aktuelle Sortierreihenfolge  
    private Direction sortDirection = Direction.DESCENDING;
```

Model-View-Controller

Beispiel JFace-Tabelle: Sortierung



```
// Sortierung festlegen. Wenn sich die Spalte ändert, wird immer
// aufsteigend sortiert. Ansonsten wird die Sortierreihenfolge umgedreht.
public void doSort(int column) {
    if (column == sortColumn) {
        sortDirection = (sortDirection == Direction.ASCENDING)
            ? Direction.DESCENDING : Direction.ASCENDING;
    }
    else {
        sortColumn = column;
        sortDirection = Direction.ASCENDING;
    }
}
```



```
// Vergleiche für die Tabelle durchführen → siehe java.util.Comparator
public int compare(Viewer viewer, Object object1, Object object2) {
    int result = 0;
    Person person1 = (Person) object1;
    Person person2 = (Person) object2;

    switch (sortColumn) {
        case PersonIndices.NAME_INDEX:
            result = getComparator().compare(person1.getName(),
                                              person2.getName());
            break;
        case PersonIndices.AGE_INDEX:
            result = person1.getAge() - person2.getAge();
            break;
    }
    result = sortDirection == Direction.DESCENDING ? result : -result;
    return result;
}
```

- Der „Sortierer“ kann auch Objekte zu Gruppen (Kategorien) zusammenfassen → siehe API.



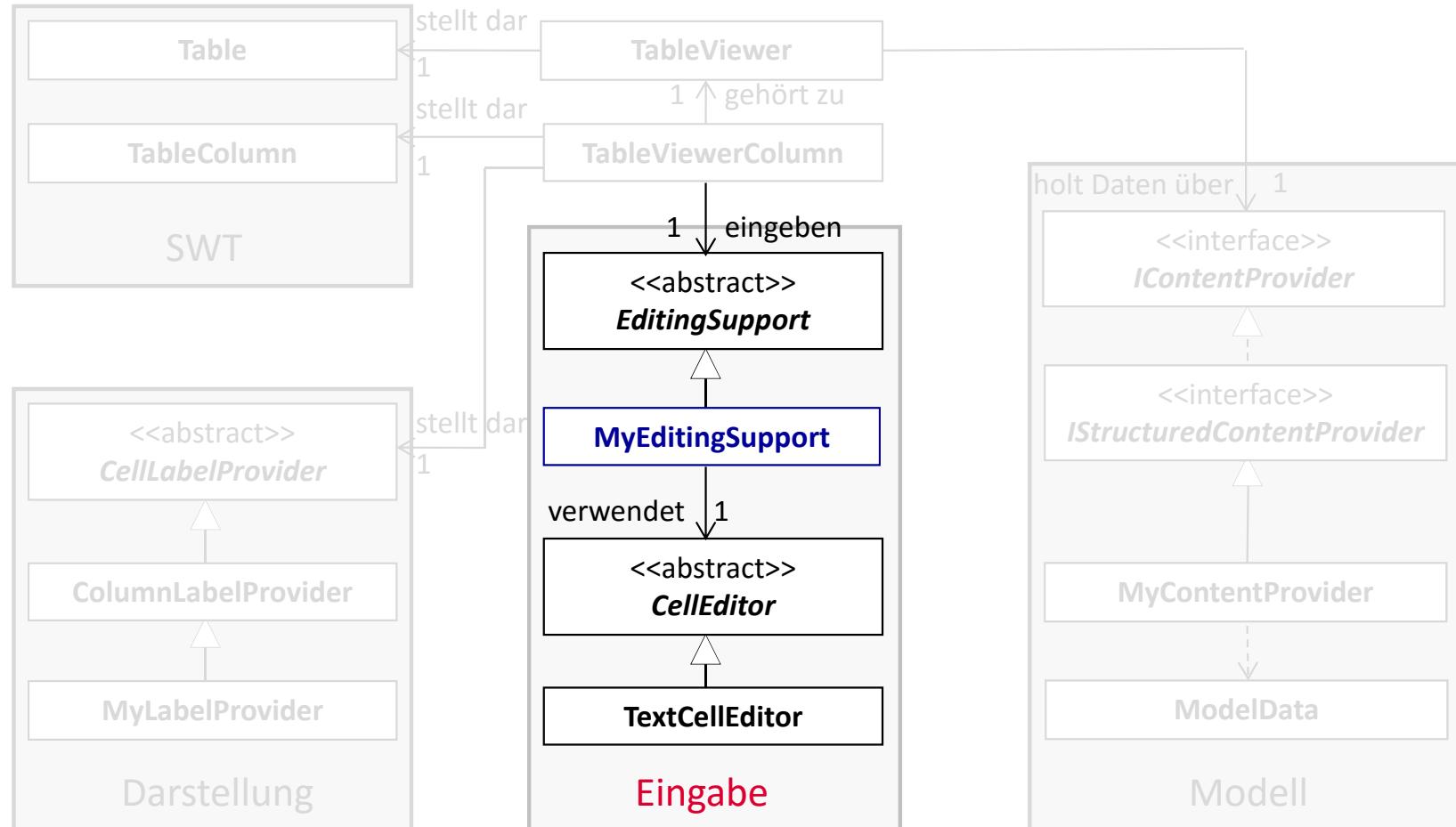
- Filterung von Tabelleninhalten:
 - ◆ Erben von der abstrakten Basisklasse **ViewerFilter**.
 - ◆ Überschreiben der Methode **select**, die bestimmt, ob ein Objekt (eine Person) angezeigt werden soll oder nicht.
 - ◆ Registrieren beliebig vieler Filter am **TableViewer** mit **setFilters**. Alle Filter werden befragt, ob das Objekt angezeigt werden soll oder nicht.
- Große Tabellen, mit **SWT.VIRTUAL** erzeugt:
 - ◆ Es existiert ein spezieller **ILazyContentProvider**.
 - ◆ Der **DeferredContentProvider** kann bei solchen Tabellen das Sortieren und Filtern in einem Hintergrundthread vornehmen.
- Freier gestaltetes Aussehen von Zellen:
 - ◆ In der **TableViewerColumn** einen **StyledCellLabelProvider** registrieren.
 - ◆ Weitere Methode im **ColumnLabelProvider** für Zeichensatz, Farbe, ... überschreiben.
 - ◆ Beliebige Widgets als Zellen: etwas komplizierter...

Model-View-Controller

Beispiel JFace-Tabelle: Eingabe



- Bisher erlauben die JFace-Tabellen in den Beispielen keine Eingabe.

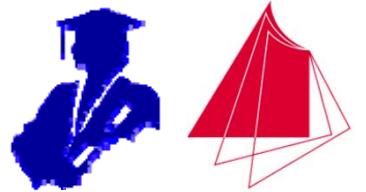




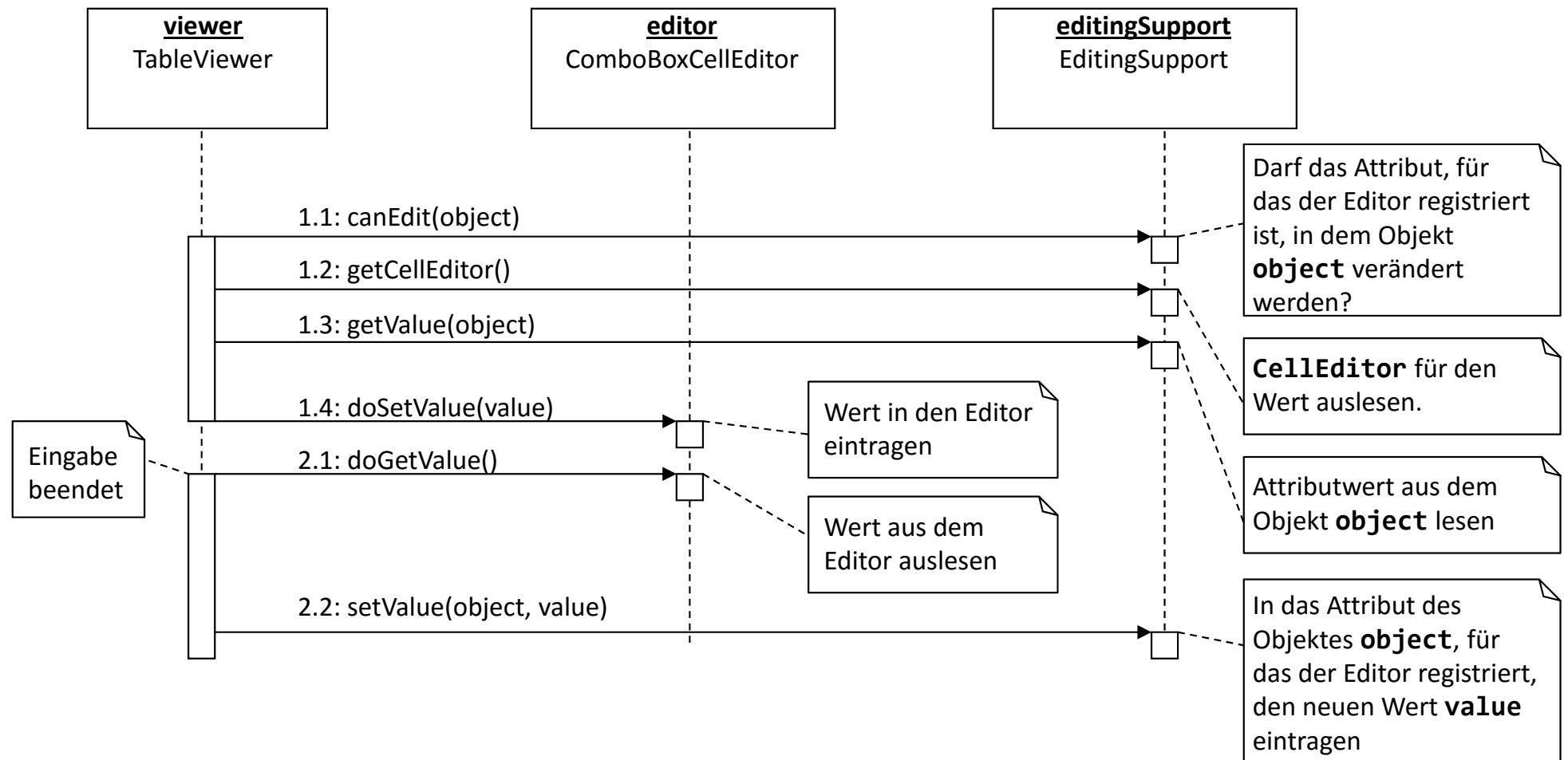
- Direkt von JFace werden die folgenden Editortypen unterstützt:
 - ◆ **TextCellEditor**: einzeilige Eingabe in ein Textfeld
 - ◆ **CheckboxCellEditor**: Auswahl eines Boole'schen Wertes
 - ◆ **ComboBoxCellEditor**: Auswahl aus einer Menge von Vorgaben durch ein CCombo-Widget
 - ◆ **ColorCellEditor**: Auswahl einer Farbe durch einen Dialog
 - ◆ **DialogCellEditor**: Auswahl durch einen eigenen Dialog, Anzeige des Wertes in der Tabelle normalerweise durch ein Label (Wert) und eine Taste (Start des Auswahldialogs)
- Weitere Editoren können durch Überschreiben von **CellEditor** selbst implementiert werden.

Model-View-Controller

Beispiel JFace-Tabelle: Eingabe



- Prinzipieller Ablauf bei der Arbeit mit Tabellen-Editoren (stark vereinfacht):



Model-View-Controller

Beispiel JFace-Tabelle: Eingabe



- Zunächst bekommt die Person ein zusätzliches Attribut, damit später auch eine Combo-Box zur Eingabe verwendet werden kann:

```
public class Person {  
    public enum Status { STUDYING, WORKING, RETIRED, DEAD };  
  
    private String name;  
    private int age;  
    private Status status;  
    // usw. (Getter, Setter)  
}
```

- Das soll das Ergebnis werden (Anzeige, Eingabe des Zustandes):

Daten			
Name	Age	Status	
Name 1	42	WORKING	
Name 2	66	RETIRED	
Name 3	33	DEAD	
Name 4	77	WORKING	

Daten			
Name	Age	Status	
Name 1	42	WORKING	▼
Name 2	66	STUDYING	
Name 3	33	WORKING	
Name 4	77	RETIRED	
		DEAD	



- Editing-Support zur Namenseingabe (`@Override` weggelassen):

```
public class PersonNameEditingSupport extends EditingSupport {  
    // Zur Bearbeitung des Namens wird ein Text-Widget verwendet.  
    private TextCellEditor cellEditor;  
  
    public PersonNameEditingSupport(TableViewer viewer) {  
        super(viewer);  
        cellEditor = new TextCellEditor(viewer.getTable());  
    }  
  
    protected boolean canEdit(Object element) { // Name kann immer  
        return true;                                // bearbeitet werden.  
    }  
  
    protected CellEditor getCellEditor(Object element) {  
        return cellEditor;  
    }  
  
    protected Object getValue(Object element) { // Das Namensattribut  
        return ((Person) element).getName();      // des Objektes  
    }                                              // auslesen.
```



```
protected void setValue(Object element, Object value) {  
    // Den neuen Wert in das Namensattribut eintragen und  
    // TableViewer veranlassen, die Zeile mit dieser  
    // Person neu zu zeichnen.  
    ((Person) element).setName((String) value);  
    getViewer().update(element, null);  
}  
}
```

- Und den Editor an der Spalte registrieren:

```
tableColumn.setEditingSupport(new PersonNameEditingSupport(tableViewer));
```



- Eingabe des Zustandes über eine Combo-Box (der eingegebene Wert ist der Index des ausgewählten Wertes):

```
public class PersonStatusEditingSupport extends EditingSupport {  
    // Zur Bearbeitung des Status wird ein CCombo-Widget verwendet.  
    private ComboBoxCellEditor cellEditor;  
  
    public PersonStatusEditingSupport(TableViewer viewer) {  
        super(viewer);  
        cellEditor = new ComboBoxCellEditor(viewer.getTable(),  
            person.getStatusValuesAsStringArray(), SWT.READ_ONLY);  
    }  
  
    protected boolean canEdit(Object element) { // Der Status kann immer  
        return true; // bearbeitet werden.  
    }  
  
    protected CellEditor getCellEditor(Object element) {  
        return cellEditor;  
    }  
}
```

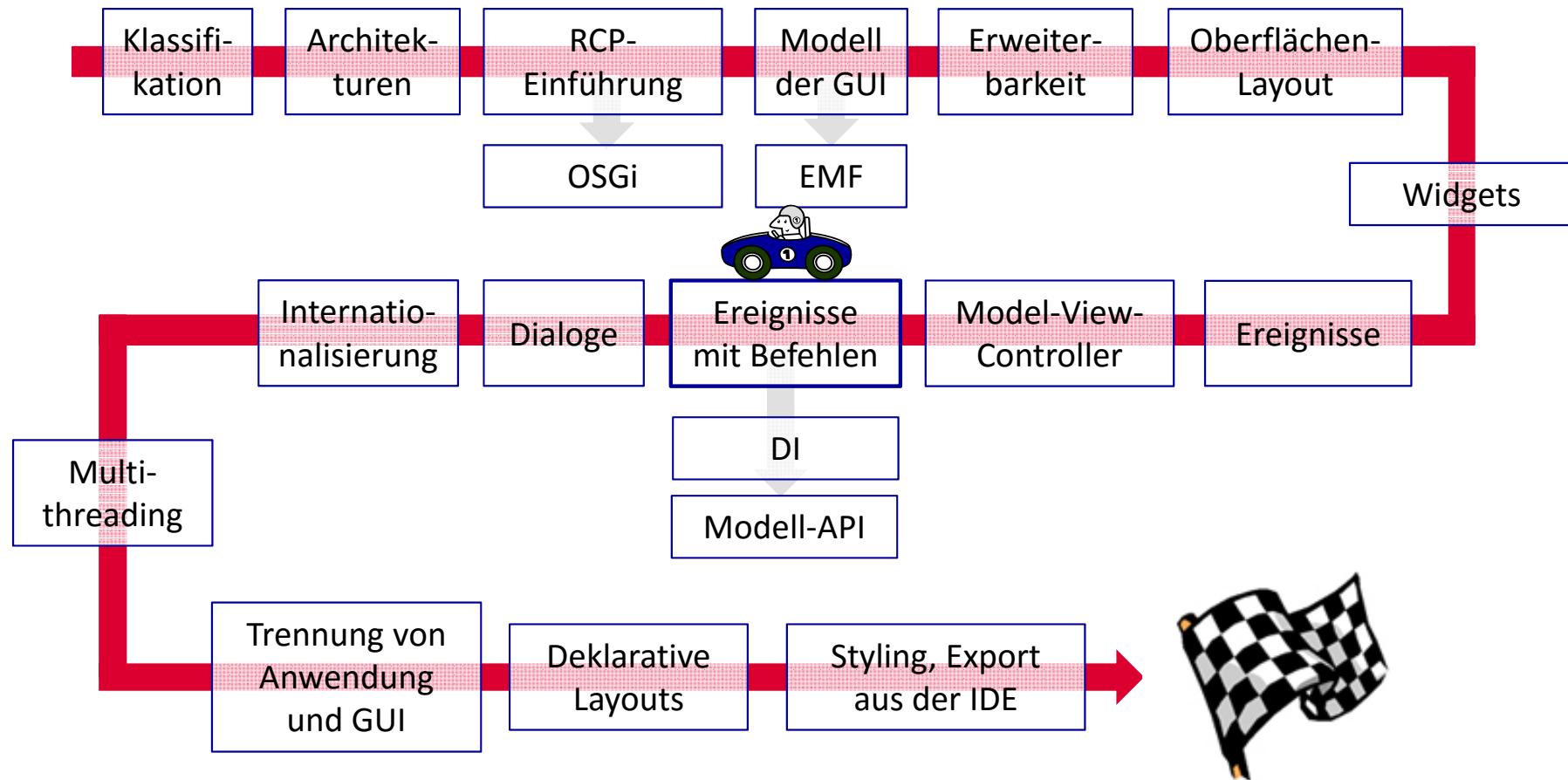


```
protected Object getValue(Object element) {  
    // Den Wert des Statusattributes auslesen und  
    // als Index innerhalb der ComboBox zurueck geben.  
    return ((Person) element).getStatus().ordinal();  
}  
  
protected void setValue(Object element, Object value) {  
    // Den neuen Wert in das Statusattribut eintragen und  
    // TableViewer veranlassen, die Zeile mit dieser  
    // Person neu zu zeichnen.  
    ((Person) element).setStatus((Integer) value);  
    getViewer().update(element, null);  
}  
}
```

- Das vollständige Beispiel ist in dem RCP-Projekt zur Tabelle zu finden.
- Einfach zu verwendende Erweiterung der JFace-Tabelle:
<http://www.ralfebert.de/blog/eclipsercp/tableviewerbuilder/>

Ereignisbehandlung durch Befehle

Einführung





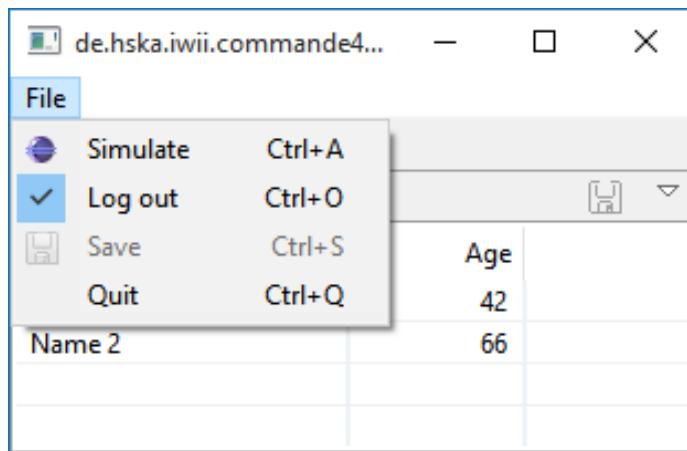
- Menüs, Toolbars und Kontextmenüs können mit SWT bzw. JFace programmgesteuert erzeugt werden (siehe Skript).
- Warum ist das für eine modulare Anwendung wie mit RCP keine gute Idee?
 - ◆ RCP-Anwendungen sind keine monolithischen Gebilde. Sie bestehen aus vielen Plug-ins, die Funktionalität und Oberflächenelemente steuern:
 - Neben eigenen Plug-ins gibt es auch welche von Drittanbietern.
 - Die Plug-ins müssen sich in die bestehende Menüstruktur „einklinken“ können.
 - Die Menüs sind kontextsensitiv und müssen sich den aktiven Ansichten bzw. Editoren anpassen → andere Funktionen, aktivierte/gesperrte Menüeinträge, ein- und ausgeblendete Menüeinträge.
 - Dasselbe Problem gibt es bei Ansichten und Editoren: Sie stammen auch von verschiedenen Plug-ins.



- ◆ Die programmierte Erzeugung würde durch Ausführen von Code aus verschiedenen Plug-ins erstellt. Dazu muss jedes Plug-in aktiviert werden → lange Startzeit.
- ◆ Dieselbe Aktion kann durch verschiedene Quellen ausgelöst werden (Menüeintrag, Toolbareintrag, Tastenkombination, ...) → undurchsichtige Beobachterstruktur.
- Idee im Befehlsframework:
 - ◆ Trennung von Darstellung und Funktion
 - ◆ Deklarative Beschreibung der Zusammenhänge im Anwendungsmodell
 - ◆ Deklarativ formulierte Bedingungen lassen Befehle aktiv oder passiv werden.
- Nähere Informationen unter (nur der erste Artikel ist auf dem Stand von Eclipse 4):
 - ◆ <http://www.vogella.de/articles/RichClientPlatform/article.html#commands>
 - ◆ http://wiki.eclipse.org/Platform_Command_Framework
 - ◆ <http://www.ralfebert.de/rcpbuch/>

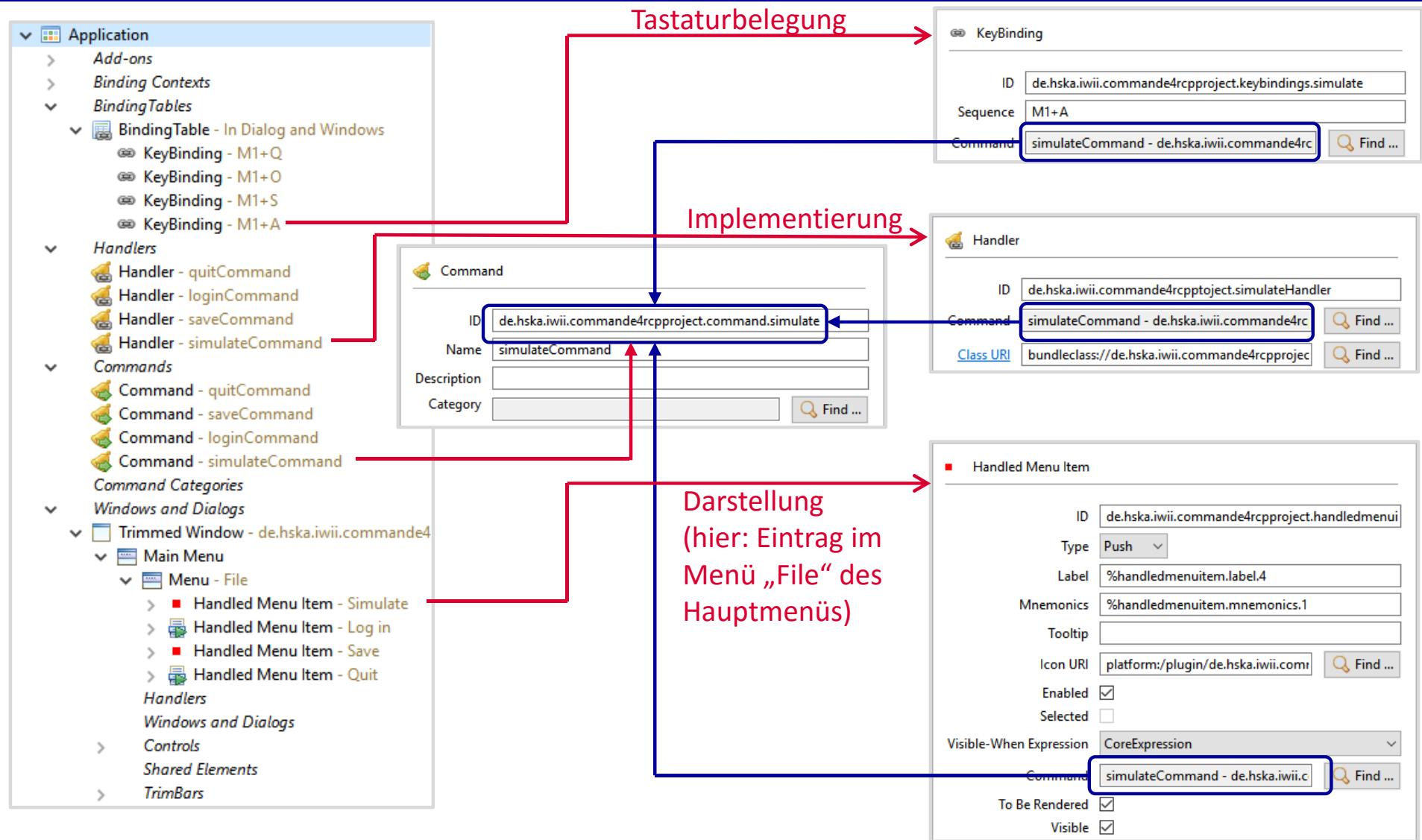


- Befehle bestehen aus:
 - ◆ Command: deklarative Beschreibung des Befehls
 - ◆ Handler: Implementierung des Verhaltens bei Auslösung des Befehls
 - ◆ UI-Zuordnung: Wo und wie soll der Befehl in die Oberfläche eingebunden werden?
 - ◆ Binding: Tastenkombination zur Auslösung des Verhaltens
- Befehle werden im Anwendungsmodell eingetragen (für Menüs, Toolbars, Kontextmenüs).
- Beispiel (Projekt [de.hska.iwii.commande4rcpproject](#), optisch unschön):



Ereignisbehandlung durch Befehle

Einführung





- Erläuterung der Modellelemente:
 - ◆ **BindingTable**: Tabelle, die Tastenkombinationen den Kommandos zuordnet
 - ◆ **Handlers**: Zuordnung des eigentlichen Befehls in Form des Befehls-Musters zu einem Kommando
 - ◆ **Commands**: logische Darstellung eines Befehls (z.B. „Kopieren“, die konkrete Ausprägung wie „Kopieren einer Datei“, „Kopieren eines Textes im Editor“ erfolgt durch einzelne Handler)
 - ◆ **Menu, HandledMenuItem, ...**: Die Darstellung der Menüs, Toolbars und Popup-Menüs sowie deren jeweilige Zuordnung zu den Kommandos
- Die Zuordnung zu einem Kommando erfolgt über dessen **Id**.

Ereignisbehandlung durch Befehle

Commands



- Das Kommando (Command) hat einige Attribute:
 - ◆ **ID**: eindeutige Identifikation, unter der dieses Kommando angesprochen wird, am besten Paketname + Bezeichnung
 - ◆ **Name**: Name des Kommandos, wird als Bezeichnung in einem Menü verwendet, wenn dort keine andere angegeben ist
 - ◆ **Description**: beschreibender Text, der in einer Toolbar z.B. als Tooltip-Text verwendet wird, wenn dort kein anderer angegeben ist
 - ◆ **Category**: Kommandos können Kategorien zugeordnet werden, die sich zur Laufzeit auslesen lassen (z.B. zur optischen Gruppierung).
 - ◆ **Parameters**: Parameter, die zur Laufzeit durch Handler ausgelesen werden können.
- Beispiel:

	Command
ID	de.hsk.a.iwii.commande4rcpproject.command.simulate
Name	simulateCommand
Description	
Category	<input type="text"/>



- Für viele Standardaktionen existieren bereits Kommandos mit teilweise vordefiniertem Verhalten:
 - ◆ Save: **org.eclipse.ui.file.save**, aktuelle Datei speichern
 - ◆ Save All: **org.eclipse.ui.file.saveAll**, alle Dateien speichern
 - ◆ Undo: **org.eclipse.ui.edit.undo**, letzte Änderung rückgängig machen
 - ◆ Redo: **org.eclipse.ui.edit.redo**, letzte Rücknahme wiederherstellen
 - ◆ Cut: **org.eclipse.ui.edit.cut**, selektiertes Element ausschneiden
 - ◆ Copy: **org.eclipse.ui.edit.copy**, selektiertes Element kopieren
 - ◆ Paste: **org.eclipse.ui.edit.paste**, Element aus der Zwischenablage einfügen
 - ◆ Delete: **org.eclipse.ui.edit.delete**, selektiertes Element löschen
 - ◆ Select All: **org.eclipse.ui.edit.selectAll**: alle Elemente im Dokument selektieren
- Vollständige Liste der vordefinierten Kommandos als Konstanten in **org.eclipse.ui.IWorkbenchCommandConstants**.

Ereignisbehandlung durch Befehle

Commands



- Warum ist es sinnvoll, die vordefinierten Kommandos einzusetzen?
 - ◆ Sie haben teilweise ein sinnvolles Verhalten implementiert.
 - ◆ Beispiele:
 - Copy/Paste/...: wird bei Textdokumenten direkt durchgeführt.
 - Save/SaveAll: RCP unterstützt das Verwalten geänderter Dokumente und einen Speichermechanismus (kommt noch...).
- Dort, wo Kommando-IDs eingetragen werden können, dürfen vordefinierte Kommandos verwendet werden.
- Wenn vordefinierte Kommandos kein Verhalten besitzen (keinen Handler haben), dann erscheinen sie in Menüs usw. als deaktiviert.



- Ein Handler kapselt die eigentliche auszuführende Anweisung. Er hat die folgenden Attribute:
 - ◆ **Id**: eindeutige Identifikation, nur erforderlich, wenn der Handler per API angesprochen (z.B. ausgelöst) oder aus einem anderen Plug-in heraus verwendet werden soll.
 - ◆ **Command**: **Id** des Kommandos, zu dem dieser Handler gehört
 - ◆ **Class URI**: Klasse, die die Anweisung implementiert
 - ◆ **Persisted State**: Schlüssel/Werte-Paare, die vom Handler persistent gehalten werden (wird hier nicht besprochen)
- Ein Handler kann selbst entscheiden, ob er aktiv ist oder nicht:
 - ◆ Die Bedingungen werden zur Laufzeit gesetzt.
 - ◆ Ein Deaktivieren bewirkt gleichzeitig ein sperren der Taste im Menü.
 - ◆ Damit können z.B. Tasten erst nach einer erfolgreichen Anmeldung freigegeben werden.

Ereignisbehandlung durch Befehle

Handler



- Beispiel für einen Handler mit Implementierung:

Handler

ID de.hskaiwii.commande4rcpproject.simulateHandler

Command simulateCommand - de.hskaiwii.commande4rc

Class URI bundleclass://de.hskaiwii.commande4rcpproject

```
public class SimulateCommandHandler {  
    // Methode, die bei Ausführung des Kommandos aufgerufen wird.  
    @Execute  
    public void execute() {  
        // Code ausführen  
    }  
  
    // Darf der Handler ausgeführt werden?  
    @CanExecute  
    public boolean canExecute() {  
        return true;  
    }  
}
```

Ereignisbehandlung durch Befehle

Handler



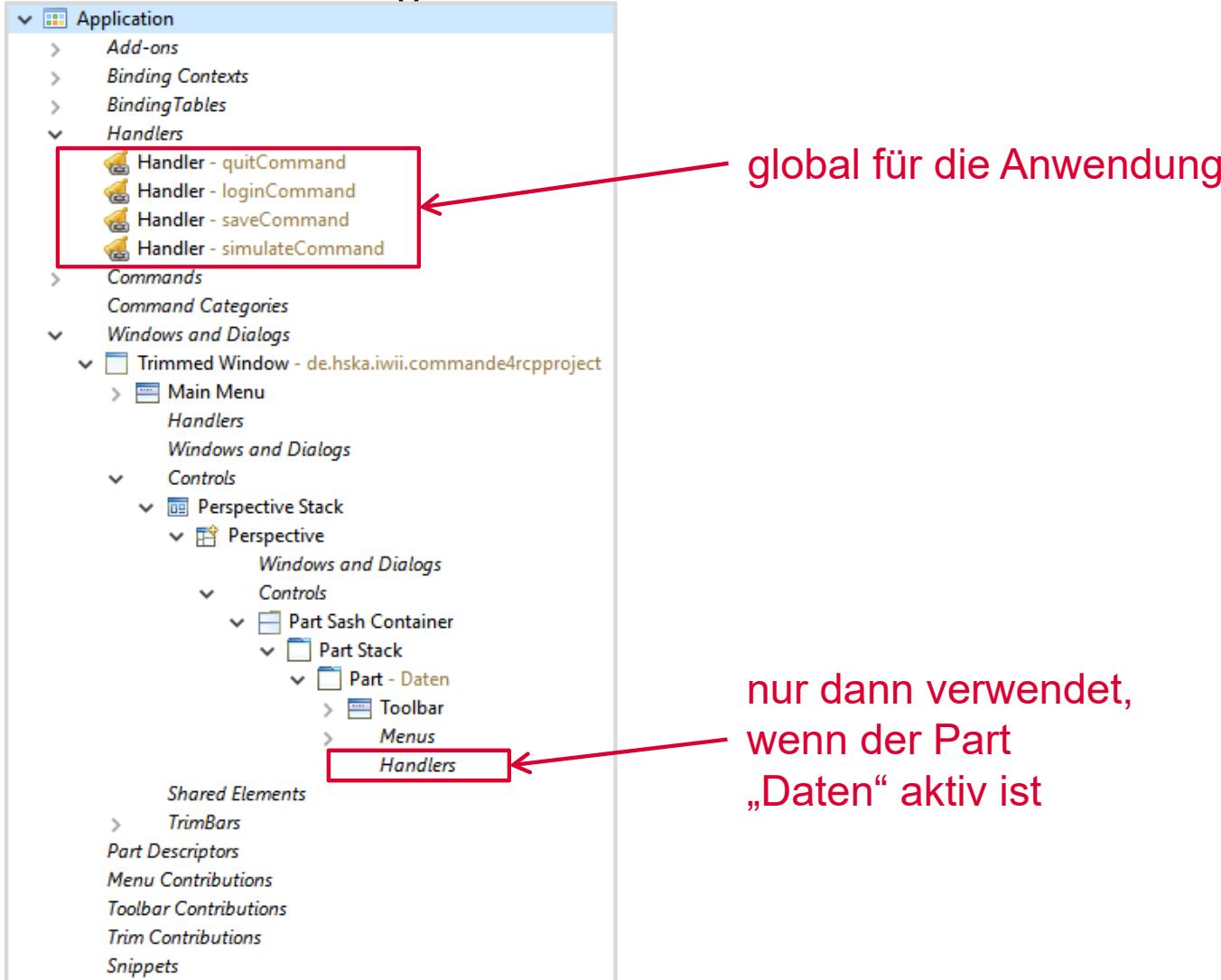
- Der Handler besitzt Methoden mit zwei Annotationen:
 - ◆ **@Execute**: Diese Methode implementiert den Code, der bei Aktivierung des Handlers ausgeführt wird. Der Name der Methode ist nicht festgelegt.
 - ◆ **@CanExecute**: Mit dieser optionalen Methode bestimmt ein Handler selbst, ob er ausgeführt werden kann oder nicht.
 - Fehlt die Methode, dann ist der Handler immer ausführbereit.
 - Der Name der Methode ist nicht festgelegt.

Ereignisbehandlung durch Befehle



Handler

- Handler haben Gültigkeitsbereiche:



Ereignisbehandlung durch Befehle

Handler



- Wozu dienen die Gültigkeitsbereiche? Beispiel:
 - ◆ Copy/Paste: dieselbe logische Aktion, aber anders implementiert (grafische Editoren, Texteditoren, ...)
 - ◆ Das Menü zeigt immer Copy/Paste.
 - ◆ Es ist dasselbe Kommando.
 - ◆ Es werden je nach aktiver Ansicht oder aktivem Editor aber automatisch die richtigen Handler verwendet.
- Suchreihenfolge nach einem Handler: „von innen nach außen“ im Baum, also vom speziellsten Eintrag hin zur Anwendung selbst

Ereignisbehandlung durch Befehle



Tastaturbindungen

- Die Bindung beschreibt die Tastenkombination zur Auslösung des Ereignisses.

Attribute:

- ◆ **Id**: eindeutige Identifikation, nur erforderlich, wenn die Bindung per API angesprochen werden oder aus einem anderen Plug-in heraus verwendet soll.
- ◆ **Sequence**: Tastenkombination, z.B. **M3-A** für **Ctrl-A**, **M1-A** für **Alt-A**:

Logische Taste	Windows und Linux	MacOS
M1	Ctrl	Command
M2	Shift	Shift
M3	Alt	Alt
M4	-	Ctrl

- ◆ **Command**: **Id** des Kommandos, dem die Tastenkombination zugeordnet ist

Ereignisbehandlung durch Befehle

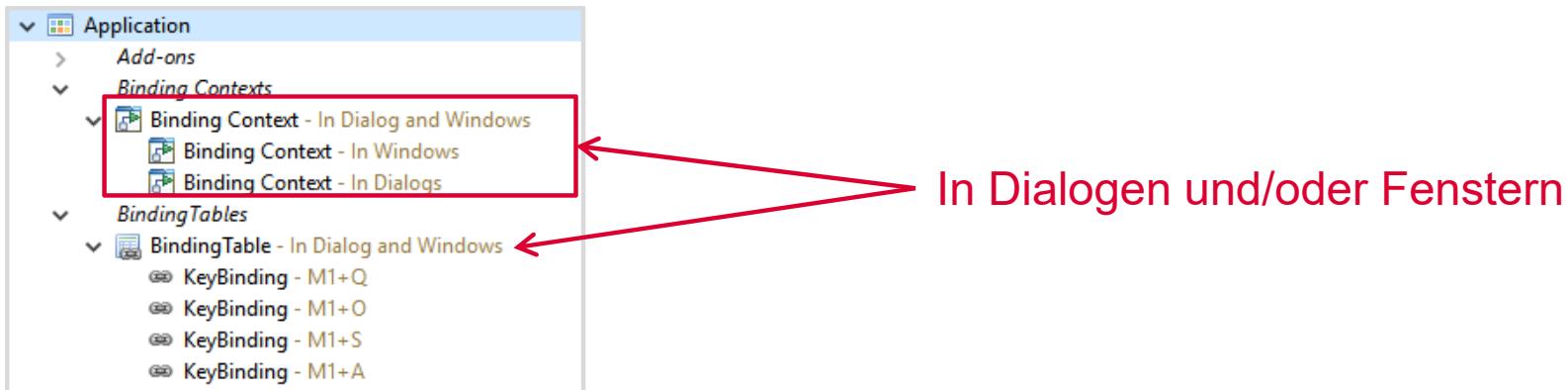
Tastaturbindungen



- Beispiel für eine Binding:



- Tastenbindungen sind Kontexten zugeordnet:



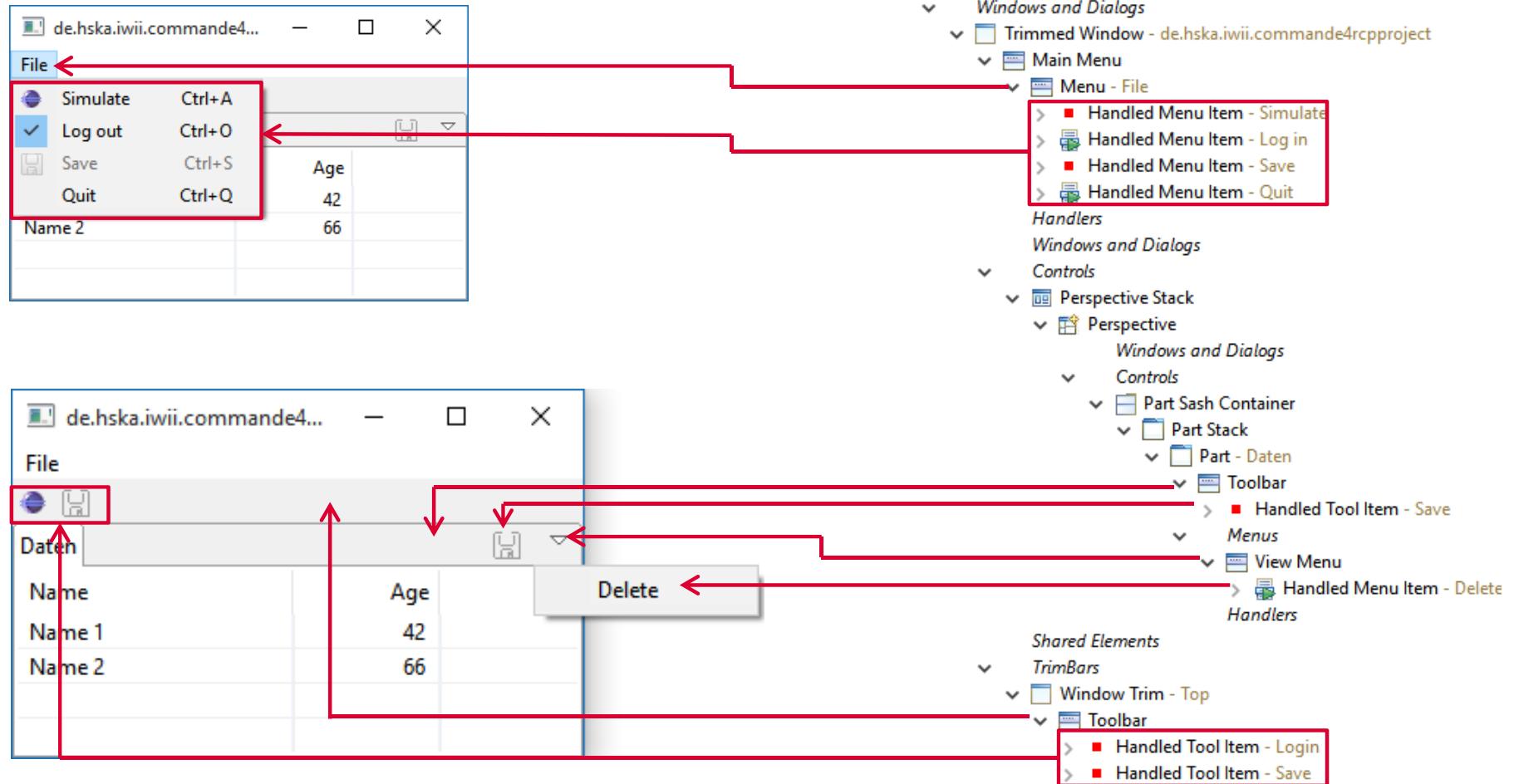
- Damit Bindungen funktionieren, müssen alle beteiligten Parts eine mit **@Focus** annotierte Methoden besitzen und dort einem Widget den Tastaturfokus zuweisen.

Ereignisbehandlung durch Befehle



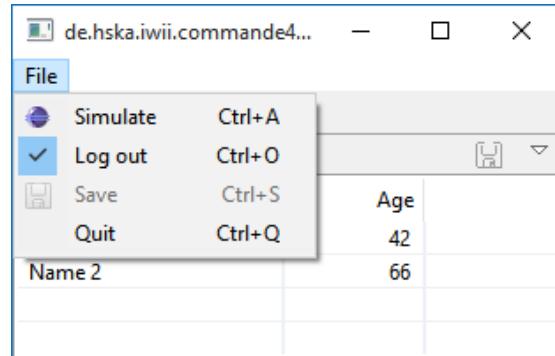
Menüs und Toolbars

- Ein Menü bzw. eine Toolbar kann an unterschiedlichen Positionen sitzen:





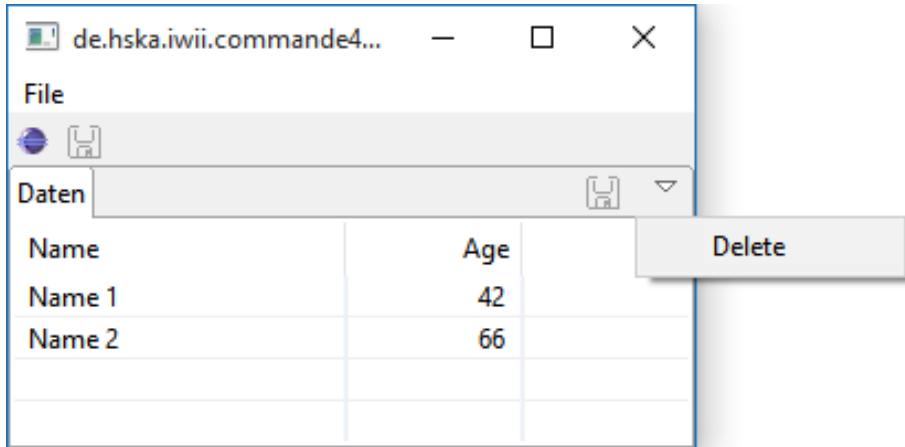
- Beschreibung eines Menüs bzw. einer Toolbar:
 - ◆ **Id**: eindeutige Kennung, einige IDs sind vordefiniert und sollten so beibehalten werden:
 - **org.eclipse.ui.main.menu**: im obersten Hauptmenü



- **org.eclipse.ui.main.toolbar**: in oberster Toolbar
- **org.eclipse.ui.popup.any**: in allen Popup-Menüs



- **<id>**: sonstige ID für andere Menüs oder Toolbars:



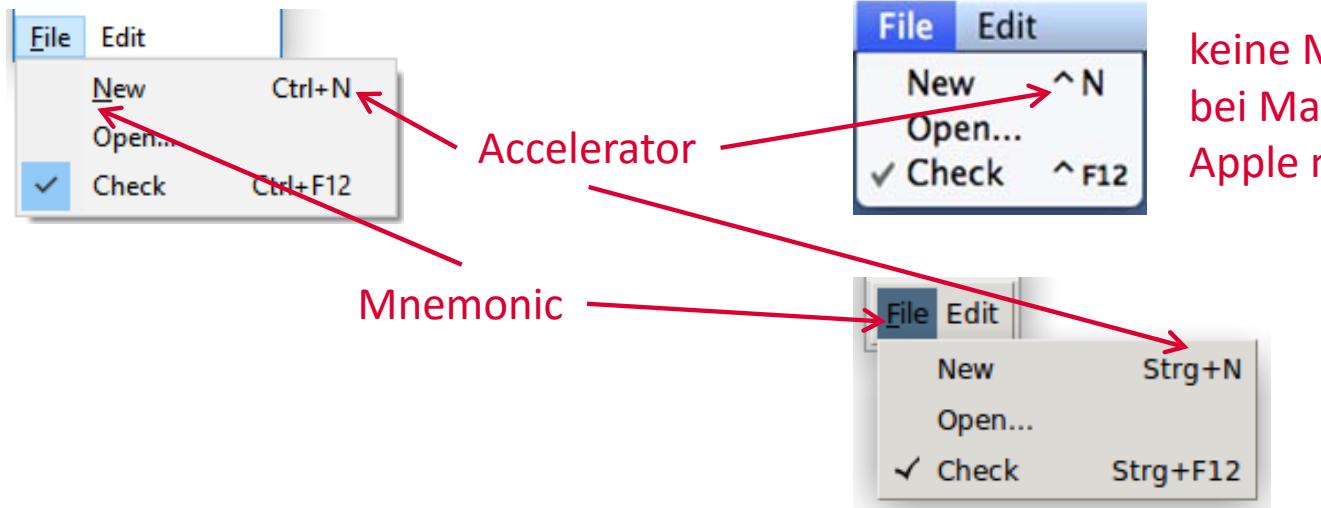
- ◆ **Label**: textuelle Bezeichnung des Menüs
- ◆ **Mnemonic**: Tastenkürzel zur Navigation
- ◆ **Icon URI**: darzustellendes Icon im Format
platform:/plugin/<plugin>/pfad/datei.erweiterung
- ◆ **Visible-When-Expression**: Bedingung, unter der das Menü sichtbar sein soll
(kommt gleich)
- ◆ **To be Rendered**: Soll die Menüdarstellung erzeugt werden?
- ◆ **Visible**: Soll die berechnete Menüdarstellung angezeigt werden?

Ereignisbehandlung durch Befehle



Menüs und Toolbars

■ Begriffe



keine Mnemonics
bei Mac OS X (von
Apple nicht empfohlen)



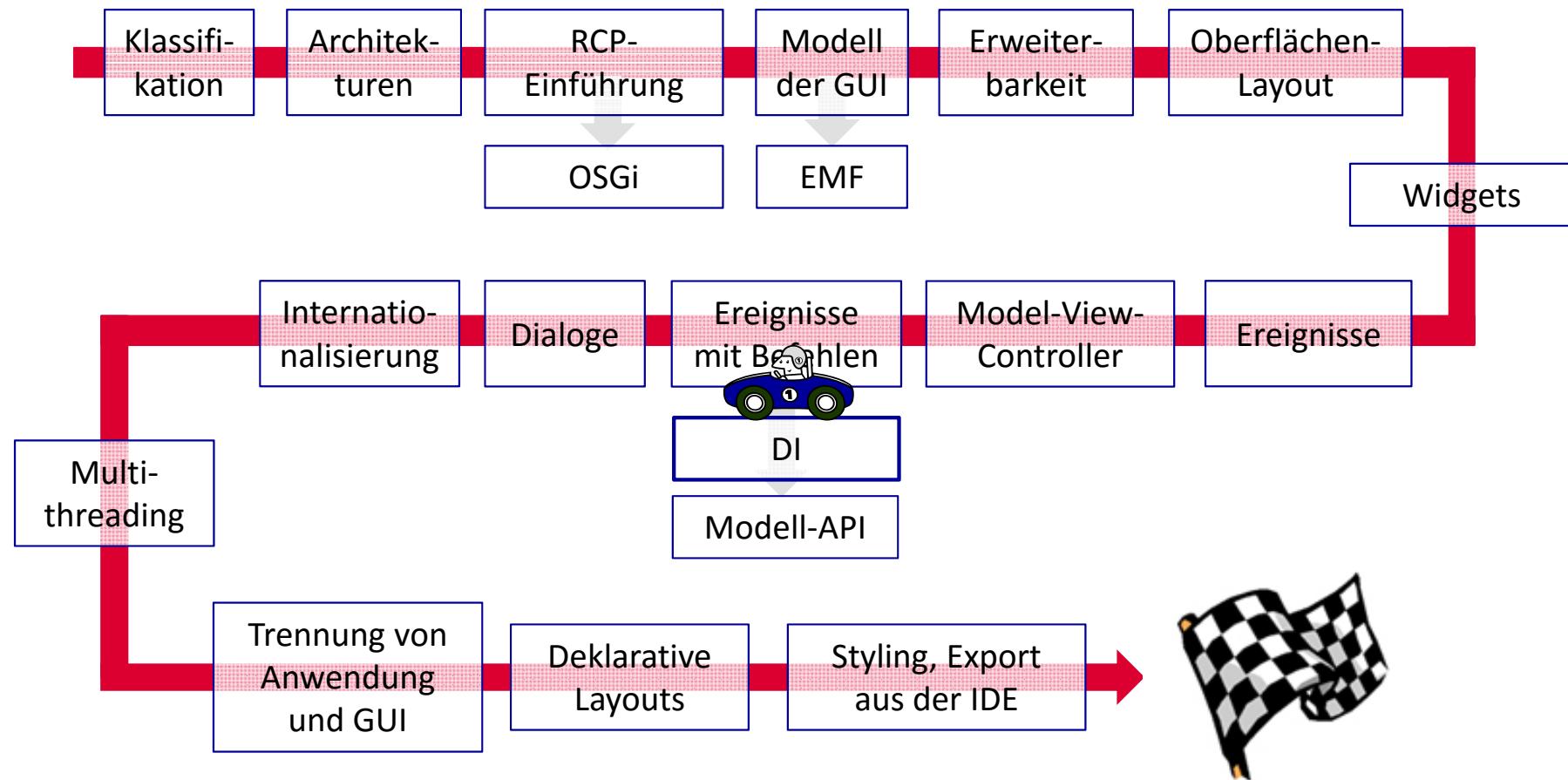
- Arten von Menüeinträgen:
 - ◆ **HandledMenuItem**: Menüeintrag, dem ein Kommando zugeordnet ist
 - ◆ **DirectMenuItem**: Menüeintrag, dem direkt ein Handler zugeordnet ist → unflexibler
 - ◆ **Separator**: Trennstrich
 - ◆ **Menu**: Untermenü
- Arten von Toolbar-Einträgen:
 - ◆ **Handled ToolItem**: Toolbareintrag, dem ein Kommando zugeordnet ist
 - ◆ **Direct ToolItem**: Toolbareintrag, dem direkt ein Handler zugeordnet ist → unflexibler
 - ◆ **ToolControl**: Element innerhalb der Toolbar, das direkt von einer Klasse erzeugt wird
 - ◆ **Separator**: Trennstrich



- Attribute:
 - ◆ **Id**: eindeutige Identifikation des Eintrags
 - ◆ **Type**: **Check**, **Radio** oder **Push**, siehe SWT-Tasten
 - ◆ **Label**: Bezeichnung
 - ◆ **Mnemonic**: Tastenkürzel zur Navigation, kann auch durch ein vorangestelltes & im Label erreicht werden.
 - ◆ **Tooltip**: Tooltip-Text
 - ◆ **Icon URI**: Icon vor dem Label darzustellendes Icon im Format
platform:/plugin/<plugin>/pfad/datei.erweiterung
 - ◆ **Enabled**: Eintrag ist aktiv/gesperrt.
 - ◆ **Selected**: Eintrag ist gewählt oder nicht (nur für die Typen **Check** oder **Radio**).
 - ◆ **Visible-When-Expression**: Bedingung, unter der der Eintrag sichtbar ist
 - ◆ **Command**: Kommando, dem der Eintrag zugeordnet ist
 - ◆ **To be Rendered**: Soll die Darstellung des Eintrags erzeugt werden?
 - ◆ **Visible**: Soll die berechnete Darstellung angezeigt werden?

Dependency Injection

Einführung

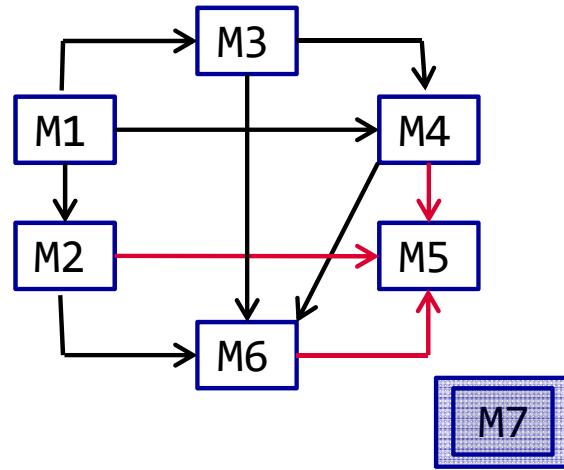


Dependency Injection

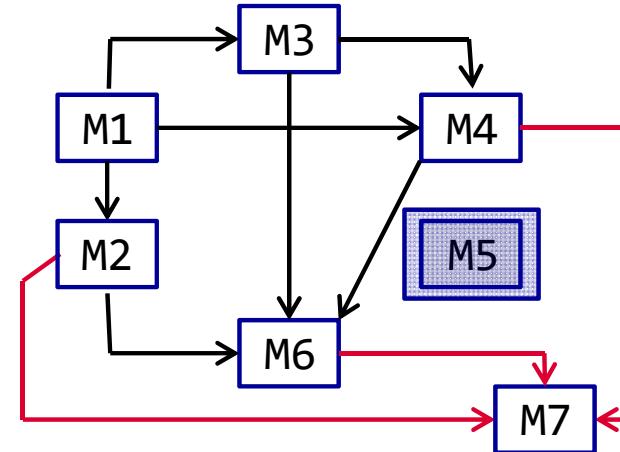


Einführung

- Fragestellungen:
 - Wie sollen Module gekoppelt (verbunden) werden? Wer macht das?
 - Soll jedes Modul seine notwendigen Abhängigkeiten selbst herstellen?
 - Problem: Je nach Kunden, auszuliefernder Version und Zustand (Test/Betrieb) der Anwendung sind unterschiedliche Kopplungen sinnvoll.



Betrieb (ohne **M7**)



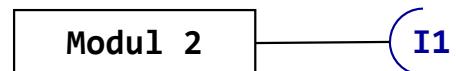
Test (ohne **M5**)



- Weitere Probleme:
 - ◆ Je nach Konfiguration sollen die Module unterschiedliche Parameter erhalten (andere Datenbank, andere Benutzeranmeldung, ...).
 - ◆ Im komplexen Graphen von Abhängigkeiten entsteht das Problem der Initialisierungsreihenfolge der Module.
- Lösungsidee („Trennung der Schnittstelle von der Implementierung“):
 - ◆ Angebot: Die Module geben Schnittstellen nach außen bekannt, die andere Module nutzen dürfen.

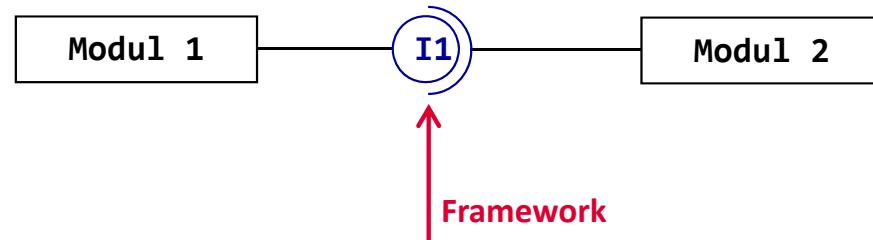


- ◆ Nachfrage: Module definieren die Schnittstellen, die sie von anderen Modulen benötigen.





- Ein externes Framework verknüpft automatisch Angebot und Nachfrage: keine direkte Kopplung durch die Module selbst



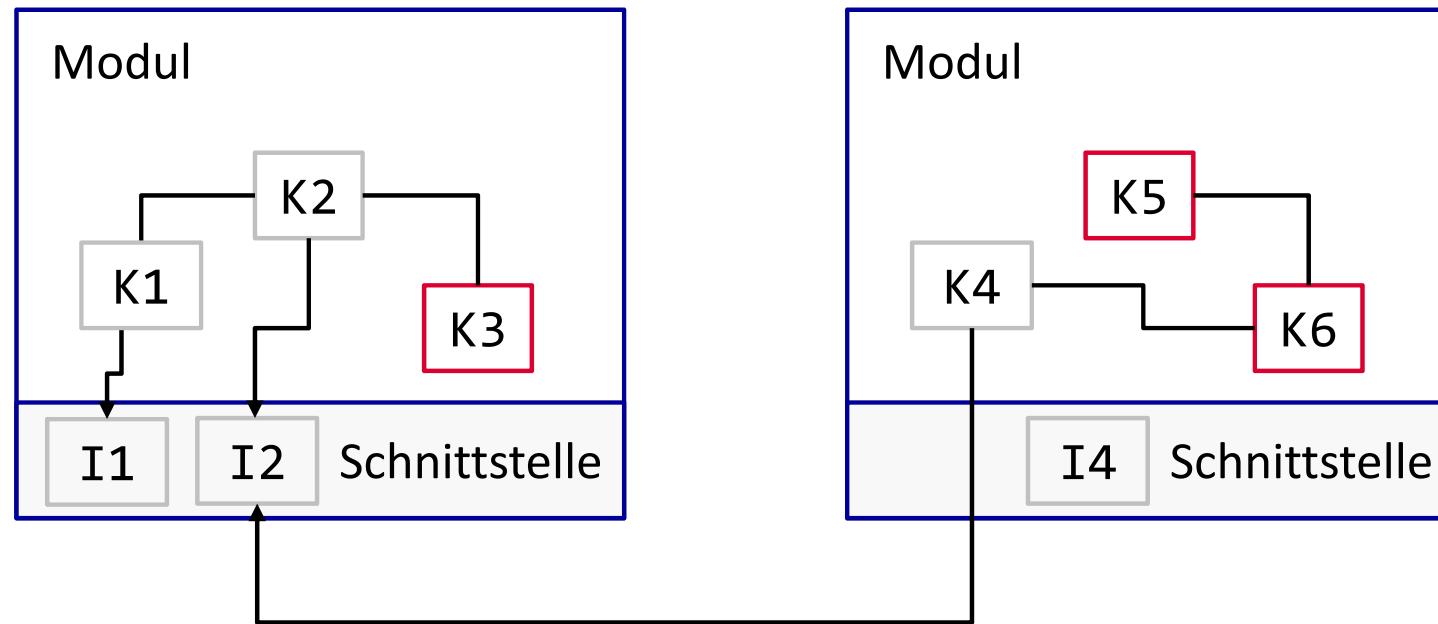
- Modul 2** löst nicht selbst die Abhängigkeit zu **Modul 1**: die Abhängigkeit wird „von außen“ injiziert → Dependency Injection!
- Anmerkung: „Schnittstelle“ muss nicht zwangsläufig ein Interface in Java sein. Es ist lediglich eine definierte, öffentliche Schnittstelle, die genutzt werden darf:
 - Java-Interfaces
 - Klassen
 - abstrakte Klassen

Dependency Injection



Einführung

- Beispiel-Module mit öffentlichen Schnittstellen:



K4 verwendet **K2**,
kennt aber nur dessen
Schnittstelle **I2**



- Beispiel für eine Ansicht:

```
public class CustomerView {  
    @Inject private Logger logger;  
    @Inject private CustomerData data;  
  
    @Inject  
    public CustomerView(Component parent) {  
        // Baue die Oberfläche  
    }  
}
```

- Die RCP sorgt dafür, dass die mit **@Inject** markierten Abhängigkeiten automatisch aufgelöst werden.
- Das funktioniert für:
 - ◆ statische und nicht-statische Attribute
 - ◆ Konstruktoren
 - ◆ statische und nicht-statische Methoden

Dependency Injection



Annotationen

Annotation	Beschreibung
@Inject	Das Eclipse-Framework injiziert ein passendes Objekt. Existiert keines, dann tritt eine Ausnahme auf.
@Named	Gibt den Namen der zu injizierenden Klasse vor. Normalerweise wird der voll-qualifizierte Klassename verwendet. Einige Standard-Werte für RCP sind in der Schnittstelle IServiceConstants zu finden.
@Optional	Der injizierte Wert ist optional. Gibt es keine passende Klasse: <ul style="list-style-type: none">• bei Attributen: Das Attribut wird nicht beschrieben.• bei Methoden: Die Methode wird nicht aufgerufen.• bei Parametern: Es wird null übergeben.
@Preference	Eclipse speichert Schlüssel/Werte-Paare in Preferences-Dateien. @Preference markiert einen Wert, der aus einer solchen Datei gelesen werden soll.
@Creatable	Annotiert eine eigene Klasse, deren Objekt automatisch erzeugt und mittels @Inject in andere Objekte injiziert werden soll.
@Singleton	Zusätzlich zu @Creatable , um nur noch ein Objekt der Klasse zu erzeugen (ansonsten wird bei jeder Injektion ein neues angelegt)



- Ablauf beim Injizieren:
 - ◆ Es werden die Objekte der Klassen erzeugt, die das Anwendungs-Modell referenziert.
 - ◆ Besitzen die referenzierten Klassen Annotationen zur „Dependency Injection“, dann werden deren Abhängigkeiten injiziert...
- Reihenfolge der Injektionen:
 - ◆ Konstruktor
 - ◆ Attribute
 - ◆ Methoden
- Automatisch unterstützte Dynamik zur Laufzeit: Ändern sich die injizierten Objekte, dann werden die Injektionen erneut durchgeführt.
- Beispiel:
 - ◆ Es wird die aktuelle Selektion injiziert.
 - ◆ Ändert sich die Selektion, dann wird diese erneut injiziert.
- Konsequenz: Keine manuelle Listener-Verwaltung!



- Eclipse speichert die zur Injektion verfügbaren Objekte in einem Kontext. Dazu gehören u.A.
 - ◆ alle Objekte, die zum Anwendungs-Modell gehören,
 - ◆ alle Objekte, die manuell (z.B. per API-Aufruf) zum Kontext hinzugefügt wurden.
 - ◆ alle Preferences-Werte zum Konfigurieren der Anwendung und
 - ◆ alle OSGi-Dienste.
- Der Kontext verwaltet Schlüssel/Werte-Paare:
 - ◆ Schlüssel ist der Name einer Klasse, der Wert ein Objekt der Klasse → anhand eines gesuchten Klassennamens wird das abgelegte Objekt injiziert
 - ◆ Schlüssel als String ist ein Schlüssel aus der Preferences-Datei → der Wert aus der Datei wird injiziert
 - ◆ Der Zugriff erfolgt ähnlich wie bei einer **Map** in der Collections-API.



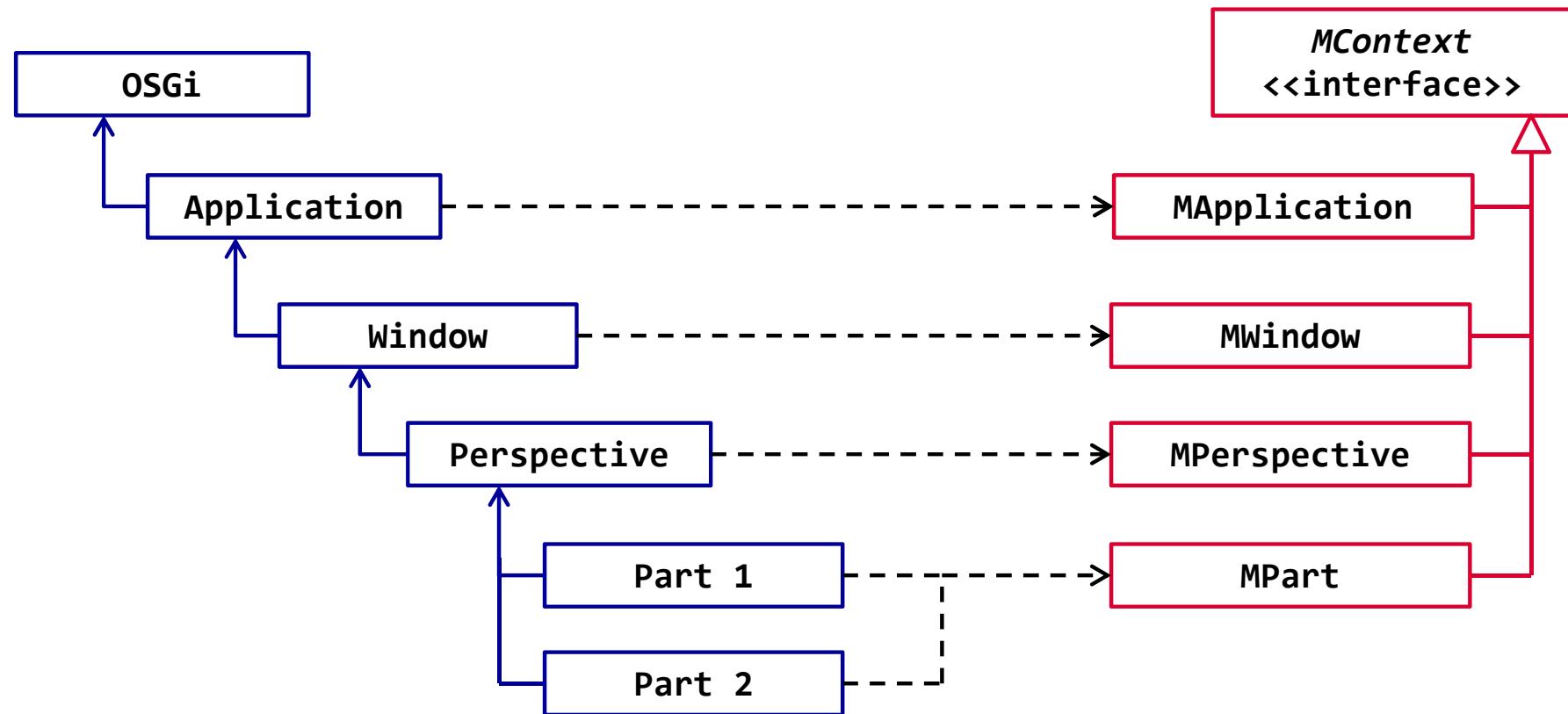
- Ändert sich der Wert zu einem Schlüssel, dann werden alle Injektionen zu diesem Schlüssel erneut durchgeführt.
 - ◆ Beispiel 1:
 - Einige Ansichten hängen von dem aktiven Editor ab.
 - Der aktive Editor wird im Kontext abgelegt und mittels Dependency Injection in die Ansichten injiziert.
 - Ändert sich der Editor, werden die Ansichten automatisch informiert.
 - ◆ Beispiel 2:
 - Die aktuelle Selektion wird im Kontext gespeichert.
 - Ändert sich die Selektion, wird auch diese in alle abhängigen Objekte neu injiziert.
 - So kann das Anwendungsmodell z.B. automatisch die richtigen Handler ermitteln (bei bedingten Handlern → kommt gleich).
- Die RCP besitzt eine ganze Anzahl vordefinierter Objekte im Kontext. Deren Schlüssel sind als Konstanten in der Schnittstelle **IServiceConstants** zu finden.

Dependency Injection

Kontext und Gültigkeitsbereich



- Der Kontext ist hierarchisch aufgebaut und orientiert sich am Anwendungs-Modell:





- Gründe für die hierarchische Konstruktion:
 - ◆ Daten können lokal dort abgelegt werden, wo sie benötigt werden.
 - ◆ Daten können mit demselben Schlüssel mehrfach vorkommen. Beispiel:
 - Jede Ansicht hat ihr eigenes Vaterobjekt, das als **Component** dem Konstruktor der Ansicht übergeben wird.
 - Der Vater wird im lokalen Kontext der Ansicht abgelegt.

```
public class Part1 {  
    @Inject  
    public CustomerView(Component parent) {  
        // Baue die Oberfläche  
    }  
}
```

- Es wird das im lokalen Kontext abgelegte **Component**-Objekt injiziert.
- Der Entwickler muss sich beim Injizieren nicht um die Hierarchie kümmern: Es wird immer von dem spezialisierten Kontext zum allgemeinen gesucht.

Dependency Injection

Kontext und Gültigkeitsbereich



- Weiteres Beispiel:

```
// Eigene Klasse, deren Objekt automatisch erzeugt und in andere Objekte
// injiziert werden kann.
@Creatable
public class Receiver {

    // Die Methode wird beim Anlegen des Objektes nur dann aufgerufen, wenn
    // ein Objekt der Klasse Xyz im Kontext vorhanden ist und so injiziert
    // werden kann. Ansonsten wird die Methode nicht aufgerufen.
    @Inject
    @Optional
    public void receive(Xyz message) {
        // ...
    }

    // Die Methode wird beim Erzeugen des Objektes auf jeden Fall aufgerufen.
    // Existiert kein Objekt der Klasse Xyz im Kontext, so wird null übergeben.
    @Inject
    private void receiveInternal(@Optional Xyz message) {
        // ...
    }
}
```

Dependency Injection



Annotationen zur Steuerung des Verhaltens

- In Eclipse 4 wurden für viele Klassen die starren Vererbungsbeziehungen abgeschafft.
- Stattdessen kann das Verhalten häufig flexibler durch Annotationen gesteuert werden.
- Diese legen fest, wann Methoden aufgerufen werden sollen.

Annotation	Beschreibung
@PostConstruct	wird aufgerufen, nachdem das Objekt erzeugt und alle Abhängigkeiten injiziert wurden → wenn alles initialisiert ist
@PreDestroy	wird aufgerufen, bevor die letzte Referenz auf das Objekt gelöscht wird → kann eigene Aufräumarbeiten durchführen
@Focus	wird aufgerufen, wenn ein Part den Tastaturfokus erhält → muss in Parts vorhanden sein, ansonsten sinnlos
@Persist	wird aufgerufen, wenn die Daten eines Parts gesichert werden sollen → Beispiel kommt noch
@PersistState	wird aufgerufen, damit ein Part seinen visuellen Zustand speichern kann → Beispiel kommt noch

Dependency Injection

Annotationen zur Steuerung des Verhaltens



- Beispiel:

```
public class Part1 {  
    @Inject private CustomerData data;  
  
    @Inject  
    public CustomerView(Component parent) {  
        // Baue die Oberfläche  
    }  
  
    // Der Konstruktor wurde aufgerufen, die Referenz in das Attribut  
    // data ist bereits injiziert.  
    @PostConstruct  
    public void showContents() {  
        // Mache etwas mit den Daten  
    }  
  
    @Focus  
    public void setFocus() {  
        // Tastaturfokus auf ein Widget setzen  
    }  
}
```



Speziellere Verhaltensannotationen:

Annotation	Beschreibung
@Execute	markiert die Methode in einer Handler-Klasse, die im Falle eines Ereignisses aufgerufen werden soll
@CanExecute	ermöglicht einem Handler, selbst zu bestimmen, ob er ausführbar ist oder nicht
@GroupUpdates	Wird für extrem häufig auftretende Wertänderungen im Kontext verwendet: Sollte sich ein Wert sehr schnell ändern, dann muss er immer wieder in die abhängigen Objekte injiziert werden, was u.U. viel Zeit kostet. Mit @GroupUpdates kann die Anzahl an Aufrufen pro Zeitraum für eine Injektion festgelegt werden.
@EventTopic und @UIEventTopic	zum Empfang von Ereignissen des Event-Admin-Dienstes → kommt noch

Paket **javax.annotation** importieren (sonst werden falsche Annotationen mit demselben Namen verwendet!!!)



- Beispiel (automatisch erzeugter Handler zum Sichern):

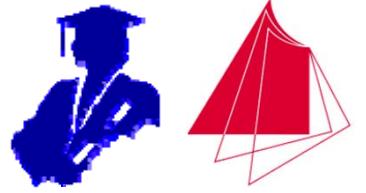
```
public class SaveHandler {  
    // Wird aufgerufen, um zu prüfen, ob der Handler aktiv ist:  
    // Der Handler soll nur dann aktiv sein, wenn der aktive Part  
    // verändert wurde ("dirty" ist).  
    // @Named wählt hier einen von mehreren möglichen Parts aus.  
    @CanExecute  
    public boolean canExecute(@Named(IServiceConstants.ACTIVE_PART)  
                               MDirtyable dirtyable) {  
        if (dirtyable == null) {  
            return false;  
        }  
        return dirtyable.isDirty();  
    }  
}
```

Dependency Injection



Annotationen zur Steuerung des Verhaltens

```
// Die Methode wird beim Ausführen eines Kommandos aufgerufen.  
// @param IEclipseContext: Anwendungs-Modell.  
// @param shell Das aktive Fenster, ausgewählt durch @Named.  
// @param contribution Der aktive Part, ausgewählt durch @Named.  
  
@Execute  
public void execute(IEclipseContext context,  
                    @Named(IServiceConstants.ACTIVE_SHELL) Shell shell,  
                    @Named(IServiceConstants.ACTIVE_PART) final MContribution contribution) {  
    // Sichern der Daten  
}  
}
```



Manuelles „Dependency Injection“ (Auswahl):

- **context.set(Class<?> c, obj):**
 - ◆ legt das Objekt **obj** unter dem Klassennamen **c** im Kontext ab.
 - ◆ Beispiel: **context.set(View.class, this);**
- **T ContextInjectionFactory.make(Class<T> c, IEclipseContext context):**
 - ◆ sucht das Objekt der Klasse **c**
 - ◆ erzeugt es, wenn es nicht existiert und injiziert dort eventuell notwendige Werte und legt es im Kontext an → nicht mit **new** erzeugen → kein DI!
 - ◆ Beispiel: **User u = ContextInjectionFactory.make(User.class, context);**
- **ContextInjectionFactory.uninject(Object obj, IEclipseContext context):**
 - ◆ sucht das Objekt **obj** im Kontext und entfernt es, wenn es nicht in anderen Objekten per DI injiziert wurde und dort benötigt wird

Dependency Injection

Annotationen zur Steuerung des Verhaltens



- Häufig muss eine Anwendung beim Start oder in bestimmten Zuständen Aktionen unternehmen.
- Dazu kann am Erweiterungspunkt **org.eclipse.core.runtime.products** eine Klasse registriert werden, deren Methoden mit Annotationen versehen werden.

The screenshot shows the Eclipse IDE's Extensions view for a plugin named "de.hska.iwii.commande4rcpproject". In the left pane, under "All Extensions", there is a tree structure. The "org.eclipse.core.runtime.products" node has a child node "de.hska.iwii.commande4rcpproject (product)". This product node contains several properties: "clearPersistedState", "applicationCSS", "appName", and "lifeCycleURI". The "lifeCycleURI" property is selected. In the right pane, the "Extension Element Details" section shows the configuration for this property. The "name" field is set to "lifeCycleURI" and the "value" field is set to "bundleclass://de.hska.iwii.commande4rcpproject/c". A red box highlights the "value" field, and a red arrow points from this box to the explanatory text below.

Bundle und Name der Klasse, die den Lebenszyklus der Anwendung überwacht

Dependency Injection

Annotationen zur Steuerung des Verhaltens



- Für diese Klasse zur Lebenszyklusüberwachung existieren vier zusätzliche Annotationen:

Annotation	Beschreibung
@PostContextCreate	wird aufgerufen, nachdem der Kontext angelegt wurde. Kann <ul style="list-style-type: none">• zusätzliche Schlüssel/Werte-Paare im Kontext speichern• z.B. einen Login-Dialog zur Anmeldung anzeigen• usw.
@ProcessAdditions	wird aufgerufen, bevor das Anwendungs-Modell gerendert wird, kann zusätzliche Elemente zum Modell hinzufügen (Perspektiven, Parts, ...)
@ProcessRemovals	wird aufgerufen, bevor das Anwendungs-Modell gerendert wird, kann Elemente aus dem Modell entfernen
@PreSave	wird aufgerufen, bevor das Anwendungs-Modell gespeichert wird, kann vorher das Modell modifizieren

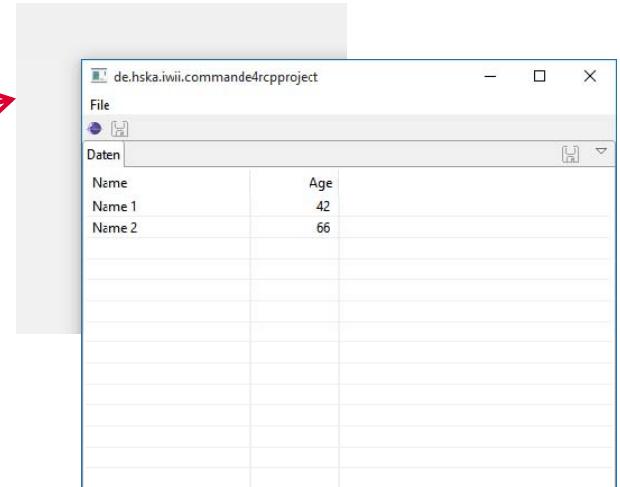
Dependency Injection

Annotationen zur Steuerung des Verhaltens



- Beispiel:

```
public class LifecycleManager {  
  
    private Shell shell;  
  
    @PostContextCreate  
    public void login() {  
        shell = new Shell(SWT.TOOL | SWT.NO_TRIM);  
        shell.setSize(300, 300);  
        // Login-Dialog bauen  
        shell.open();  
    }  
  
    @PreDestroy  
    public void preDestroy() {  
        shell.dispose();  
    }  
}
```



- Hier fehlt natürlich der eigentliche Inhalt des Dialogs.



- Es gibt mehrere Möglichkeiten, Objekte zur Injektion bereitzustellen:
 - ◆ Klasse mit **@Creatable** annotieren → kann direkt unter dem Klassennamen injiziert werden, Beispiel:

```
@Creatable  
public class Data {  
}
```

Verwendung durch:

```
@Inject  
private Data data;
```

- ◆ direkt durch das Modell erzeugt → kann normalerweise **nicht** unter dem Klassennamen injiziert werden, Beispiel:

```
public MyPart { // Stellt Part aus dem Anwendungsmodell dar  
}
```

MyPart wird im Kontext nicht unter dessen Klassennamen abgelegt. Wenn der Part aktiv ist, ist er unter **IServiceConstants.ACTIVE_PART** injizierbar.

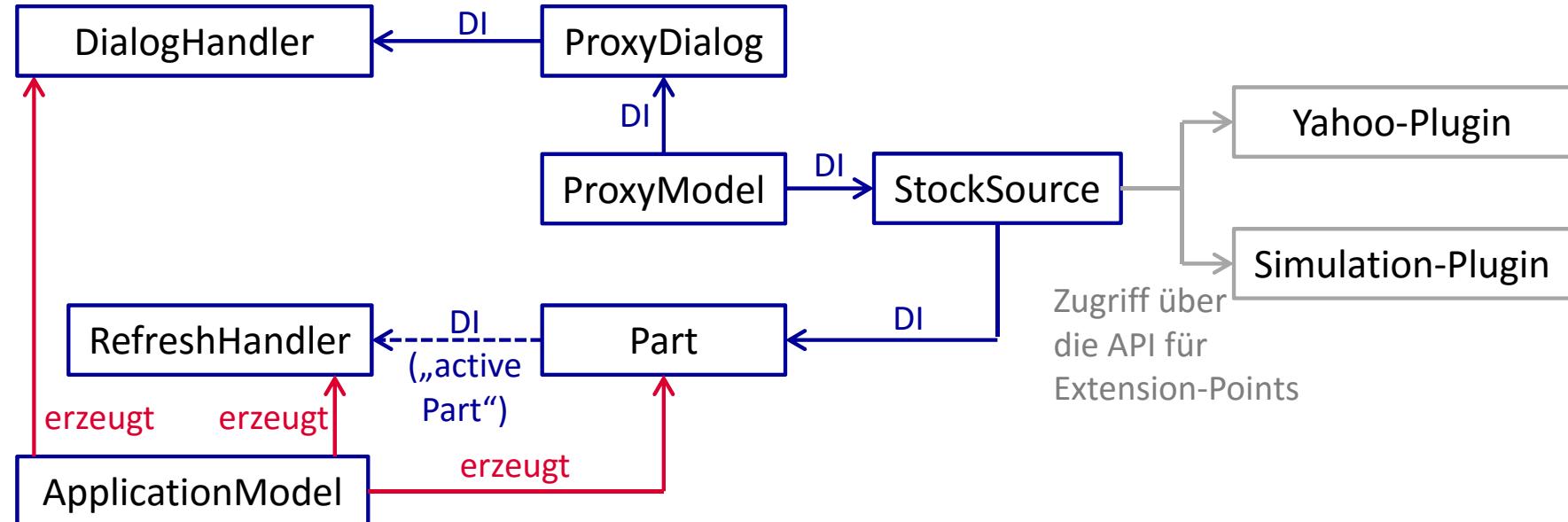
- ◆ manuell in den Kontext eingefügt → kann unter dem eingefügten Schlüsselnamen injiziert werden

Dependency Injection



Anmerkungen und Tipps

- Beispiel zur Bonusaufgabe (synchrone Abholen, Anbindung der vorgegebenen Plugins über Extension Points):

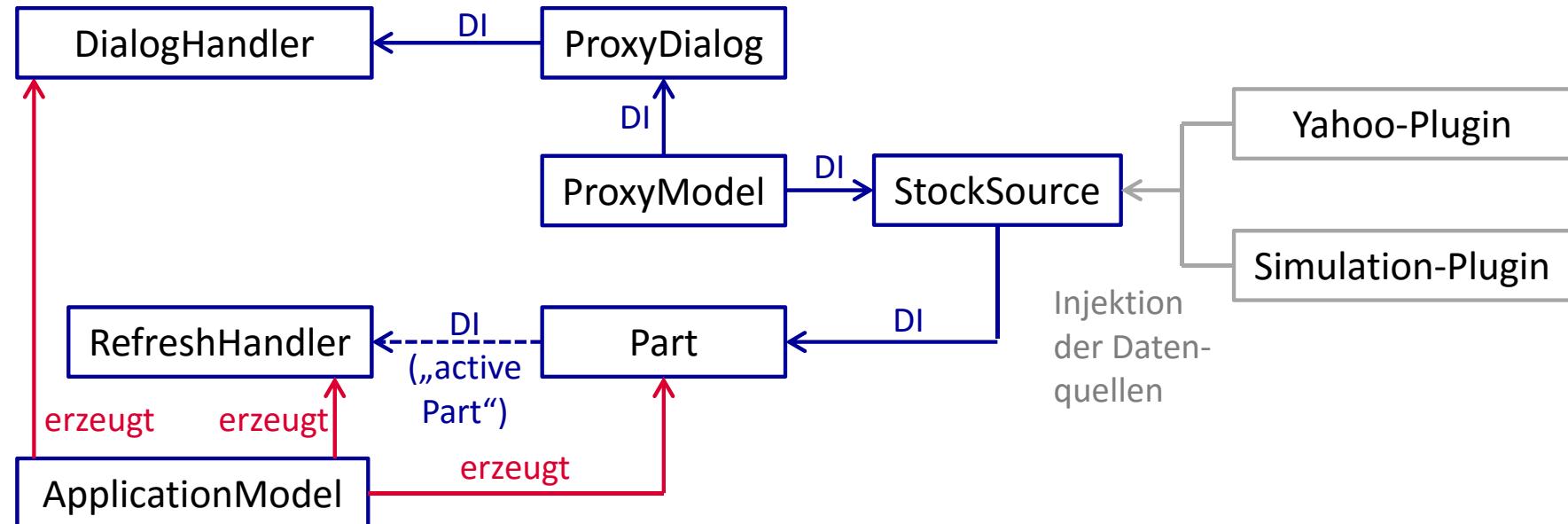


Dependency Injection



Anmerkungen und Tipps

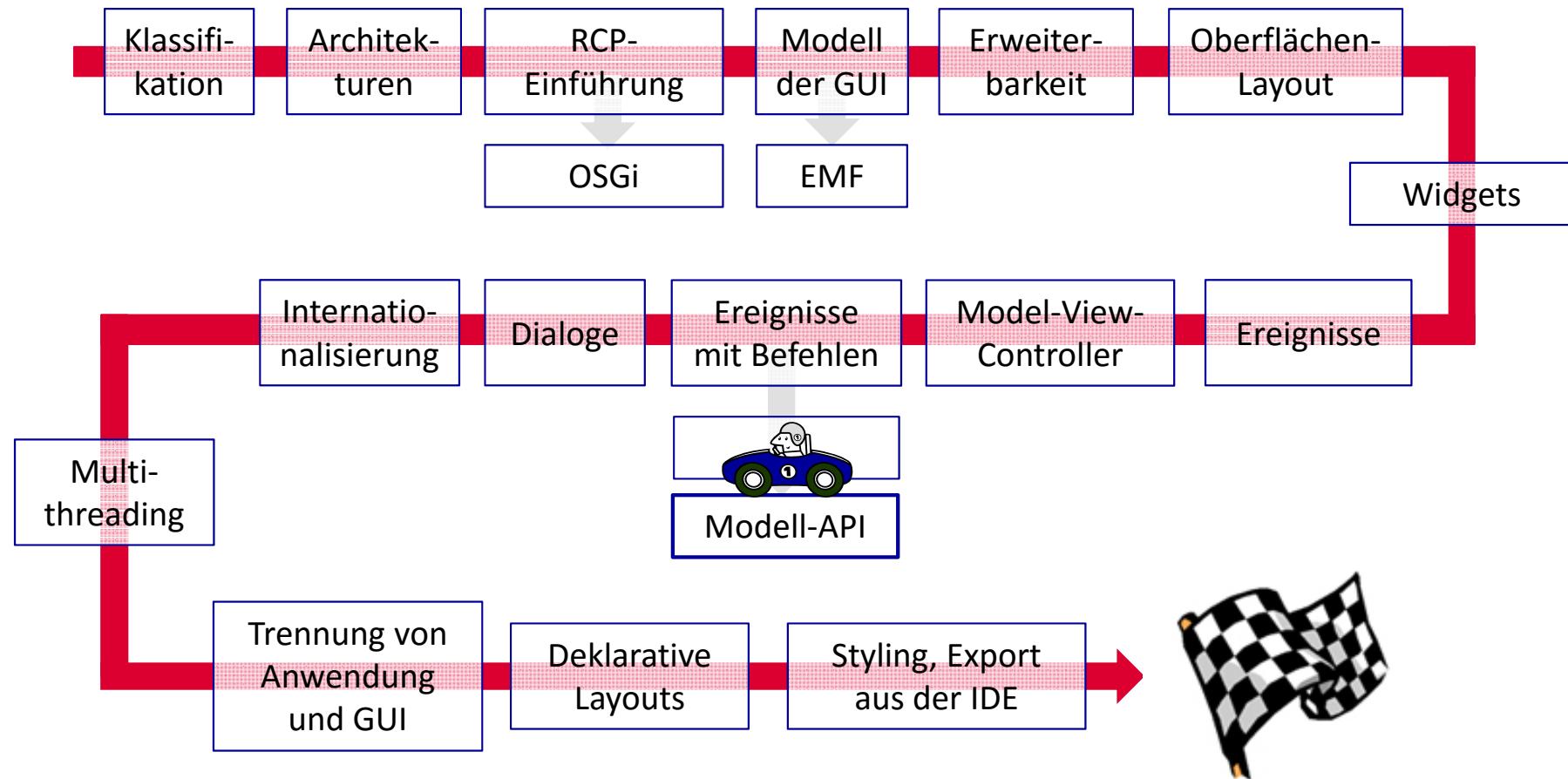
- Beispiel zur Bonusaufgabe (synchrone Abholen, Injektion der vorgegebenen Plugins):



- Weitere Informationen: http://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection

Dynamische Änderung des Anwendungsmodells

Einführung

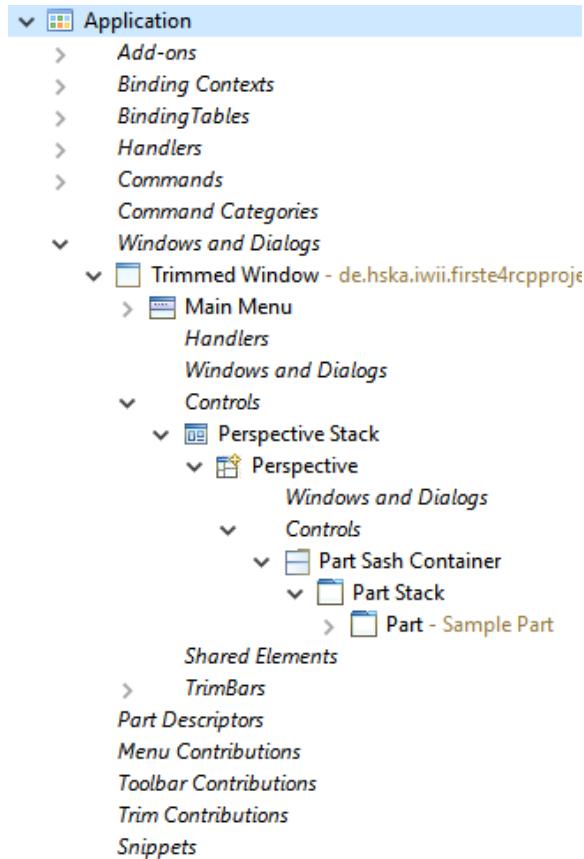


Dynamische Änderung des Anwendungsmodells

Einführung



- Rückblick auf das Anwendungs-Modell und dessen Klassen:



Annotation	Beschreibung
MApplication	das Modell der Anwendung
MWindow	Fenster einer Anwendung
MTrimmedWindow	Fenster mit Systemmenü
MPerspective	Perspektive
MPart	Part
MDirtyable	injizierbarer Teil des MPart , wird verwendet, um einen Part als „ungespeichert“ zu markieren
MPartDescriptor	Template für Parts
Snippets	vorkonfigurierte Modellelemente, die zur Laufzeit in das Modell kopiert werden können (z.B. dynamisch Perspektiven ergänzen)



- Das Anwendungs-Modell muss u.U. zur Laufzeit verändert oder ausgelesen werden:
 - ◆ Hinzufügen von Perspektiven, Parts, ...
 - ◆ Suchen von Menüeinträgen, um die Texte zu ändern
 - ◆ ...
- Dazu gibt es eine API, siehe auch http://wiki.eclipse.org/E4/UI/Modeled_UI
- Hinzufügen zu einem Modell: Die Klasse **EModelService** besitzt eine Fabrikmethode: **modelService.create(Class<?> modelElement)**
- Beispiel, um ein neues Fenster zu erzeugen:

```
public class LoginHandler {  
    @Execute  
    public void execute(MApplication application, EModelService modelService) {  
        MTrimmedWindow window = (MTrimmedWindow)  
            modelService.createModelElement(MTrimmedWindow.class);  
        window.setWidth(400);  
        window.setHeight(300);  
        application.getChildren().add(window); // Fenster zur Anwendung hinzufügen  
    }  
}
```

Dynamische Änderung des Anwendungsmodells

API



- Beispiel zu einem Editor, der das interne Modell-Flag auf „dirty“ setzt, wenn sich sein Zustand geändert hat:

```
public class Editor {  
    @Inject  
    private MDirtyable dirty; // true: Editorinhalt muss gespeichert werden  
  
    @Inject  
    public void create(Composite parent) {  
        // Textfeld einfügen und bei Textänderung den Editorinhalt als  
        // "dirty" markieren.  
        Text text = new Text(parent, SWT.LEFT | SWT.WRAP);  
        text.addModifyListener(new ModifyListener() {  
            public void modifyText(ModifyEvent event) {  
                dirty.setDirty(true);  
            }  
        });  
    }  
}
```



```
@Persist  
public void save() {  
    // Inhalt des Textfeldes speichern - fehlt hier  
  
    // Inhalt ist wieder gesichert:  
    dirty.setDirty(false);  
}  
}
```

- Eclipse ruft **save** nur dann auf, wenn im Modell **MDirtyable** für diesen Editor bzw. für eine Ansicht auf **true** gesetzt ist.
- Das Flag **MDirtyable** wird einfach per Dependency Injection eingetragen.
- → siehe auch Beispiel zum Windows Media Player

Dynamische Änderung des Anwendungsmodells

API

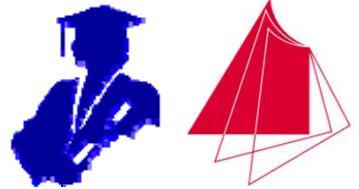


- Weitere Modelländerungen sind über die „Eclipse 4 Platform Services“ möglich.

Service	Beschreibung
EModelService	Modellelemente suchen „Snippets“ erzeugen und klonen, neue Elemente ins Modell einfügen
ESelectionService	aktuelle Selektion der Workbench auslesen oder ändern
ECommandService	lesender und schreibender Zugriff auf Kommandos, Ausführung von Kommandos
EHandlerService	lesender und schreibender Zugriff auf Handler, Ausführung von Handlern
EPartService	suche nach Parts, Wechseln der Perspektive, Ein- und Ausblenden von Parts
IEventBroker	Senden und Empfangen von Nachrichten → kommt noch
StatusReporter	Statusausgaben
EContextService	Aktivierung und Deaktivierung von Tastaturbindungen

Dynamische Änderung des Anwendungsmodells

API



Service	Beschreibung
IThemeEngine	Definition von Themes, Wechsel der Themes zur Laufzeit
Logger	einheitliche Logging-API
IShellProvider	Zugriff auf das aktuelle SWT-Fenster

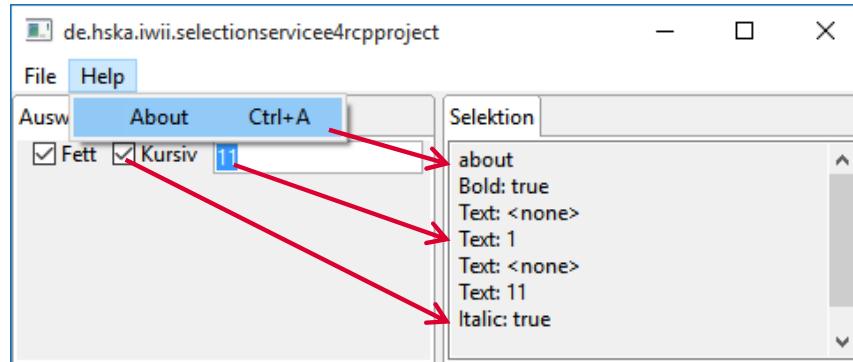
- Verwendung: Dienste per Dependency Injektion zur Verfügung stellen
- Viele Beispiele unter: <http://www.vogella.com/articles/Eclipse4Services/article.html>

Dynamische Änderung des Anwendungsmodells



API: Selektionsverwaltung

- Selektionsereignisse sind sehr wichtige Ereignisse:
 - ◆ Sie werden an unterschiedlichen Stellen ausgelöst.
 - ◆ Es gibt viele verschiedene Beobachter, die darauf warten.
 - ◆ Wie sollen diese untereinander „verdrahtet“ werden?
- Die Eclipse-RCP unterstützt diesen zentralen Beobachter-Mechanismus durch den **ESelectionService**.
- Das komplette Beispiel finden Sie im Projekt
de.hska.iwii.selectionservicee4rcpproject.





- Die Selektion ist hier ein String. Ändert sich die Selektion, dann wird rechts im Textfeld die neue Selektion ausgegeben.
- Vorgehensweise:
 - ◆ Die Quellen der Selektionsänderungen lassen sich einen **ESelectionService** injizieren und melden darüber die aktuelle Selektion.
 - ◆ Die Empfänger (Beobachter) der Selektionsänderungen implementieren eine Empfängermethode, die den **ESelectionService** automatisch übergeben bekommt.

Dynamische Änderung des Anwendungsmodells



API: Selektionsverwaltung

- Sender (Tasten „Bold“, „Italic“ und das Textfeld) in einem Part:

```
public class InputPart {  
    // Injektion des ESelectionProviders  
    private @Inject ESelectionService selectionService;  
    private StyledText text;  
  
    @Inject  
    public void createPartControl(Composite parent) {  
        // Aufbau der Oberfläche, hier verkürzt  
        text = new StyledText(parent, SWT.BORDER);  
        // Selektion melden  
        text.addSelectionListener(new SelectionAdapter() {  
            @Override  
            public void widgetSelected(SelectionEvent event) {  
                if (event.x != event.y) {  
                    selectionService.setSelection("Text: " +  
                        text.getText().substring(event.x, event.y));  
                }  
                else {  
                    selectionService.setSelection("Text: <none>");  
                }  
            }  
        });  
    }  
}
```

Dynamische Änderung des Anwendungsmodells



API: Selektionsverwaltung

- Sender (Handler „About“), Achtung: **ESelectionService** nicht als Attribut injizieren (falscher Service):

```
public class AboutHandler {  
    @Execute  
    public void execute(ESelectionService selectionService) {  
        selectionService.setSelection("about");  
    }  
}
```

- Empfänger in einem Part (beim Erzeugen des Objektes gibt es keine Selektion):

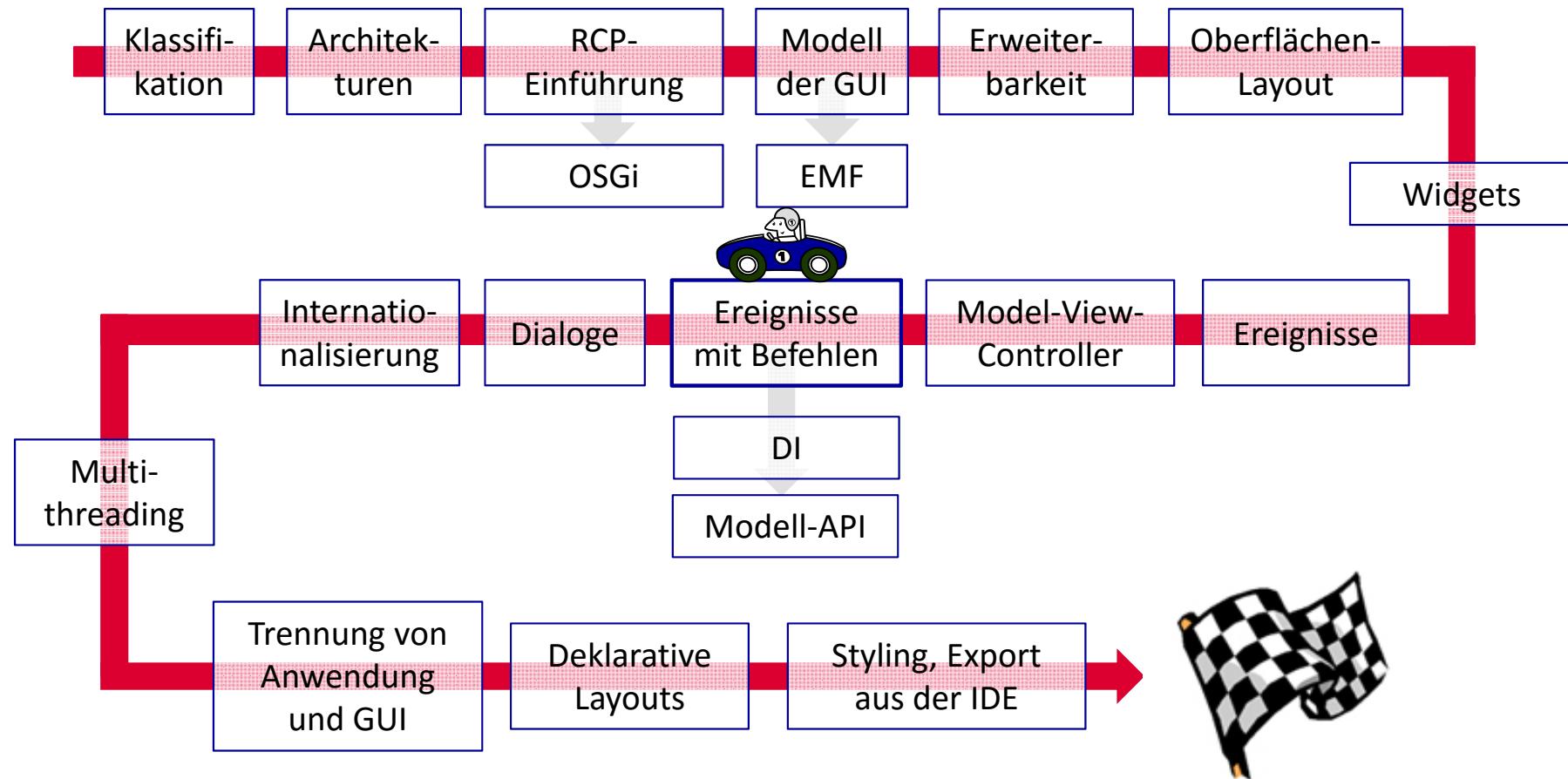
```
public class LogPart {  
    // Textausgabefeld  
    private Text logField;  
    // Aufbau der Oberfläche weggelassen  
    @Inject  
    public void selected(@Optional @Named(IServiceConstants.ACTIVE_SELECTION)  
                         String selection) {  
        if (selection != null) {  
            logField.append(selection + "\n");  
        }  
    }  
}
```



- Die Selektion kann ein Objekt einer beliebigen Klasse sein.
- Die Empfänger werden nur aufgerufen, wenn die Klasse der Selektion zum Methodenparameter passt.
- Die Selektionsverwaltung ist für „normale“ Tasten (Typ **Push**) nicht sonderlich gut geeignet, weil sich die Selektion beim wiederholten Klick auf dieselbe Taste nicht ändert.

Ereignisbehandlung durch Befehle

Bedingte Handler und Menüeinträge





- Wiederholung: Aktivierung und Deaktivierung von Kommandos:
 - ◆ Direkt durch den Handler:
 - Handler können durch implementieren einer mit **@CanExecute**-Methode selbst festlegen, ob sie aktiv sind oder nicht.
 - Ist kein Handler für ein Kommando aktiv, dann sind alle Menü- und Toolbar-Einträge für das Kommando deaktiviert.
 - Nachteil: Das Plug-in, aus dem der Handler stammt, muss gestartet werden.
 - Was passiert, wenn mehrere Handler ausführbar sind? Dann wird der mit dem „speziellsten“ Kontext ermittelt (von allgemein bis speziell sortiert, Auswahl):
 - aktive Shell
 - aktives Workbench-Fenster
 - aktiver Part
 - aktuelle Selektion



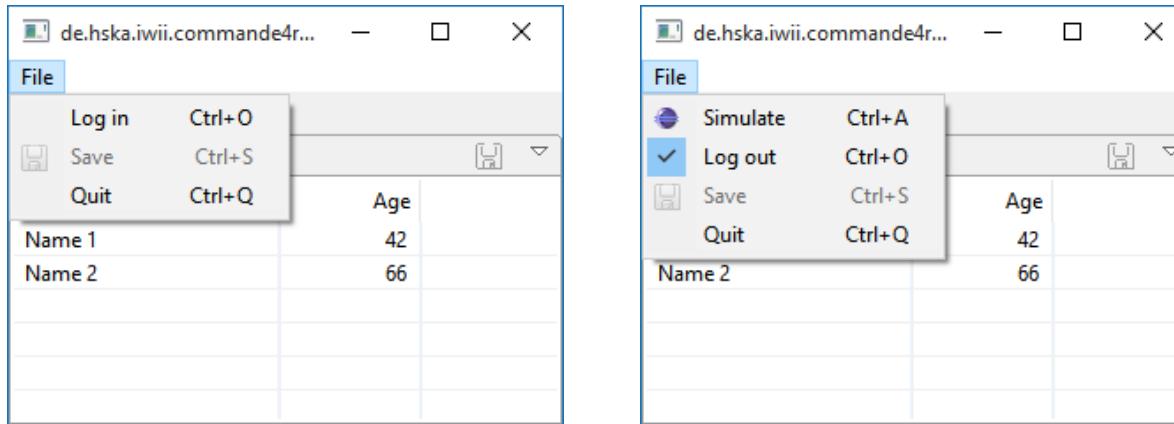
- ◆ Durch eine deklarative Festlegung von Bedingungen:
 - Menü- und Toolbar-Einträge lassen sich mit Bedingungen versehen.
 - Die Bedingungen können großenteils ohne Start des zugehörigen Plug-ins ausgewertet werden.
 - Vorteil: Es müssen nur die Plug-ins gestartet werden, deren Funktionalität genutzt wird. Zum Aufbau der Oberfläche ist das meist nicht erforderlich.
 - Die Bedingung kann zur Zeit nur die Sichtbarkeit des Eintrags steuern.

Ereignisbehandlung durch Befehle



Bedingte Handler und Menüeinträge

- Das folgende Beispiel legt die Boole'sche Variable **de.hska.iwii.loggedIn** im Kontext ab. Ist diese **true**, dann soll der Menüeintrag „Simulate“ sichtbar sein, ansonsten nicht.



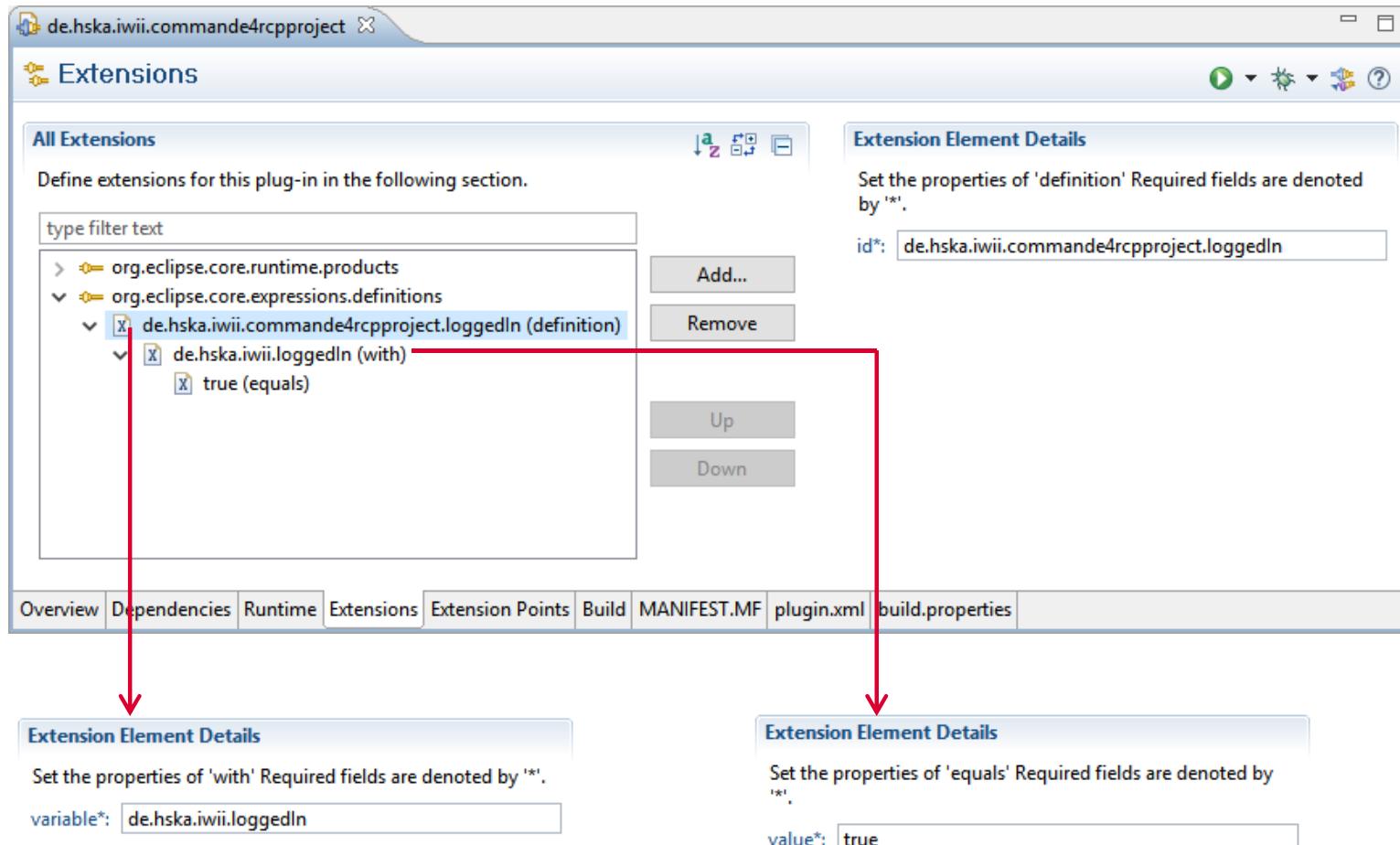
- Vorgehen: Definition der Variablenbedingung und Registrierung am Erweiterungspunkt **org.eclipse.core.expressionsdefinitions**

Ereignisbehandlung durch Befehle

Bedingte Handler und Menüeinträge



- Definition und Registrierung (der Wert der Variablen `de.hska.iwii.loggedIn` muss **true** sein)



Ereignisbehandlung durch Befehle

Bedingte Handler und Menüeinträge



- Im Anwendungs-Modell wird festgelegt, dass die Sichtbarkeit des Simulations-Eintrags im Menü von der Bedingung abhängt:

The screenshot illustrates the configuration of a menu item 'Simulate' in a plugin's configuration interface. The top-left window shows the 'Windows and Dialogs' tree, with the 'Main Menu' expanded. The 'File' menu item is selected, and its submenu 'File' is also expanded. The 'Simulate' item is highlighted with a red box. Below this, a smaller window shows the 'Core Expression' configuration for this menu item. The 'Visible-When Expression' field contains the value 'CoreExpression'. A red arrow points from the 'Simulate' menu item in the tree to this expression field. Another red arrow points from the 'Core Expression' window down to a separate 'Core Expression' configuration window at the bottom, which has an 'Expression ID' of 'de.hska.iwii.commande4rcpproject.loggedIn'.

Ereignisbehandlung durch Befehle



Bedingte Handler und Menüeinträge

- Der Handler **Loginhandler** simuliert einen Login-Vorgang. Bei jedem Klick auf den Menüeintrag wird der Login-Zustand umgeschaltet (an- bzw. abgemeldet).

```
public class LoginHandler {  
    public static final String LOGGED_IN_VARIABLE = "de.hska.iwii.loggedIn";  
  
    /**  
     * Schaltet den Login-Zustand um.  
     * @param context Eclipse-Kontext, in dem alle Objekte für das  
     * Dependency Injection gespeichert wird.  
     */  
    @Execute  
    public void execute(IEclipseContext context) {  
        Boolean loggedIn = (Boolean) context.get(LOGGED_IN_VARIABLE);  
        loggedIn = loggedIn == null || loggedIn == false ? true : false;  
        context.set(LOGGED_IN_VARIABLE, loggedIn);  
    }  
}
```

Extension Element Details

Set the properties of 'with' Required fields are denoted by '*'.

variable*: de.hska.iwii.loggedIn

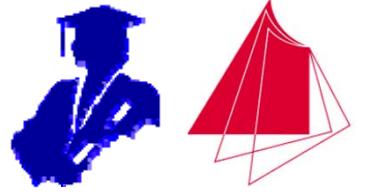


- Der Handler kann auch den Text des „eigenen“ Menü-Eintrags ändern (hat aber nichts mehr mit Bedingungen zu tun):

```
public class LoginHandler {  
    private boolean loggedIn = false;  
    public static final String LOGGED_IN_VARIABLE = "de.hska.iwii.loggedIn";  
    /**  
     * Schaltet den Login-Zustand um.  
     * @param context Eclipse-Kontext, in dem alle Objekte für das  
     * Dependency Injection gespeichert wird.  
     * @param application Modellobjekt für die komplette Anwendung  
     * @param modelService Klasse zur Manipulationen zur Lesen des  
     * Anwendungsmodells  
     */  
    @Execute  
    public void execute(IEclipseContext context, MApplication application,  
                       EModelService modelService) {  
        Boolean loggedIn = (Boolean) context.get(LOGGED_IN_VARIABLE);  
        loggedIn = loggedIn == null || loggedIn == false ? true : false;  
        context.set(LOGGED_IN_VARIABLE, loggedIn);  
        // Referenz auf das Modellobjekt für das Fenster suchen  
        List<MWindow> windows = modelService.findElements(application, null,  
                                                          MWindow.class, null);
```

Ereignisbehandlung durch Befehle

Bedingte Handler und Menüeinträge



```
// Menü-Eintrag mit der angegebenen ID ändern
changeMenuItem(windows.get(0).getMainMenu(), modelService,
    "de.hska.iwii.commande4rcpproject.handledmenuitem.login", loggedIn);
}
/*
 * Text des Menü-Eintrags je nach Anmeldestatus ändern.
 * @param root Startelement im Modell, ab dem gesucht werden soll.
 * @param modelService Klasse zur Manipulationen zur Lesen des
 * Anwendungsmodells
 * @param id ID des gesuchten Menü-Eintrags
 */
private void changeMenuItem(MUIElement root, EModelService modelService,
    String id, boolean loggedIn){
    // Alle Menü-Einträge mit der angegebenen ID suchen (sollte nur einer
    // sein)
    List<MHandledMenuItem> foundElements = service.findElements(root, id,
        MHandledMenuItem.class, null);
    // Eintrag selektieren oder deselektieren (je nach Login-Zustand)
    foundElements.get(0).setSelected(loggedIn);
    // Eintragstext je nach Login-Zustand anpassen
    foundElements.get(0).setLabel(loggedIn ? "Log out" : "Log in");
}
```



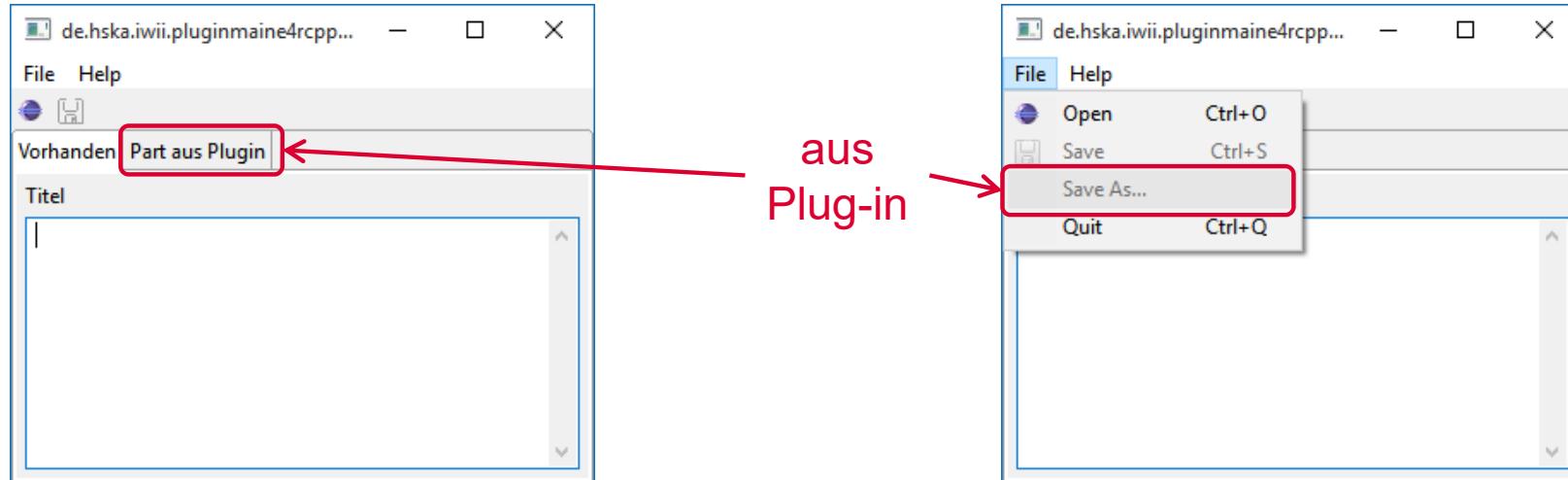
- Es sind auch komplexere Anfragen möglich:
 - ◆ Anzahl Elemente in einer Datenstruktur
 - ◆ Selektion enthält Objekte bestimmter Klassen
 - ◆ ...
- Es gibt eine ganze Anzahl vordefinierter Variablen im Kontext, siehe Schnittstelle **IServiceConstants** (z.B. die bereits bekannte **activePart**)
- Nähere Informationen: http://wiki.eclipse.org/Command_Core_Expressions

Ereignisbehandlung durch Befehle



Beiträge durch Plug-ins

- Wie können jetzt Plug-ins eigene Menüs, Menüeinträge, Toolbar-Einträge, Parts, Kommandos, Handler, usw. bereitstellen?
- Das Anwendungs-Modell kann ja nicht verändert werden.
- Dazu dienen Modell-Fragmente innerhalb von Plug-ins. Diese erweitern das eigentliche Anwendungs-Modell.
- Beispiel (**de.hska.iwii.mainplugin4rcpproject** und Plug-in **de.hska.iwii.plugin4rcpproject**):

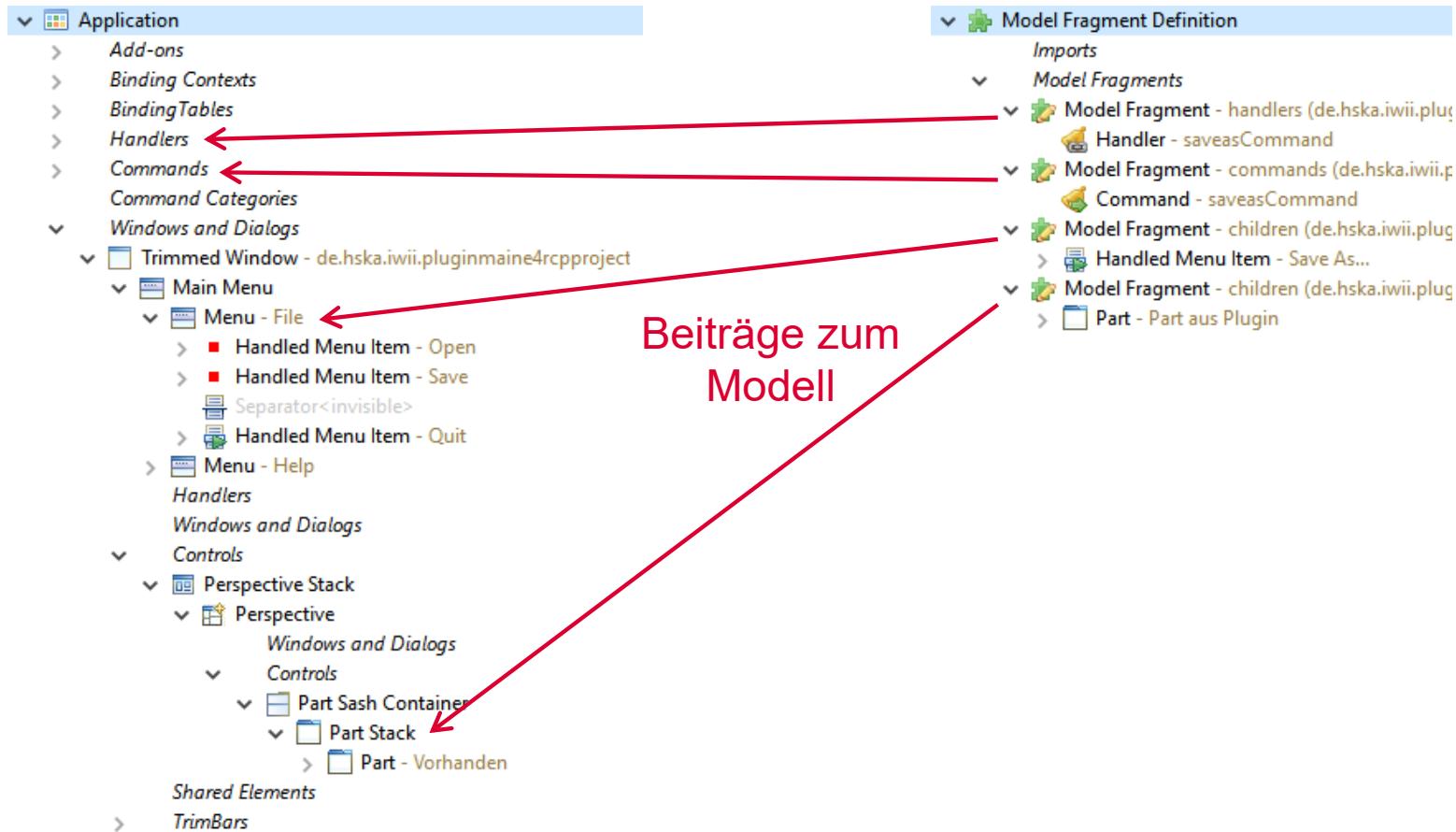


Ereignisbehandlung durch Befehle

Beiträge durch Plug-ins



- Die „Hauptanwendung“ besitzt das Anwendungs-Modell, das Plug-in ein Fragment zur Erweiterung des Modells:





- Jedes Modell-Fragment bezieht sich auf einen Teil des Anwendungs-Modells. Es besitzt:
 - ◆ einen Typ, der die Art der Erweiterung beschreibt (z.B. Kommando),
 - ◆ einen Verweis auf die Vater-ID des Elements im Anwendungs-Modell, zu dem das Fragment hinzugefügt werden soll
 - ◆ und eventuell die Position, an der es eingefügt werden soll (z.B. nach Menüeintrag mit der ID **xyz**).
- Die folgenden Seiten stellen Modell-Fragmente anhand von Menüeinträgen mit eigenem Kommando und Handler sowie von Parts vor. Weitere Fragmente funktionieren ähnlich.

Ereignisbehandlung durch Befehle

Beiträge durch Plug-ins



- Es soll ein Part als letztes Kind im PartStack eingefügt werden. Anwendungs-Modell:

The screenshot shows the Eclipse RCP application structure on the left and a configuration dialog on the right.

Application Structure (Left):

- Application
 - Add-ons
 - Binding Contexts
 - BindingTables
 - Handlers
 - Commands
 - Command Categories
 - Windows and Dialogs
 - Trimmed Window - de.hska.iwii.pluginmaine4rcpproject
 - Main Menu
 - File
 - Handled Menu Item - Open
 - Handled Menu Item - Save
 - Separator<invisible>
 - Handled Menu Item - Quit
 - Help
 - Handlers
 - Controls
 - Perspective Stack
 - Perspective
 - Windows and Dialogs
 - Controls
 - Part Sash Container
 - Part Stack
 - Part - Vorhanden
 - Shared Elements
 - TrimBars

Ereignisbehandlung durch Befehle

Beiträge durch Plug-ins



Das Fragment beschreibt die Einfügeposition im Modell:

The screenshot shows the 'Model Fragment Definition' interface on the left and the 'Model Fragment' configuration dialog on the right. The 'Model Fragment Definition' tree includes nodes like 'Model Fragment - handlers', 'Model Fragment - commands', 'Model Fragment - children', and 'Model Fragment - children'. A red arrow points from the 'Model Fragment - children' node to the 'Model Fragment' dialog. The 'Model Fragment' dialog has fields for 'Extended Element ID' (set to 'de.hska.iwii.pluginmaine4rcpproject.mainpartstack'), 'Feature Name' (set to 'children'), and 'Position in list' (set to 'after:de.hska.iwii.pluginmaine4rcpproject.mainPart'). These three fields are also highlighted with a red box.

- Fragment-Eigenschaften:
 - ◆ **Element Id**: ID im Modell, der das Fragment zugeordnet wird
 - ◆ **Featurename**: Welcher Eigenschaft im Modellelement soll das Fragment zugeordnet werden (hier seinen Kindern)?
 - ◆ **Position in list**: Wo innerhalb der Kinder wird das Fragment platziert?

The screenshot shows the 'Part' configuration dialog on the right. It contains fields for 'ID' (set to 'de.hska.iwii.pluginmaine4rcpproject.parts.newPart'), 'Label' (set to 'Part aus Plugin'), 'Accessibility Phrase', 'Tooltip', 'Icon URI', 'Class URI' (set to 'bundleclass://de.hska.iwii.pluginmaine4rcpproject/de'), 'Container Data', 'ToolBar' (unchecked), 'Closeable' (unchecked), 'To Be Rendered' (checked), and 'Visible' (checked). A red arrow points from the 'Part' dialog back to the 'Model Fragment Definition' interface.



- Hinweis zur **Position in list**: Erlaubt sind folgende Angaben:

Angabe	Beschreibung
first	Das Fragment wird vorne in die Liste eingefügt.
index: pos	Das Fragment wird an der angegebenen Position pos die Liste eingefügt.
before: id	Das Fragment wird vor dem Element mit der ID id in die Liste eingefügt.
after: id	Das Fragment wird nach dem Element mit der ID id in die Liste eingefügt.

- Viele Angaben benötigen also die IDs von Modellelementen → Modellelemente sollten immer IDs besitzen!

Ereignisbehandlung durch Befehle



Beiträge durch Plug-ins

- Handler, zugeordnet werden die eigenen Handler wie im Anwendungsmodell

The screenshot shows the 'Model Fragment' editor with the 'handlers' feature selected. The 'Extended Element ID' field contains 'de.hksa.iwii.pluginmaine4rcpproject.application'. The 'Feature Name' field contains 'handlers'. A red arrow points from the text 'fester Name' to the 'Feature Name' field. Another red arrow points from the text 'ID der Anwendung im Anwendungsmodell' to the 'Extended Element ID' field. Below the feature configuration, a list of handlers is shown, including 'Handler - saveasCommand'.

- Kommandos, zugeordnet werden die eigenen Kommandos wie im Anwendungsmodell

The screenshot shows the 'Model Fragment' editor with the 'commands' feature selected. The 'Extended Element ID' field contains 'de.hksa.iwii.pluginmaine4rcpproject.application'. The 'Feature Name' field contains 'commands'. A red arrow points from the text 'fester Name' to the 'Feature Name' field. Another red arrow points from the text 'ID der Anwendung im Anwendungsmodell' to the 'Extended Element ID' field. Below the feature configuration, a list of commands is shown, including 'Command - saveasCommand'.

Ereignisbehandlung durch Befehle

Beiträge durch Plug-ins



■ Menüeintrag:

Model Fragment

Extended Element ID: de.hska.iwii.pluginmaine4rcpproject.file

Feature Name: children

Position in list: before:de.hska.iwii.pluginmaine4rcpproject.filomenuseparator

Addon Add Remove Down

Handled Menu Item - Save As...

Handled Menu Item

ID: de.hska.iwii.pluginmaine4rcpproject.menu.saveas

Type: Push

Label: Save As...

Mnemonics:

Tooltip:

Icon URI:

Enabled:

Selected:

Visible-When Expression: <None>

Command: saveasCommand - de.hska.iwii.pluginmaine4rcpproject

To Be Rendered:

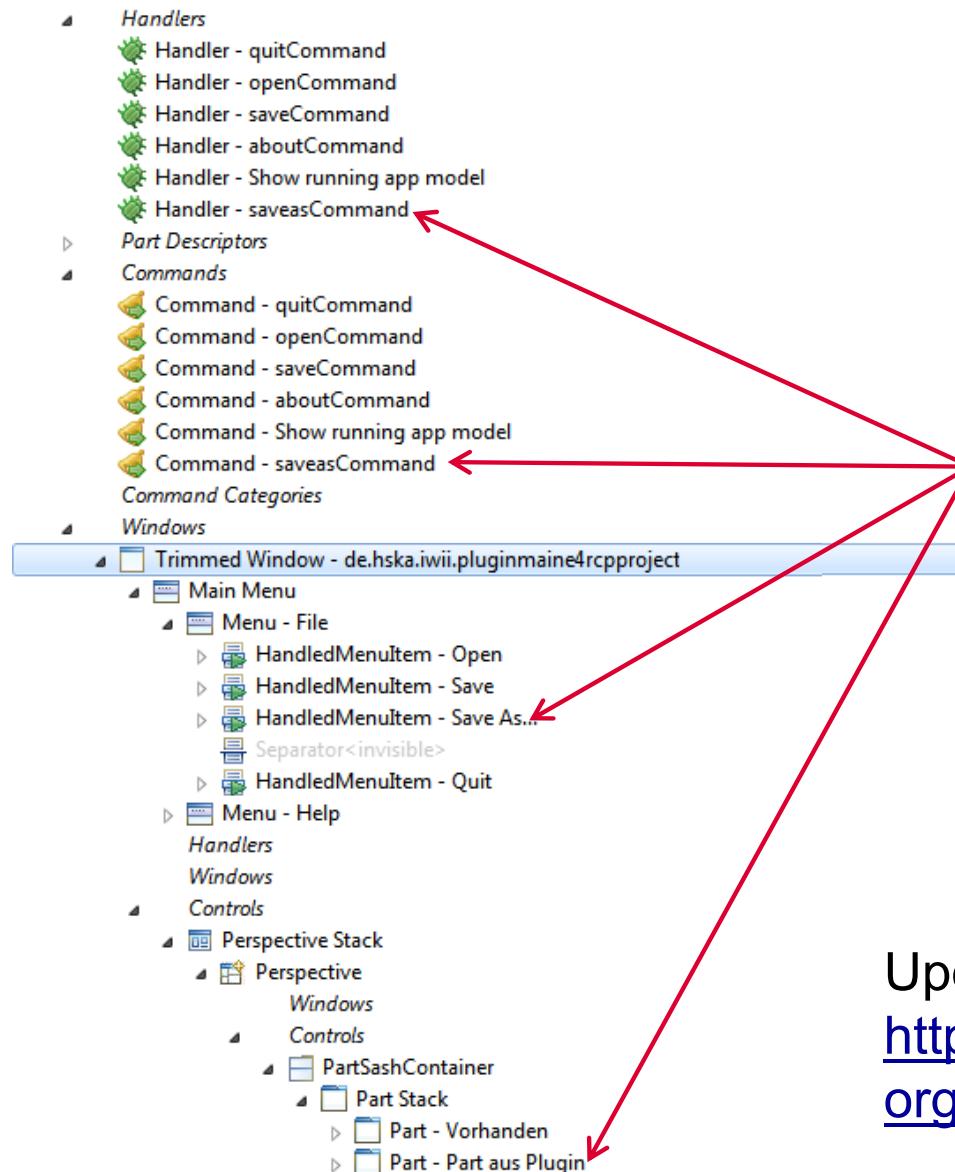
Visible:

ID des Menüs

Position: vor dem (unsichtbaren) Trennstrich

Ereignisbehandlung durch Befehle

Beiträge durch Plug-ins



Lifeeditor: Die Beiträge des Plugins sind beim Start der Anwendung in das Modell integriert worden.

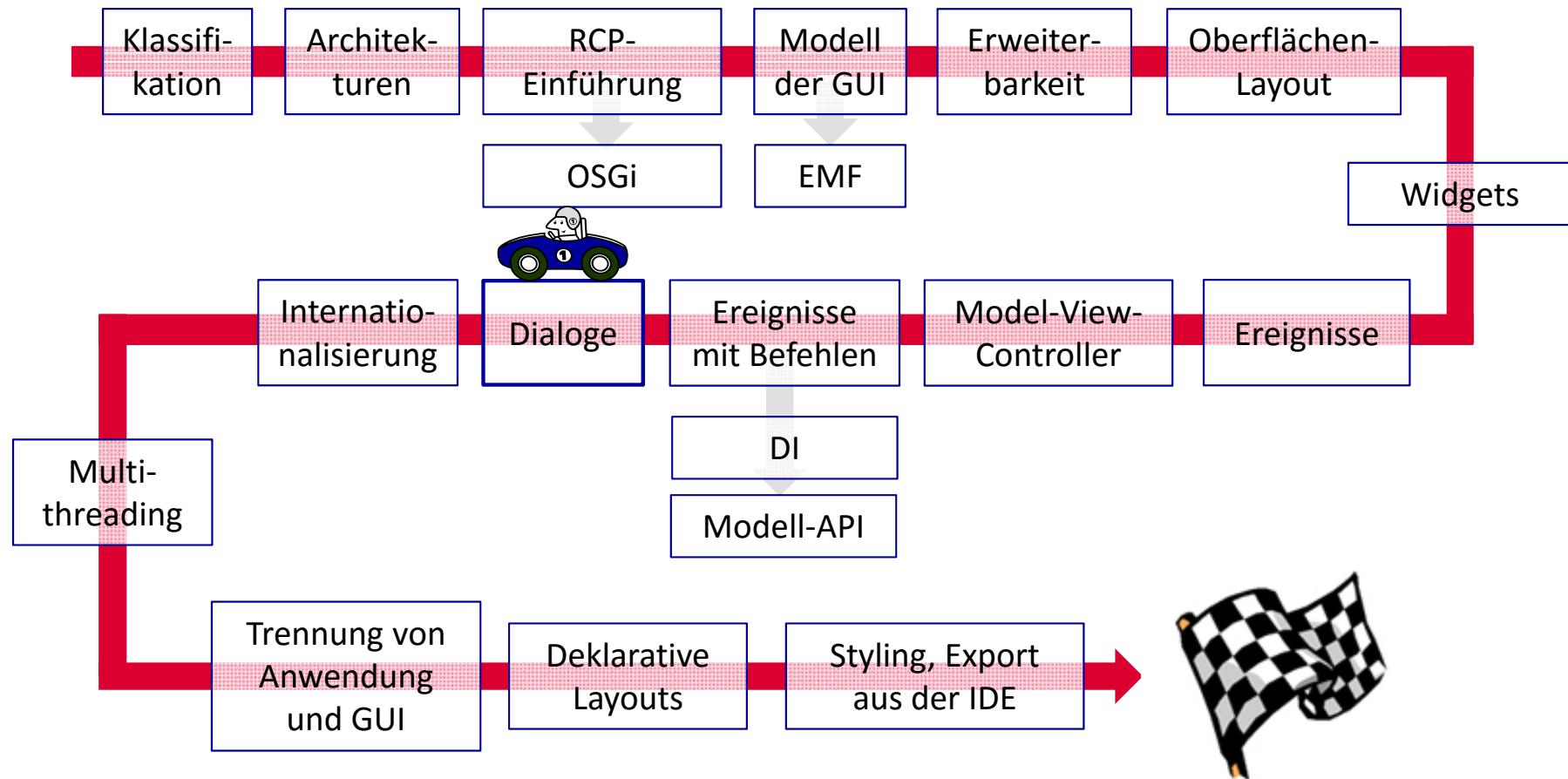
Update Site:
[http://download.eclipse.org/e4/snapshots/
org.eclipse.e4.tools/](http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools/)



- Erstellung eines Plug-ins mit Oberflächenbeiträgen:
 - ◆ **File → New → Other → Plug-in Development → Plug-in Project**
 - ◆ Projektnamen eintragen, Verzeichnis wählen
 - ◆ Activator je nach Projekt aus- oder abwählen
 - ◆ auf der kommenden Seite **Create a plug-in using one of the templates** abwählen
 - ◆ Mit **File → New → Other → Eclipse 4 → Model → New Model Fragment** hinzufügen
- Neben der Verwendung von Modell-Fragmenten gibt es auch noch „Processors“, mit denen die Erweiterungen programmatisch erfolgen → soll hier nicht besprochen werden.

Dialoge

Einführung





- Einfache Dialoge müssen nicht mühsam über Layout-Manager gebaut werden.
- Weiterer Vorteil: Es werden Betriebssystem-spezifische Eigenheiten (z.B. Reihenfolge der Tasten) automatisch berücksichtigt.
- Für bestimmte Aufgaben lassen sich vordefinierte und konfigurierbare Dialoge einsetzen:
 - ◆ SWT:
 - **MessageBox**: Ausgabe einfacher textueller Nachrichten
 - **ColorDialog**: Farbauswahl
 - **DirectoryDialog**: Verzeichnisauswahl
 - **FileDialog**: Einfach- oder Mehrfachauswahl von Dateien zum Laden oder Speichern
 - **FontDialog**: Zeichensatzauswahl
 - **Dialog**: Bau eigener einfacher Dialoge → besser **Dialog** von JFace verwenden



- ◆ JFace:
 - **ErrorDialog**: Darstellung mehrerer Fehlermeldungen (einschließlich Ausnahmen möglich)
 - **InputDialog**: Dialog zur Eingabe eines einzeiligen Textes
 - **MessageDialog**: ähnlich der **MessageBox** aus SWT, etwas weniger Code erforderlich
 - **ProgressMonitorDialog**: Führt lang-laufende Operationen (optional in einem eigenen Thread) aus und stellt einen Fortschrittsbalken dar
 - **TitleAreaDialog**: Dialog mit vordefiniertem Aussehen einer Titelzeile
 - **IconAndMessageDialog**: Basisklasse für verschiedene Dialoge
 - **Wizard**: Dialog zur Führung des Benutzers in einer vorgegebenen Reihenfolge.
 - **PreferenceDialog**: Baumartiger Dialog aus mehreren Seiten zur Eingabe von Einstellungen für das Programm.



- ◆ RCP:
 - Wizards
 - Dialogfenster für automatische Updates



Dialoge

MessageBox

- Generelle Verwendung der SWT-Dialoge:
 - ◆ Konstruktor aufrufen, dabei Vaterfenster und Stile übergeben
 - ◆ Daten eintragen (Titelzeile, Meldungen, vorgegebene Farbe, ...).
 - ◆ Aufruf der **open**-Methode des Dialogs. Die Methode gibt die ID der gedrückten Taste als Ergebnis zurück. Der Dialog wird dabei geschlossen.
 - ◆ Reaktion auf die zurück gegebene ID.
- Beispiel anhand der **MessageBox**-Klasse:





Dialoge

MessageBox

- Quelltext:

```
MessageBox message = new MessageBox(shell,  
                                  SWT.YES | SWT.NO | SWT.ICON_QUESTION);  
message.setText("L\u00fchsen?");  
message.setMessage("Sollen die Dateien wirklich gel\u00fchsen werden?");  
System.out.println("Ergebnis: " + (message.open() == SWT.YES));
```

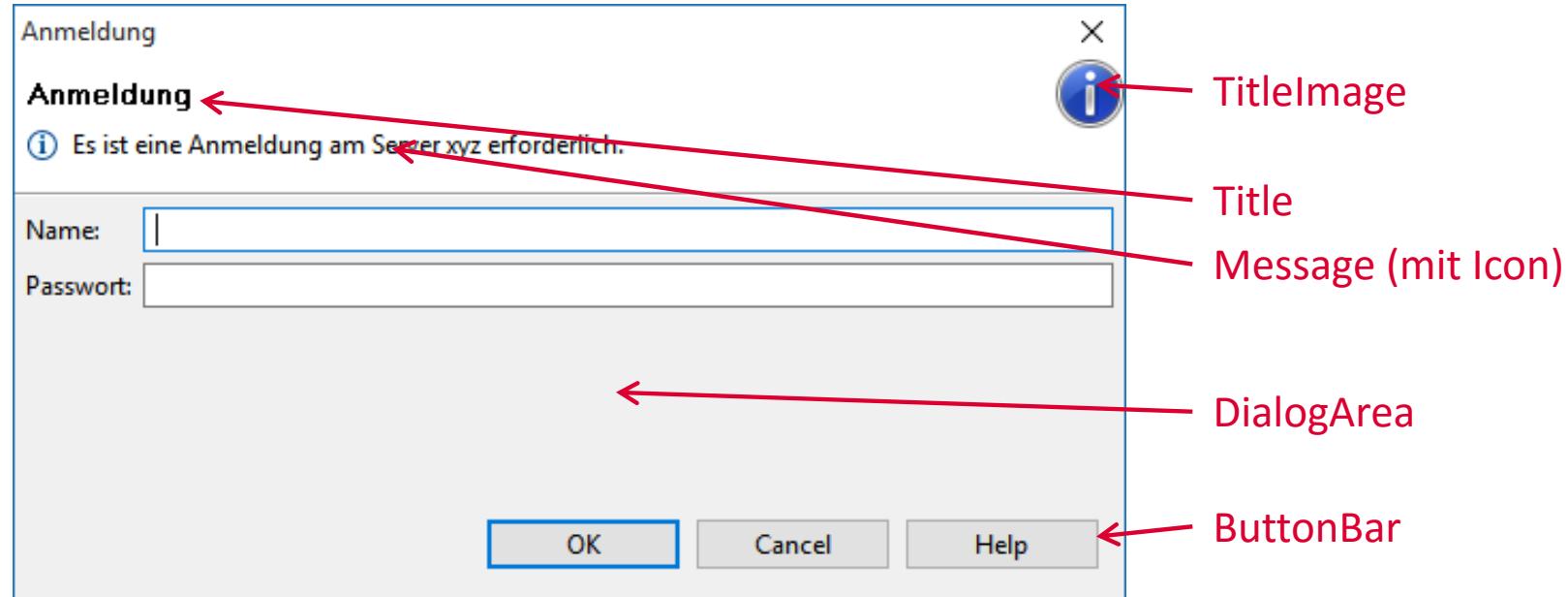
- Die anderen Dialogtypen funktionieren \u00e4hnlich.



Dialoge

TitleAreaDialog

- Verwendung des **TitleAreaDialog**:



- Achtung: **setTitleImage** muss vor dem Setzen des Titels und der Meldung aufgerufen werden.



- Es müssen verschiedene Methode überschrieben werden:
 - ◆ **createContents**: Hier wird am besten der Titelbereich gefüllt.
 - ◆ **createDialogArea**: In dieser Methode wird der eigentliche Inhalt des Dialogs erzeugt.
 - ◆ **createButtonsForButtonBar**: Hier wird i.d.R. die Methode **createButton** für jede zu erzeugende Taste aufgerufen. Die Reihenfolge der Tasten in der Darstellung hängt vom Betriebssystem ab. Jede Taste erhält eine eindeutige ID (durch Konstanten vordefiniert).
 - ◆ **createButton**: Die Standardimplementierung der Methode ist normalerweise ausreichend. Die Methode kann überschrieben werden, um selbst die Kontrolle über die Tastenerzeugung zu erhalten.
- Die Ereignisbehandlung erfolgt durch Überschreiben der Methode
 - ◆ **buttonPressed**: Der Methode wird die ID der gedrückten Taste übergeben. Die Standardimplementierung ruft im Falle der OK-Taste die Methode **okPressed** und im Falle der Cancel-Taste **cancelPressed** auf. In diesen beiden Fällen wird der Dialog automatisch geschlossen.



Dialoge

TitleAreaDialog

- Beispielhafte Verwendung:

```
public class TitleTextAreaJFaceLoginDialog extends TitleAreaDialog {  
    public TitleTextAreaJFaceLoginDialog(Shell shell) {  
        super(shell);  
    }  
  
    // Titelbereich anlegen  
    @Override  
    protected Control createContents(Composite parent) {  
        Control contents = super.createContents(parent);  
        setTitleImage(getShell().getDisplay()  
            .getSystemImage(SWT.ICON_WORKING));  
        setMessage("Es ist eine Anmeldung am Server xyz erforderlich.",  
            IMessageProvider.INFORMATION);  
        setTitle("Anmeldung");  
        getShell().setText("Anmeldung");  
        return contents;  
    }  
}
```



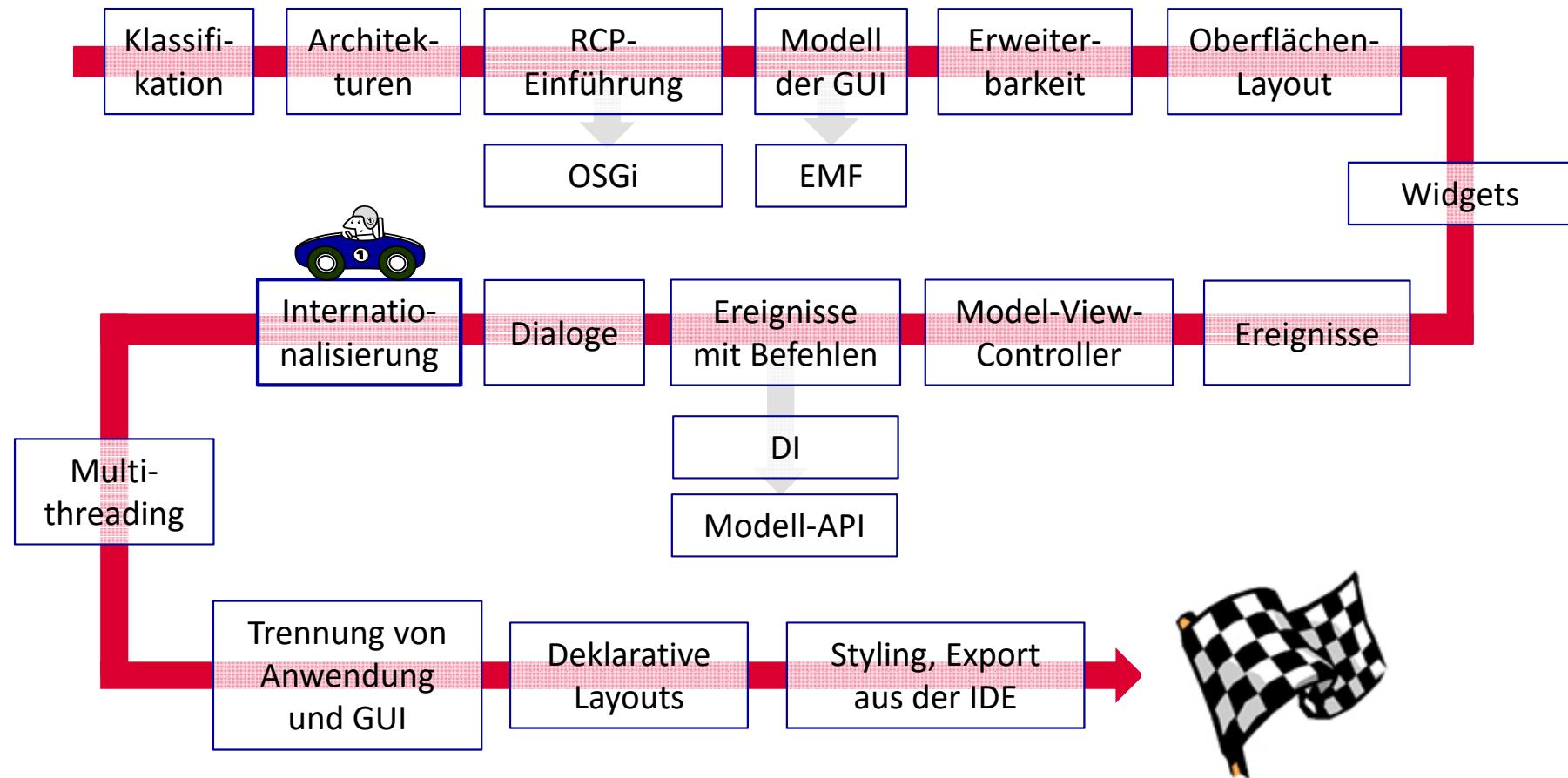
Dialoge

TitleAreaDialog

```
// Alle Tasten erzeugen.  
protected void createButtonsForButtonBar(Composite parent) {  
    createButton(parent, IDialogConstants.OK_ID,  
                IDialogConstants.OK_LABEL, true);  
    createButton(parent, IDialogConstants.CANCEL_ID,  
                IDialogConstants.CANCEL_LABEL, false);  
    createButton(parent, IDialogConstants.HELP_ID,  
                IDialogConstants.HELP_LABEL, false);  
}  
  
protected Control createDialogArea(Composite parent) {  
    Composite contents = (Composite) super.createDialogArea(parent);  
    // Inhalt erzeugen  
    return contents;  
}  
  
// Ereignisbehandlung.  
protected void buttonPressed(int buttonId) {  
    // ...  
    close();  
}
```

Internationalisierung

Einführung





- Häufig werden Anwendungen in vielen Ländern eingesetzt → Die Benutzungsoberflächen muss dementsprechend mehrere Sprachen unterstützen:
 - ◆ Texte: Meldungen, Beschriftungen von Widgets, Hilfetexte, ...
 - ◆ Formatierungen von Daten: Zahlenformate, Datum- und Uhrzeitangaben, Währungen, Telefonnummern, Adressen
 - ◆ Maßeinheiten, Anreden, akademische Grade, ...
 - ◆ Multimediale Daten: Bilder, Videos, Sprachein- und ausgaben, Farben
- Eventuell müssen Unterscheidungen je nach Betriebssystem getroffen werden:
 - ◆ Zeichensatznamen
- Problem: Soll für jede Übersetzung eine Kopie des Quelltextes verändert werden? Konsequenzen:
 - ◆ Pflege vieler Programmkopien für neue Versionen.
 - ◆ Übersetzer sollten besser nicht den Quelltext anfassen...



Erkennung des Landes und der Sprache

- Java hat eine eingebaute Unterstützung für beliebige Sprachen durch sogenannte Locales (geographische, politische oder kulturelle Region):
- Damit kann ein Entwickler Code mit den folgenden Eigenschaften erstellen:
 - ◆ Übersetzung der Anwendung in verschiedene Sprachen
 - ◆ Parallel Verwendung mehrerer Ortseinstellungen
 - ◆ Interne Ablage von Datums-, Zeitwerten, Währungen usw. in einem sprachunabhängigen Format, Ausgabe für den Benutzer aber in seinem bevorzugten Format

Beispiele für Datumsangaben:

November 3, 1997 (English)

3 novembre 1997 (French)

Beispiel für Zahlen:

\$1,234.50 (U.S. Währung) 1.234,50 € (Deutsche Währung)

- Verwendung der Klassen des Pakets **java.text**



Verwaltung der Übersetzungen von Texten sowie der Bilder

- Im Quelltext befindet sich nur ein Verweis (ein Schlüssel) auf den eigentlichen Text.
- Der sprachabhängige, auszugebende Text wird aus einer anderen Quelle (z.B. einer Datei) gelesen.
- Vorteile:
 - ◆ Ein Übersetzer braucht nur die Sprachdateien anzupassen.
 - ◆ Es lassen sich identische Texte mehrfach nutzen.
 - ◆ Keine Neuübersetzung der Anwendung bei Sprachwechsel.
 - ◆ Sprachwechsel können sogar zur Laufzeit erfolgen.



- Alle darzustellenden **String**-Werte werden aus einer Datei gelesen (der sogenannten Ressource-Datei).
- Die Datei enthält Zeilen, bestehend aus Schlüssel/Wert-Paaren.
- Der Quelltext enthält nur noch die Schlüssel. Die Werte werden aus der Ressource-Datei gelesen.
- Beispiel (Datei für englische Sprachunterstützung):

```
I18NLoginDialog_Message=A login on server xyz is required.  
I18NLoginDialog_Name=Name:  
I18NLoginDialog_Password=Password:  
I18NLoginDialog_Title=Login
```

- Beispiel (Datei für deutsche Sprachunterstützung):

```
I18NLoginDialog_Message=Es ist eine Anmeldung am Server xyz erforderlich.  
I18NLoginDialog_Name=Name:  
I18NLoginDialog_Password=Passwort:  
I18NLoginDialog_Title=Anmeldung
```



Suche nach der Ressource-Datei

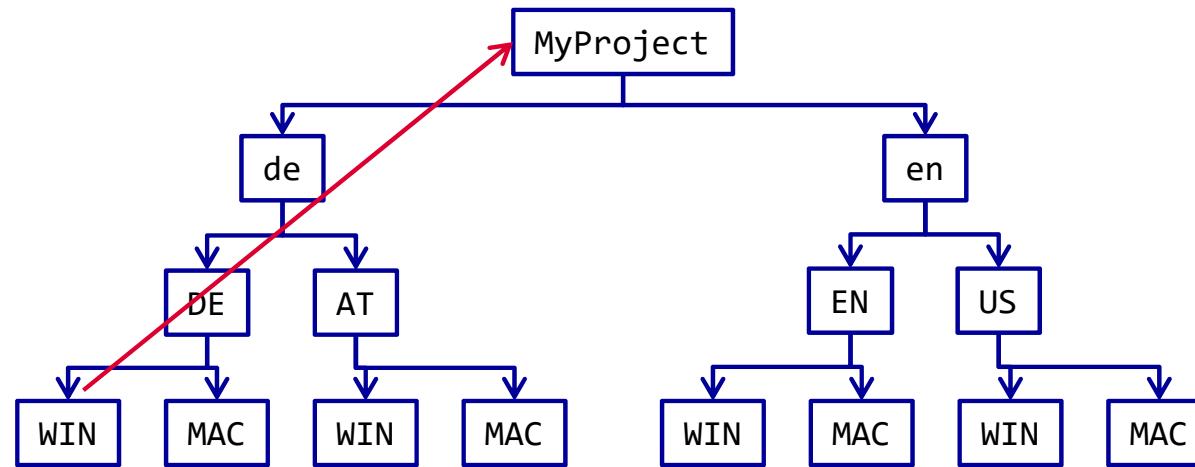
- Der komplett spezifizierte Name der Datei ist:

base + "_" + lang + "_" + country + "_" + var.properties

- ◆ **base**: Basisname der Ressource-Datei
- ◆ **lang**: Sprache im ISO-Sprachcode (ISO 639, http://www.loc.gov/standards/iso639-2/php/code_list.php) der Sprache, deren Texte in der Datei vorhanden sind (**en, de, ...**). Details:
<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>
- ◆ **country**: ISO-Ländercode (ISO 3166, http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm, http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html) der Sprache der Datei (**EN, US, DE, AT, ...**).
- ◆ **var**: Hersteller- oder browserspezifische Varianten des Codes (**WIN, MAC, POSIX, Traditional_WIN, ...**).

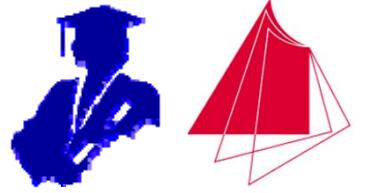


- Hierarchischer Aufbau, Beispiel:



Suchpfad: Aufstieg im Baum von der genauesten
Angabe bis zur ungenauesten

- Anmerkung:
 - ◆ Es können auch Dateien im XML-Format verwendet werden (siehe API-Dokumentation zur Klasse **ResourceBundle.Control**).
 - ◆ Es lässt sich ein eigener Caching-Mechanismus implementieren.



- Problem:
 - ◆ Der Inhalt der Ressource-Datei wurde mit einer Zeichenkodierung erstellt, die weder Latin 1 noch Unicode war.
 - ◆ Dann werden viele Zeichen auf anderen Betriebssystemen vermutlich falsch dargestellt.
- Lösung:
 - ◆ Auf der Plattform, auf der die Datei erstellt wurde, wird mit derselben Kodierung das Programm **native2ascii** aufgerufen.
 - ◆ **native2ascii** konvertiert alle „problematischen“ Zeichen in deren Unicode-Darstellung:
Aus ä wird z.B. \u00e4
- Besser: Dateien gleich als Latin 1 oder Unicode erstellen.
- Beispieldateien der Vorlesung wurden unter Windows 10 erstellt, Darstellung unter Mac OS X und Ubuntu 10 sowie Linux Mint 17 sind einwandfrei.



- Internationalisierung: Klasse **java.util.ResourceBundle**.
- Wichtige Methoden:

Methode	Beschreibung
static ResourceBundle getBundle(String basename)	Erzeugt ein Ressource-Bundle für den angegebenen Datei-Basisnamen mit Standard-Ortseinstellung.
static ResourceBundle getBundle(String basename, Locale locale)	Erzeugt ein Ressource-Bundle für den angegebenen Datei-Basisnamen. Es wird versucht, ein Ressource-Datei für die angegebene Ortseinstellung zu finden.
String getString(String key)	Ermittelt den Wert für den angegebenen Schlüssel.

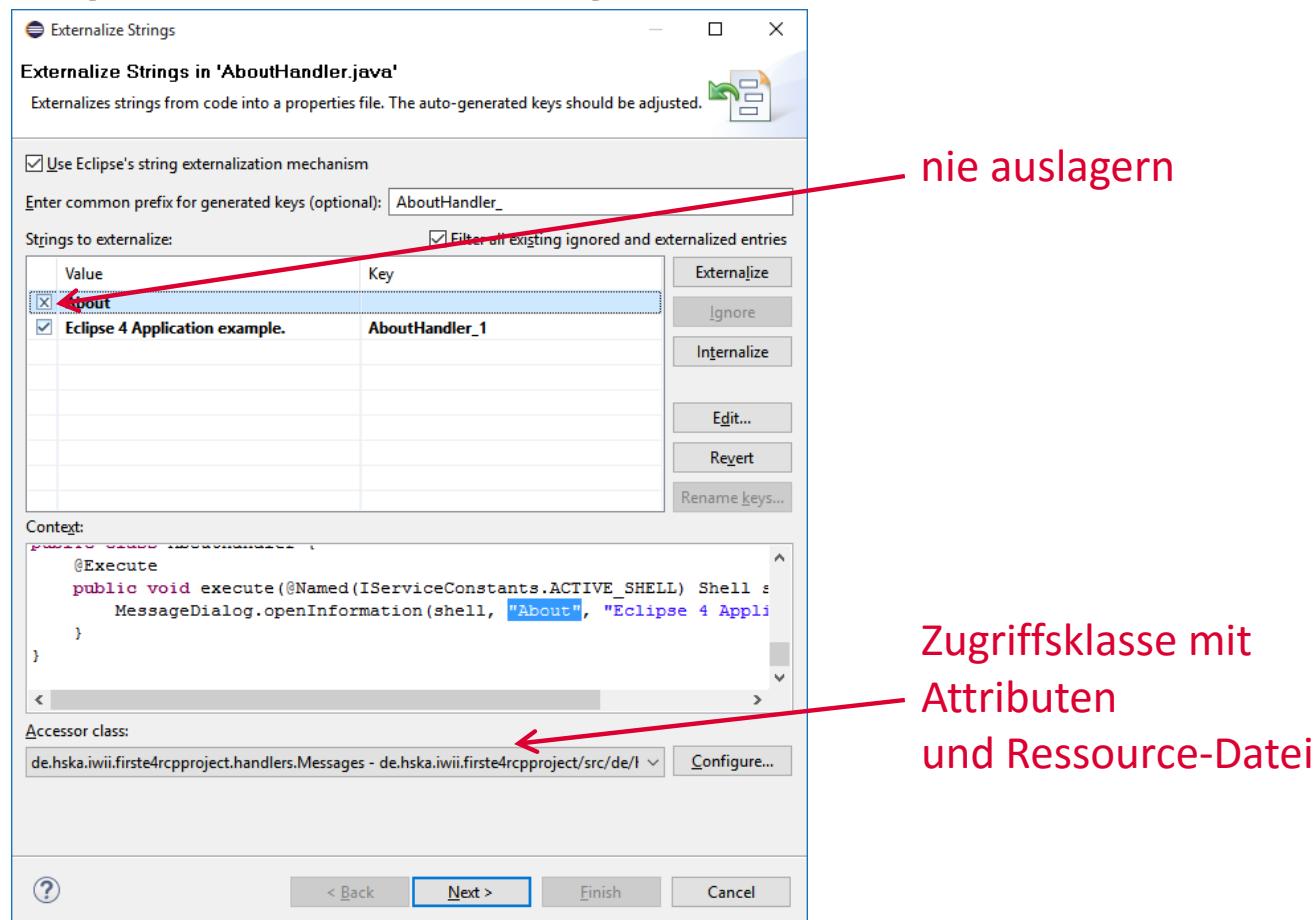
- Im Fehlerfall: **MissingResourceException**, falls entweder die Datei oder der Schlüssel in der Datei nicht vorhanden sind.
- Mehrere Ressource-Dateien lassen sich parallel geöffnet gehalten.

Internationalisierung

Ressource-Dateien – Variante 2



- Vorgehen, wenn die Zeichenketten bereits im Quelltext vorhanden sind:
 - ◆ In Eclipse lassen sich mit **Source → Externalize Strings** die Zeichenketten elegant in eine Datei auslagern.



Internationalisierung

Ressource-Dateien – Variante 2



- Nächste Schritte: Vorschau auf das Ergebnis

The screenshot shows two consecutive steps of the 'Externalize Strings' refactoring dialog for the file 'AboutHandler.java'.

Step 1: The first window displays a list of found problems. It shows one problem: 'Property file 'de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers/messages.properties''. Below this, a message says 'No context information available'. At the bottom are buttons for '?', '< Back', 'Next >', 'Finish', and 'Cancel'.

```
Externalize Strings  
Externalize Strings in 'AboutHandler.java'  
Review the information provided in the list below. Click 'Next >' to view the next item or 'Finish'.  
Found problems  
Property file 'de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers/messages.properties'  
No context information available  
? < Back Next > Finish Cancel
```

Step 2: The second window shows the changes to be performed. It lists three tasks:

- Create file de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers/Messages.java
- AboutHandler.java - de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers
- Create file de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers/messages.prop

Below the list is a preview of the generated Java code for the 'Messages' class. The code includes imports for `de.hska.iwii.firste4rcpproject.handlers` and `org.eclipse.osgi.util.NLS`, and defines a class `Messages` that extends `NLS`. It contains a static final string `BUNDLE_NAME` and a static block to initialize the resource bundle. A private constructor is also shown.

```
Externalize Strings  
Externalize Strings in 'AboutHandler.java'  
The following changes are necessary to perform the refactoring.  
Changes to be performed  
Create file de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers/Messages.java  
AboutHandler.java - de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers  
Create file de.hska.iwii.firste4rcpproject/src/de/hska/iwii/firste4rcpproject/handlers/messages.prop  
package de.hska.iwii.firste4rcpproject.handlers;  
  
import org.eclipse.osgi.util.NLS;  
  
public class Messages extends NLS {  
    private static final String BUNDLE_NAME = "de.hska.iwii.firste4rcpproject.handlers.Messages";  
    public static String AboutHandler_1;  
  
    static {  
        // initialize resource bundle  
        NLS.initializeMessages(BUNDLE_NAME, Messages.class);  
    }  
  
    private Messages() {  
    }  
}  
? < Back Next > Finish Cancel
```



- Zum Schluss werden aus Zeilen wie dieser

```
MessageDialog.openInformation(shell, "About", "Eclipse 4 Application example.");
```

die Zeichenketten ausgelagert:

```
MessageDialog.openInformation(shell, Messages.AboutHandler_0,  
                             Messages.AboutHandler_1);
```

- Der Zugriff erfolgt über eine generierte Klasse mit Attributen, die aus einer Ressource-Datei gefüllt werden:

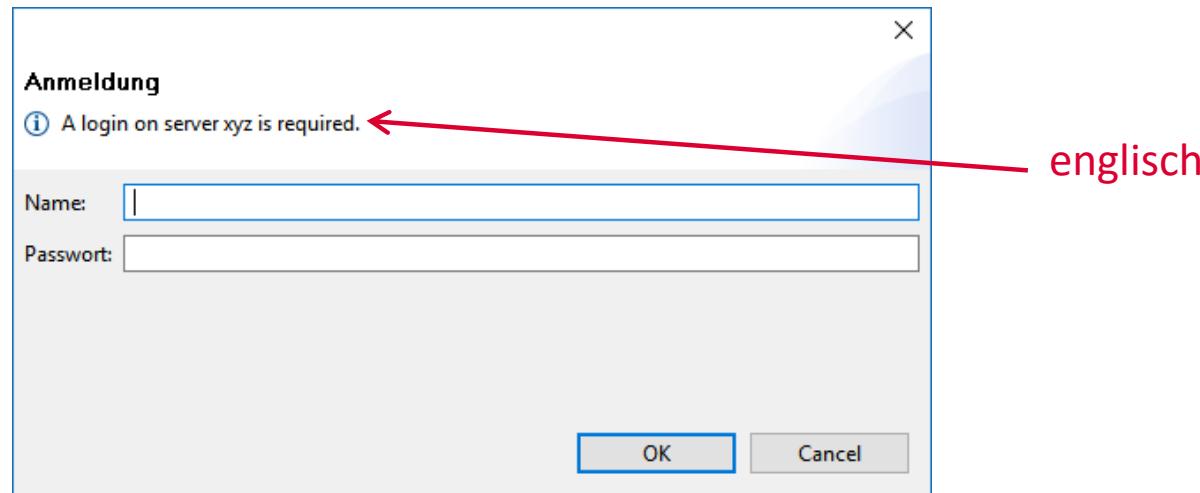
```
private static final String BUNDLE_NAME =  
    "de.hska.iwii.i18nrcpproject.handlers.messages"; //NON-NLS-1$  
public static String AboutHandler_0;  
public static String AboutHandler_1;  
static {  
    // initialize resource bundle  
    NLS.initializeMessages(BUNDLE_NAME, Messages.class);  
}  
  
private Messages() {  
}
```



- Zeichenketten, die nie ausgelagert werden sollen, werden automatisch mit dem Kommentar **//\$NON-NLS-1\$** versehen. Beispiel:

```
FormLayout layout = new FormLayout(  
    "4dlu, left:pref, 4dlu, fill:max(80dlu;pref):grow, 4dlu", //$/NON-NLS-1$  
    "4dlu, pref, 4dlu, pref, 4dlu"); //$/NON-NLS-1$
```

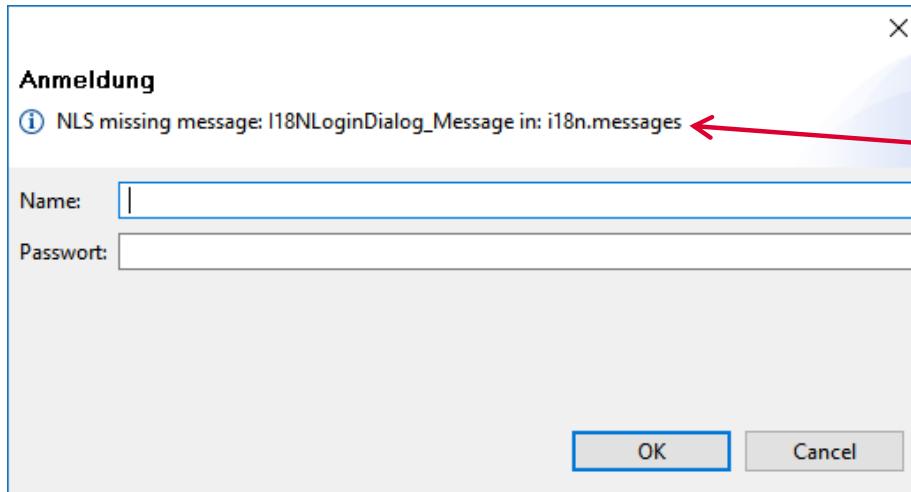
- Was passiert, wenn der gesuchte Text (hier deutsch) fehlt?



- Es wird der Text der Standardsprache (in diesem Fall englisch) verwendet.



- Was passiert, wenn ein Text gar nicht vorhanden ist?

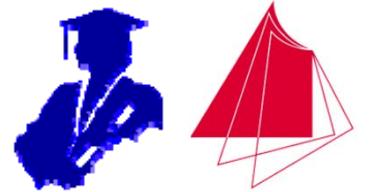


Fehlermeldung
und Verwendung
des Schlüssels

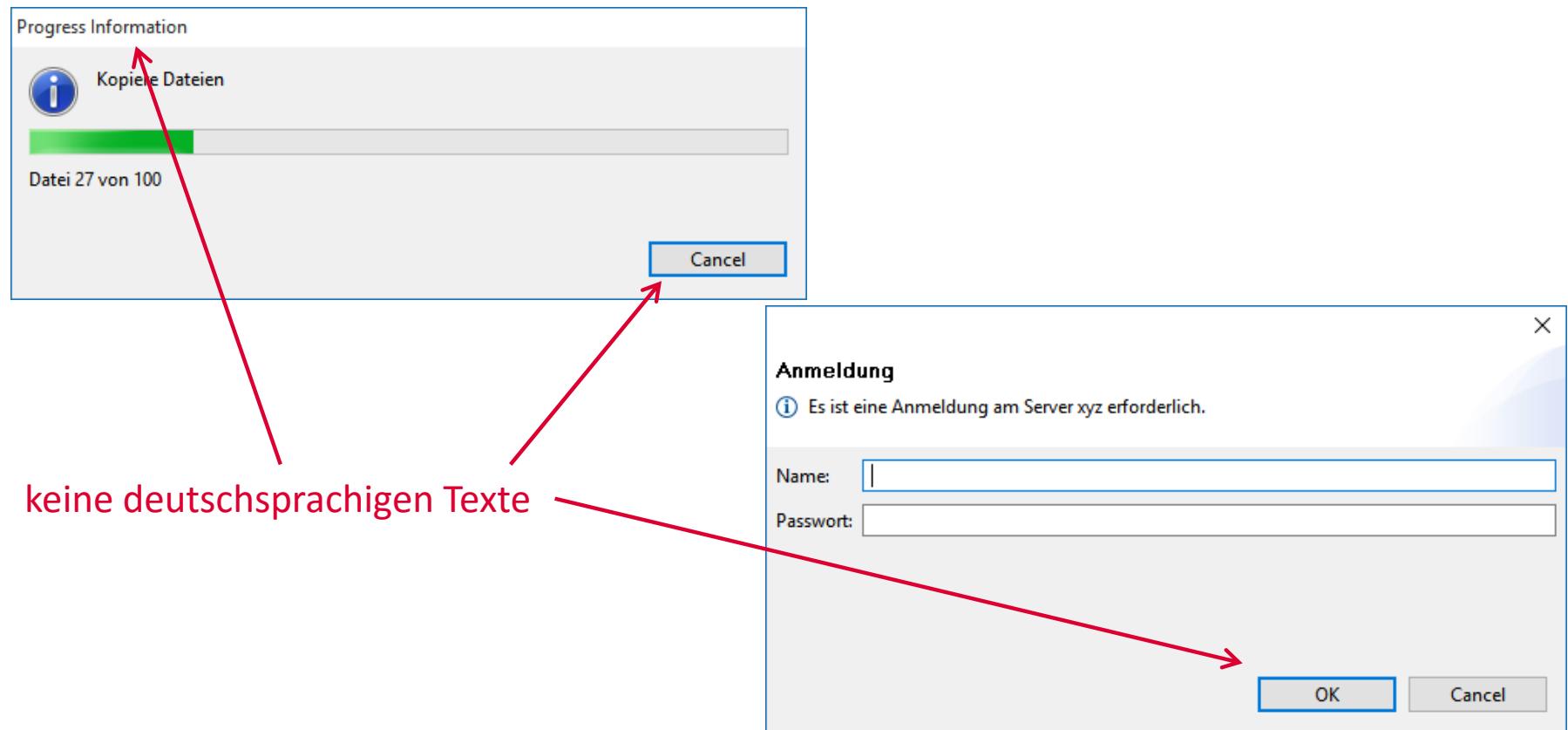
Zusätzlich: Konsolenausgabe

```
Warning: NLS missing message: I18NLoginDialog_Message in: i18n.messages
```

- Bei nicht verwendeten Texten erfolgt auch eine Warnung.



- JFace besitzt für Dialoge bereits Standardtexte, die aus Ressource-Dateien ausgelesen werden:
 - Beschriftungen für „OK“, „Abbruch“ und weitere Tasten
 - Dialogüberschriften:





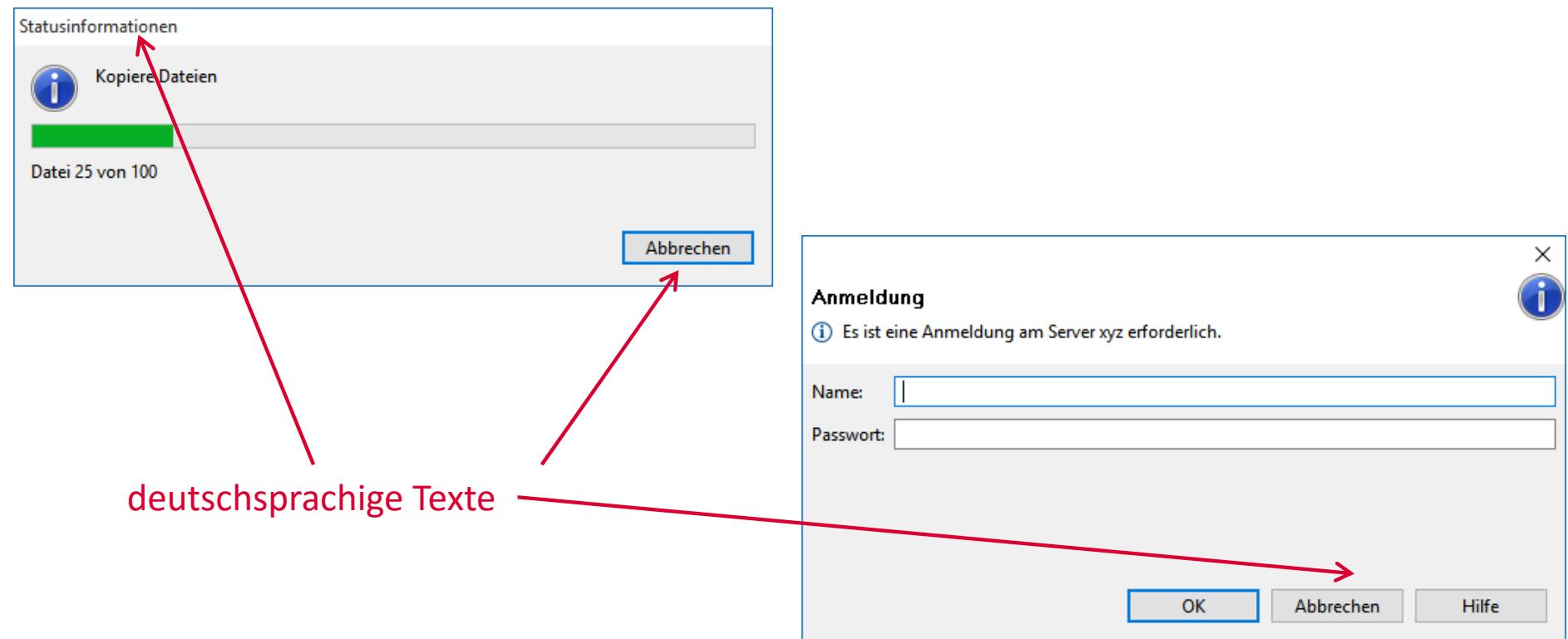
- Eclipse bzw. das RCP-SDK werden nur mit englischsprachigen Texten ausgeliefert:

Name	Größe	Gepackte Größe	Geändert am	Erstell ^
layout	37 850	16 506	2015-06-02 20:36	
menus	2 987	1 482	2015-06-02 20:36	
operation	23 574	10 093	2015-06-02 20:36	
preference	239 833	102 535	2015-06-02 20:36	
resource	128 609	56 246	2015-06-02 20:36	
util	119 058	53 078	2015-06-02 20:36	
viewers	662 154	288 948	2015-06-02 20:36	
window	63 816	27 958	2015-06-02 20:36	
wizard	74 187	34 163	2015-06-02 20:36	
messages.properties	7 883	2 213	2015-06-02 20:36	

- Für andere Sprachen müssen die so genannten „Language Packs“ nachinstalliert werden: <http://www.eclipse.org/babel/downloads.php> (siehe Einführung zur Installation)



- Die Versionsnummer muss nicht exakt passen, da die Übersetzungen nie ganz aktuell sind.
- Wegen RCP ist die Komplettinstallation des „Language Pack“ trotzdem sinnvoll → Updatefähigkeit.
- Dann sieht das Ergebnis so aus:





- Aber: Was passiert mit den Texten im Anwendungs-Modell?

The screenshot shows the Eclipse RCP Application Model Editor (*Application.e4xmi) with the 'Handled Menu Item' configuration for the 'Log in' menu item.

Left Panel (Tree View):

- Application
- Add-ons
- Binding Contexts
- BindingTables
- Handlers
- Commands
- Command Categories
- Windows and Dialogs
 - Trimmed Window - de.hskal.iwii.commande4rc
 - Main Menu
 - File
 - Handled Menu Item - Simulate
 - Handled Menu Item - Log in
 - Handled Menu Item - Save
 - Handled Menu Item - Quit

Right Panel (Handled Menu Item Configuration):

Handled Menu Item

 - ID:** de.hskal.iwii.commande4rcproject.handledmenuitem.login
 - Type:** Check (circled in red)
 - Label:** Log in (circled in red)
 - Mnemonics:** [empty]
 - Tooltip:** [empty]
 - Icon URI:** [empty]
 - Enabled:**
 - Selected:**
 - Visible-When Expression:** <None>
 - Command:** loginCommand - de.hskal.iwii.commande4rcpp
 - To Be Rendered:**
 - Visible:**

Bottom Navigation:

 - Form
 - List
 - XML



- Auch diese Texte lassen sich auslagern:

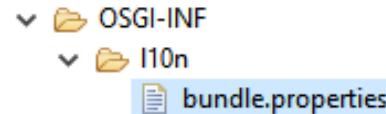
The screenshot shows the Eclipse RCP Application.e4xmi editor. A context menu is open over an element in the tree view, with 'Externalize Strings' selected. A red arrow points from the menu entry to the 'Externalize Strings' dialog window. The dialog title is 'Externalize Strings' and contains the text: 'Externalizing manifest files extracts translatable strings and stores them in a properties file for multi-language support.' It features a table with columns: Element, Attribute, Key, and Value. The table data is as follows:

Element	Attribute	Key	Value
TrimmedWindow	label	trimmedwindow.label.2	de.hska.iwii.commande4rcpproject
HandledMenuItem	label	handledmenuitem.label.8	Delete
HandledMenuItem	label	handledtoolitem.label.1	Save
Menu	label	menu.label.4	File
HandledMenuItem	label	handledtoolitem.label.2	Login
HandledMenuItem	label	handledtoolitem.label.3	Save
Command	commandName	command.commandname.1	quitCommand
Command	commandName	command.commandname.2	saveCommand
Command	commandName	command.commandname.3	loginCommand
Command	commandName	command.commandname.4	simulateCommand

At the bottom of the dialog are 'OK' and 'Cancel' buttons.



- Die Datei **OSGI-INF/I10N/bundle.properties** bzw. mit den entsprechenden Länder- und Sprachkürzeln nimmt die Schlüssel/Werte-Paare auf.
- Im Modell stehen nur noch die Schlüssel (mit % eingeleitet).



```
bundle.properties
trimmedwindow.label.1 = de.hska.iwii.commande4Rrcpproject
part.label.1 = Daten
menu.label.1 = M1
handledMenuItem.label.1 = Find
menu.label.2 = VM1
handledMenuItem.label.2 = Delete
popupmenu.label.1 = Popup
handledMenuItem.label.3 = M1
menu.label.3 = File
menu.mnemonics.1 = F
handledMenuItem.label.4 = Simulate
handledMenuItem.mnemonics.1 = i
handledMenuItem.label.5 = Log in
handledMenuItem.label.6 = Save
handledMenuItem.mnemonics.2 = S
handledMenuItem.label.7 = Quit
```

ID	de.hska.iwii.commande4rcpproject.handledmenuiten
Type	Check
Label	%handledMenuItem.label.5
Mnemonics	
Tooltip	
Icon URI	
Enabled	<input checked="" type="checkbox"/>
Selected	<input type="checkbox"/>
Visible-When Expression	<None>
Command	loginCommand - de.hska.iwii.comma
To Be Rendered	<input checked="" type="checkbox"/>
Visible	<input checked="" type="checkbox"/>
Default	Supplementary

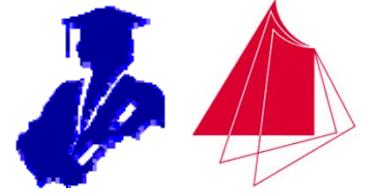


- Das Auslagern von Texten aus der Datei **plugin.xml** funktioniert genauso.
- In Eclipse 4 lässt sich der Translation-Service sehr elegant nutzen, um Werte aus Ressource-Dateien auszulesen:

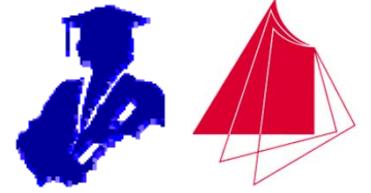
```
@Inject  
private TranslationService trService;  
  
// ...  
  
// Liest den Wert zum Schlüssel key aus der Datei  
// /OSGI-INF/I10N/bundle.properties  
// des Bundles mit der ID Plugin-ID  
String value = trService.translate("%key", "platform:/plugin/Plugin-ID");
```

- Lesenswert für Eclipse-Plug-ins:

<http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html>



- Nachteile der Variante 2:
 - ◆ Statische Attribute und Methoden verhindern, dass Klassen durch den Garbage-Collector beseitigt werden können.
 - ◆ Sprachwechsel zur Laufzeit werden nicht unterstützt.
- Ausweg:
 - ◆ Mit Eclipse 4.4 (Luna) gibt es eine neue „Eclipse Message Extension“.
 - ◆ Statt statischer Attribute werden nicht-statische verwendet.
 - ◆ Sprachwechsel zur Laufzeit sind möglich.
 - ◆ Leider noch keine IDE-Unterstützung → manuelle Arbeit erforderlich.
- Vorgehen:
 - ◆ Abhängigkeit in **Manifest.MF** zum Plug-in **org.eclipse.e4.core.services** herstellen.
 - ◆ Klassen, die auf **Messages** enden, werden automatisch mit den Werten aus dem OSGi-spezifischen Ressource-Bundle gefüllt.
 - ◆ Die Klassen sind reine POJOs.



- Beispiel:

```
public class MyMessage {  
    public String name_label; // Attribut-Name = Schlüssel-Name aus der Datei  
    public String ageLabel; // entspricht dem Schlüsselnamen age.label  
}
```

- Die Ressource-Dateien werden automatisch gefunden, wenn Sie im selben Pfad liegen.
- Mit der Annotation **@Message** kann der Ort der Ressource-Datei (z.B. in einem anderen Plug-in) angegeben werden.
- Die Message-Klasse wird per Dependency-Injection in eigene Klassen eingefügt (**@Inject**).
- Die Message-Klasse kann eigene Initialisierungen vornehmen (**@PostConstruct**).
- Es gibt viele weitere Möglichkeiten (Beeinflussung des Cachings, Sprachwechsel zur Laufzeit, ...) → aus Zeitgründen hier weggelassen.



Allgemeine Zahlenformate

- In Abhängigkeit von Land und Sprache werden Zahlen unterschiedlich dargestellt.
- Die Zahl 345987.246 wird folgendermaßen ausgegeben:
 - ◆ 345 987,246 (fr_FR)
 - ◆ 345.987,246 (de_DE)
 - ◆ 345,987.246 (en_US)
- Die Klasse **NumberFormat (java.text)** bietet die notwendige Funktionalität (Beispiel für Standard-Ortseinstellung):

```
double amount = 345987.246;
String amountOut;
```

```
NumberFormat numberFormatter = NumberFormat.getNumberInstance();
amountOut = numberFormatter.format(amount);
System.out.println(amountOut);
```



Währungsformate

- Das Währungsformat sowie die Währung selbst werden vom Land bestimmt.
- Der Betrag 9876543.21 wird folgendermaßen ausgegeben:
 - ◆ 9 876 543,21 € (fr_FR)
 - ◆ 9.876.543,21 € (de_DE)
 - ◆ \$9,876,543.21 (en_US)
 - ◆ Die Klasse **NumberFormat** (`java.text`) bietet die notwendige Funktionalität (Beispiel für Standard-Ortseinstellung):

```
double amount = 9876543.21;
String amountOut;
```

```
NumberFormat currFormatter = NumberFormat.getCurrencyInstance();
amountOut = currFormatter.format(amount);
System.out.println(amountOut);
```



Datumsformate

- Das Datumsformat wird vom Land bestimmt.
- Beispieldaten für ein Datum:
 - ◆ 9 avr 98 (fr_FR)
 - ◆ 9.4.1998 (de_DE)
 - ◆ 09-Apr-98 (en_US)
- Programmcodebeispiel (Klasse **DateFormat** aus **java.text**):

```
DateFormat dateFormatter = DateFormat.getDateInstance(
    DateFormat.DEFAULT);
Date today = new Date();
String dateOut = dateFormatter.format(today);
System.out.println(dateOut);
```
- Neben frei wählbaren Mustern, mit denen die Ausgabe formatiert werden kann, existieren vordefinierte Stile.



Zeitformate

- Das Zeitformat wird vom Land bestimmt.
- Beispieldarstellung für eine Zeit:
 - ◆ 15:58:45 (de_DE)
 - ◆ 3:58:45 PM (en_US)
- Codebeispiel (Klasse **DateFormat** aus **java.text**):

```
DateFormat timeFormatter =  
    DateFormat.getTimeInstance(DateFormat.DEFAULT);  
Date today = new Date();  
String dateOut = timeFormatter.format(today);  
System.out.println(dateOut);
```

- Weiterhin kann mit **getDateTimeInstance** ein Formatierer für Datum und Zeit erzeugt werden.
- Analog zum Datumsformat existieren dieselben Stil-Abkürzungen auch für das Zeitformat.



Nachrichtenformate

- Die Ressource-Datei kann Platzhalter im Wert eines Schlüssels enthalten:

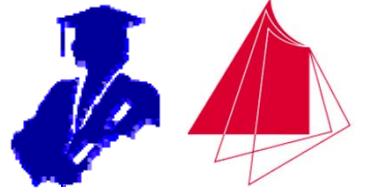
```
Greeting = You've logged in {0} times before. Last login was {1}.
```

- Programmzugriff:

```
int      count      = 42;
Date    lastLogin   = new Date();
Object[] values    = new Object[]{ new Integer(count),
                                    lastLogin };
ResourceBundle bundle = ResourceBundle.getBundle("MyBunde");
String text         = bundle.getString("Greeting");
text               = MessageFormat.format(text, values);
```

Korrekter Plural

- There are **no** files on disc.
- There is **one** file on disc.
- There are **2** files on disc.
- Es existiert die Klasse **ChoiceFormat** aus **java.text**, die genau dieses Problem relativ einfach löst → soll hier nicht besprochen werden.



- Zeichenketten können prinzipiell einfach miteinander verglichen oder sortiert werden:

```
Arrays.sort(names);
boolean before = s1.compareTo(s2) < 0;
```

- Länge einer Zeichenkette:

```
int length = s1.length();
```

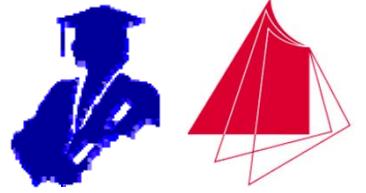
- Problem 1:

- ◆ Länderkonventionen: Wie werden lokale Zeichen wie á, à, â, å, æ usw. behandelt (z.B. kann Michèle gleich Miche`le sein)?
- ◆ Groß- und Kleinschreibung sollte ignoriert werden (siehe Lexikon)
- ◆ Lösung: Vergleich mit Hilfe der Klasse **java.text.Collator** → Berücksichtigung der Ländereinstellung:

```
Collator collator = Collator.getInstance();
Arrays.sort(names, collator);
boolean before = collator.compare(s1, s2) < 0;
```

Oder für ein bestimmtes Land:

```
Collator collator = Collator.getInstance(Locale.US);
```

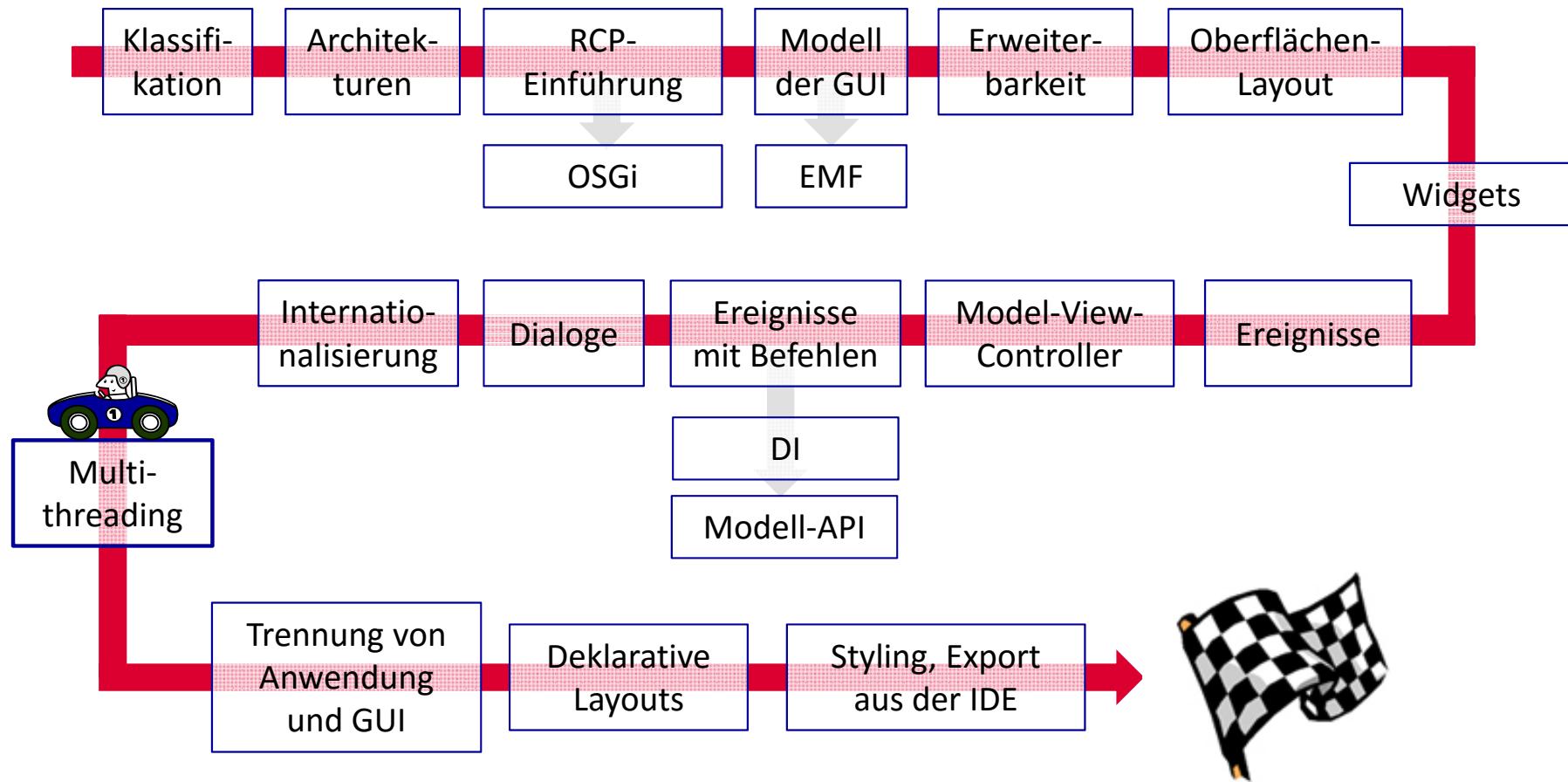
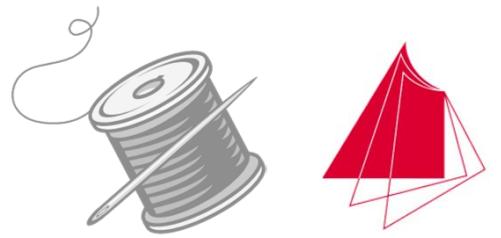


- Problem 2: Die Länge einer Zeichenkette entspricht nicht unbedingt der Anzahl Zeichen!
 - ◆ Mit Java 5 wurde der Unicode-Standard 4 eingeführt.
 - ◆ Die vielen neuen Zeichen „passen“ nicht mehr in die 16 Bit eines **char** → Einführung sogenannter „surrogate pair“ → Zwei **char** enthalten einen Buchstaben.
 - ◆ Die neuen Buchstaben in dem Bereich U+10000 bis U+10FFFF werden „supplementary characters“ genannt.
 - ◆ Die **length**-Methode zählt in **char**-Einheiten → sie kennt keine „supplementary characters“.
 - ◆ Ermittlung der Anzahl Zeichen:

```
String s2 = "abcd\u5B66\uD800\uDF30";
int chars = s2.codePointCount(0, s2.length());
```
- Zusammenfassung:
<http://java.sun.com/mailers/techtips/corejava/2006/tt0822.html>

Multithreading

Einführung



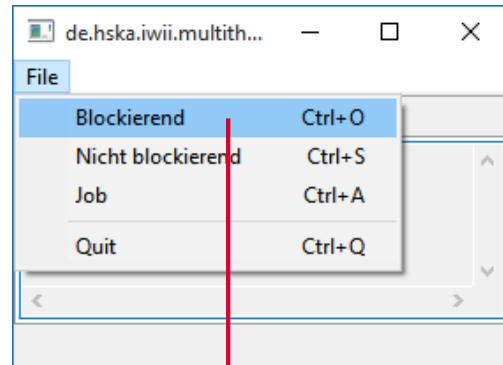


Time is nature's way of keeping everything from happening at once (Woody Allen).

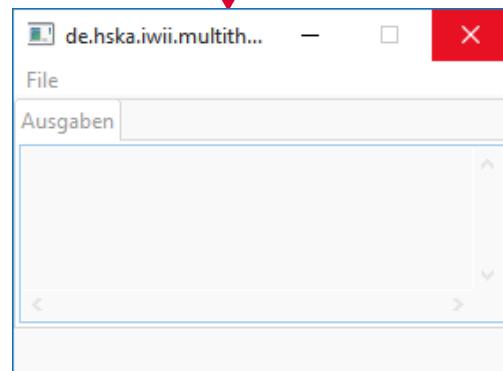
- Der Start eines Java-Programms führt zum Start eines einzigen Threads.
 - ◆ **main**-Methode bei Applikationen
 - ◆ **init**-Methode bei Applets
- Die Ereignisschleife wird sowohl bei SWT als auch JFace in dem Thread (UI-Thread) ausgeführt, in dem das **Display**-Objekt der Anwendung erzeugt wurde:
 - ◆ Alle Widget-Zugriffe müssen im UI-Thread erfolgen. Ansonsten tritt eine **SWTException** („Invalid thread access“) auf.
 - ◆ Alle Beobachter werden im Kontext des UI-Threads aufgerufen → in der Ereignisbehandlung kann direkt auf Widgets zugegriffen werden.



- Daraus resultieren Probleme:
 - ◆ Eine sehr zeitaufwändige Operation blockiert die Oberfläche der Anwendung. Es erfolgt auch kein Neuzeichnen mehr!
 - ◆ Beispiel ([multithreadinge4rcpproject](#)):



Anwendung blockiert





- Ein aktiver Thread kann testen, ob er der UI-Thread ist:
Display.getDefault().getThread() == Thread.currentThread()
- Da die Ereignisbehandlung im UI-Thread stattfindet, sollte dieser nie zu lange blockiert sein:
 - ◆ Wenn eine Ereignisbehandlung sehr zeitaufwändig ist, sollte diese in einem eigenen Thread stattfinden
 - ◆ Sonst würde die Benutzeroberfläche in der Zeit weder auf Benutzereingaben reagieren noch sich selbst neu zeichnen.



Herstellen der Thread-Sicherheit

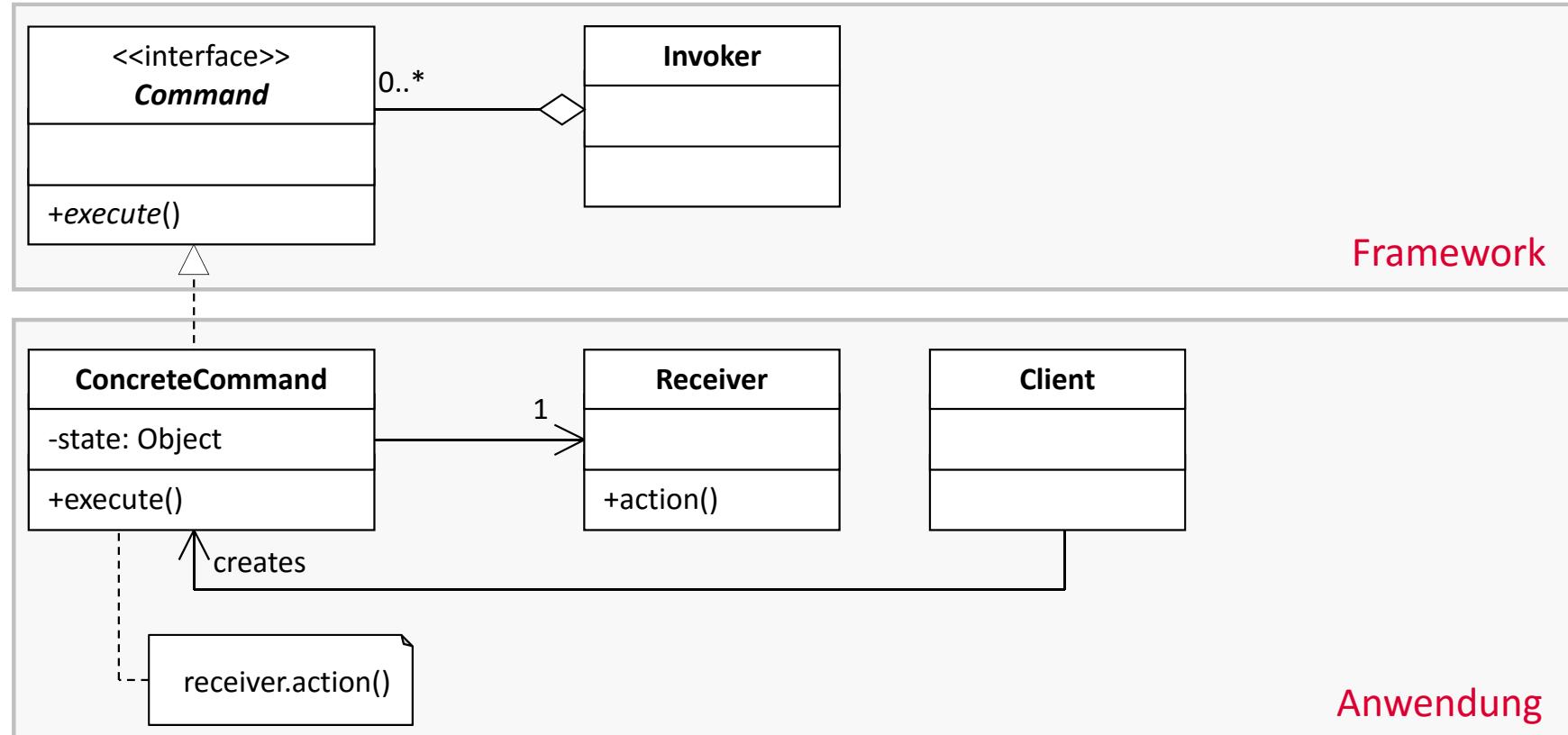
- Wenn andere Threads auf die Benutzeroberflächen-Komponenten zugreifen wollen, müssen sie dieses innerhalb des UI-Threads erledigen → Widerspruch?
- Nein:
 - ◆ Es wird das Befehlsmuster verwendet.
 - ◆ Die auszuführenden Anweisungen werden in einer Klasse gekapselt.
 - ◆ Ein Objekt der Klasse wird dem UI-Thread übergeben.
 - ◆ Die Klasse **Display** führt eine Methode der Klasse im UI-Thread aus.

Multithreading

Befehlsmuster



■ Befehlsmuster (Command Pattern)

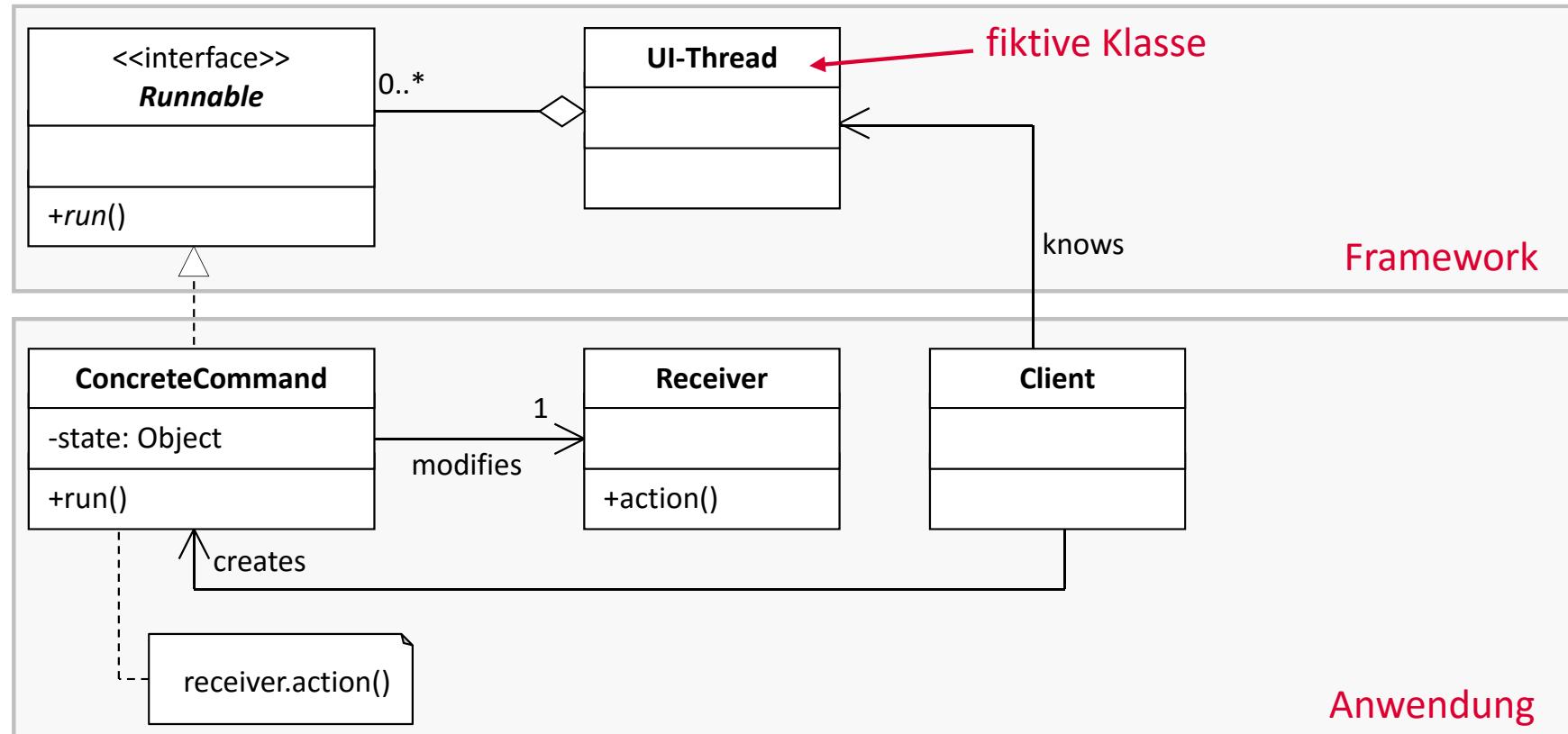


Multithreading



Befehlsmuster

- Implementierung des Befehlsmusters für die Threadsicherheit:



- Der Client erzeugt das Befehlsobjekt und übergibt es dem UI-Thread.
- Der UI-Thread führt in seinem eigenen Kontext den Befehl durch Aufruf von `run` aus.



- Die Klasse **Display** bietet zwei Methoden, die es erlauben, beliebige Anweisungen im UI-Thread auszuführen. Die Anweisungen müssen in der **run**-Methode einer Klasse stehen, die die Schnittstelle **Runnable** implementiert.
- Aufbau der Schnittstelle **Runnable**:

```
public interface Runnable {  
    void run();  
}
```



- **Asynchrone** Ausführung im UI-Thread
 - ◆ **public void asyncExec(Runnable run):**
 - Die Methode fügt ein **Runnable**-Objekt in die Ereignis-Queue ein.
 - Die Methode **run** des Objekts wird irgendwann aufgerufen.
 - Eine Ergebnisrückgabe ist nicht möglich, da der eigene Thread weiterläuft.
 - Die Methode **run** darf keine nicht abgefangene Ausnahme auslösen.
 - Die **run**-Methode muss prüfen, ob die Widgets, auf die zugegriffen werden soll, eventuell schon mit **dispose()** freigegeben wurden.
 - ◆ Beispiel (Ausschnitt):

```
Runnable runnable = new Runnable() {  
    public void run() {  
        doSomething();  
    }  
}
```

```
display.asyncExec(runnable);
```

bzw. ab Java 8 kürzer:

```
display.asyncExec(this::doSomething());
```



- **Synchrone** Ausführung im UI-Thread

- ◆ **public void syncExec(Runnable run):**

- Fügt ein Objekt der Schnittstelle **Runnable** in die Ereignis-Queue ein.
 - Die Methode **run** des Objekts wird irgendwann aufgerufen.
 - Der aufrufende Thread wird solange blockiert, bis die Methode abgearbeitet wurde.
 - Es kann ein Wert zurückgegeben werden. Der Mechanismus dazu muss vom Entwickler selbst erstellt werden.
 - Die Methode **run** darf keine nicht abgefangene Ausnahme auslösen.
 - Die **run**-Methode muss prüfen, ob die Widgets, auf die zugegriffen werden soll, eventuell schon mit **dispose()** freigegeben wurden.
 - Wird **syncExec** vom UI-Thread aufgerufen, so wird die **run**-Methode direkt beim Aufruf von **syncExec** ausgeführt.



- ◆ Beispiel (Ausschnitt):

```
// Implementiert durch eine anonyme innere Klasse
Runnable runnable = new Runnable() {
    public void run() {
        doSomething();
    }
}
```

```
display.syncExec(runnable);
```

bzw. ab Java 8 kürzer:

```
display.syncExec(this::doSomething());
```



- Zeitgesteuerte Ausführung im UI-Thread
 - ◆ **public void timerExec(int millis, Runnable run):**
 - Fügt ein **Runnable**-Objekt nach **millis** Millisekunden in die Queue ein.
 - Die Methode **run** des Objekts wird danach aufgerufen.
 - Es kann kein Wert zurückgegeben werden.
 - Die Methode **run** darf keine nicht abgefangene Ausnahme auslösen.
 - Die **run**-Methode muss prüfen, ob die Widgets, auf die zugegriffen werden soll, eventuell schon mit **dispose()** freigegeben wurden.



- ◆ Beispiel (Ausschnitt):

```
// Implementiert durch eine anonyme innere Klasse
Runnable runnable = new Runnable() {
    public void run() {
        doSomething();
    }
}
```

```
display.timerExec(1000, runnable); // 1000 msec
```

bzw. ab Java 8 kürzer:

```
display.timerExec(1000, this::doSomething());
```



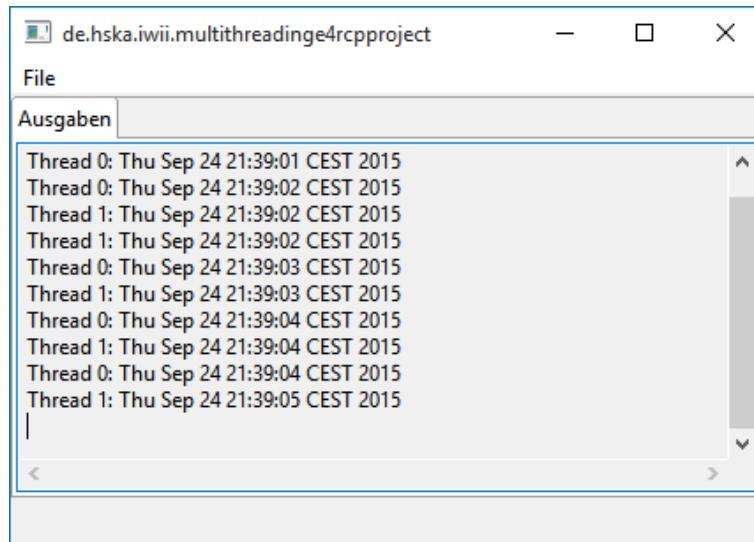
- Kleine Unschönheit: **Display** ist nicht im Kontext gespeichert und kann daher nicht per Dependency Injection übergeben werden.
- Die kleine Wrapper-Klasse **UISynchronize** bietet Abhilfe:

```
public class UISynchronize {  
    public void asyncExec(Runnable r) { /* ... */ }  
    public void syncExec(Runnable r) { /* ... */ }  
}
```

- Die Klasse delegiert die Aufrufe einfach an das aktuelle **Display**-Objekt weiter.
- Ein Objekt der Klasse **UISynchronize** ist im Kontext abgelegt. Dieses kann per DI injiziert werden.



- Für alle Methoden gemeinsam gilt:
 - ◆ Das übergebene Objekt wird am Ende der Ereignis-Queue angehängt.
 - ◆ Alle bereits anstehenden Ereignisse werden vorher behandelt.
- Die Methode **asyncExec** wird in der Regel dann benötigt, wenn lediglich eine Ausgabe notwendig ist (Update der Oberfläche).
- Erster Versuch: Das folgende Beispiel startet mehrere Threads, die abwechselnd Meldungen in ein **Text**-Widget ausgeben.





- Quelltext des Messenger-Threads (Datei **Messenger.java**):

```
public class Messenger extends Thread {  
    private View view;  
    private Display display;  
    private volatile boolean stopped = false;  
  
    public Messenger(Display display, String name, View view) {  
        this.display = display;  
        this.view = view;  
        setName(name);  
    }  
  
    public void quit() {  
        stopped = true;  
        interrupt();  
    }  
}
```

Multithreading



Befehlsmuster: Beispiel

```
@Override  
public void run() {  
    while (!stopped) {  
        try {  
            sleep((int) (600 + 400 * Math.random()));  
            if (!stopped && !display.isDisposed()) {  
                display.asyncExec(new Runnable() {  
                    @Override  
                    public void run() {  
                        if (!view.isDisposed()) {  
                            view.append(getName() + ": " + new Date() + "\n");  
                        }  
                    }  
                });  
            }  
        } catch (InterruptedException e) {  
        }  
    }  
}
```



- Nachteil von **asyncExec** und **syncExec**:
 - ◆ Viel manuelle Programmierung zur Synchronisation erforderlich:
 - Häufig müssen Ergebnisse einer Berechnung des Threads ausgegeben werden.
 - Die Ausgabe muss im UI-Thread stattfinden.
 - ◆ Synchronisation kann leicht vergessen werden → aber es gibt ja eine Ausnahme!



- Bei vielen Hintergrundaktionen ist mehr zu beachten:
 - ◆ Der Benutzer muss über den Fortschritt informiert werden:
 - Wie viele Dateien müssen noch kopiert werden?
 - Wie weit ist der Download fortgeschritten?
 - ◆ Mögliche Deadlocks müssen erkannt und vermieden werden.
 - ◆ Aktionen müssen abgebrochen oder zeitweise angehalten werden.
- Die Klasse **Job** unterstützt solche Aktionen.
 - ◆ Er führt Aktionen (optional) in einem eigenen Thread aus.
 - ◆ Ein **Job** kann den Fortschritt anzeigen:
 - als Dialog mit Fortschrittsbalken
 - mit einem Fortschrittsbalken an selbst definierter Stelle (z.B. in der Statuszeile)
 - ◆ Er kann durch den Anwender unterbrechbar sein.
- Jobs können Threads wiederverwenden → geschieht automatisch mit Hilfe von Thread-Pools.



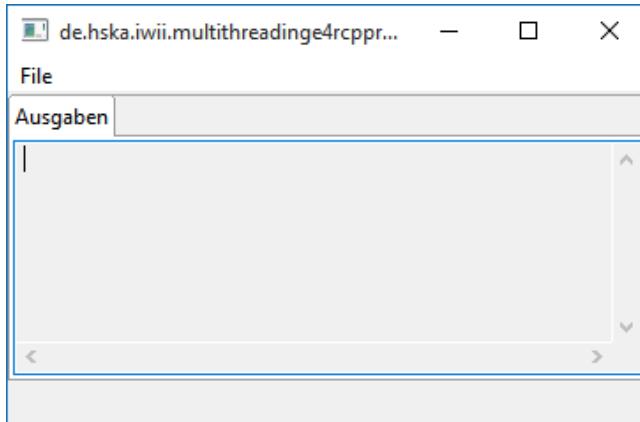
- Wichtige Klassen und Schnittstellen der Job-API:
 - ◆ **Job**: kapselt ausführbare Aktionen
 - ◆ **JobManager**: Scheduler für Jobs
 - ◆ **IProgressMonitor**: zu implementierende Schnittstelle, um Statusinformationen zum Fortschritt des Jobs auszugeben. Die Standardimplementierung ist „leer“ (ohne Ausgabe).
- Jobs haben Zustände (wichtig für Deadlock-Behandlung), u.A.:
 - ◆ **WAITING**: wartet auf die Ausführung
 - ◆ **SLEEPING**: wurde schlafen gelegt, bis er wieder aufgeweckt wird
 - ◆ **RUNNING**: läuft gerade
 - ◆ **NONE**: ist in keinem der oben genannten Zustände



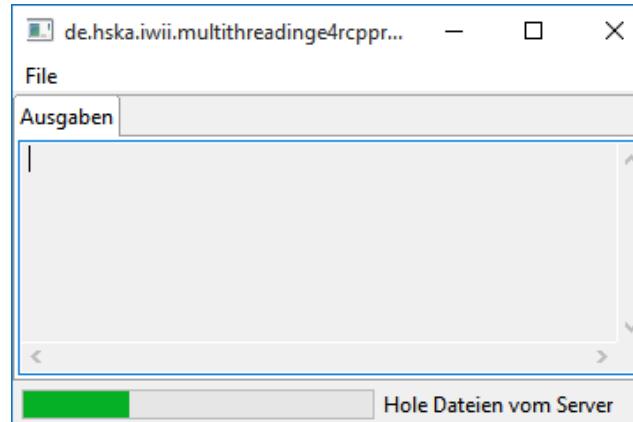
- Fortschrittsausgabe mit dem **IProgressMonitor**:
 - ◆ Klasse erstellen, die **IProgressMonitor** implementiert.
 - ◆ Einige Methoden des **IProgressMonitors**:
 - **void setTaskName(String taskName)**: Bezeichnung für die aktuelle Aufgabe
 - **void beginTask(String description, int steps)**: Aufgabe mit einer Beschreibung starten, Anzahl Schritte bis zur Fertigstellung angeben
 - **void subTask(String subTask)**: Teilaufgabe mit der übergebenen Bezeichnung starten
 - **void worked(int steps)**: Anzahl an Schritten wurde fertiggestellt
 - **void setCanceled(boolean canceled)**: Mitteilen, dass die Aktion abgebrochen wurde
 - **boolean isCancelled()**: Soll die Aktion abgebrochen werden? Der Monitor kann z.B. eine Taste besitzen, über die der Anwender abbrechen kann.
 - **void done()**: Fertigmeldung



- Beispiel, in dem der Fortschritt in der Statuszeile des Fensters ausgegeben wird:



Job läuft nicht,
keine Hintergrundaktion

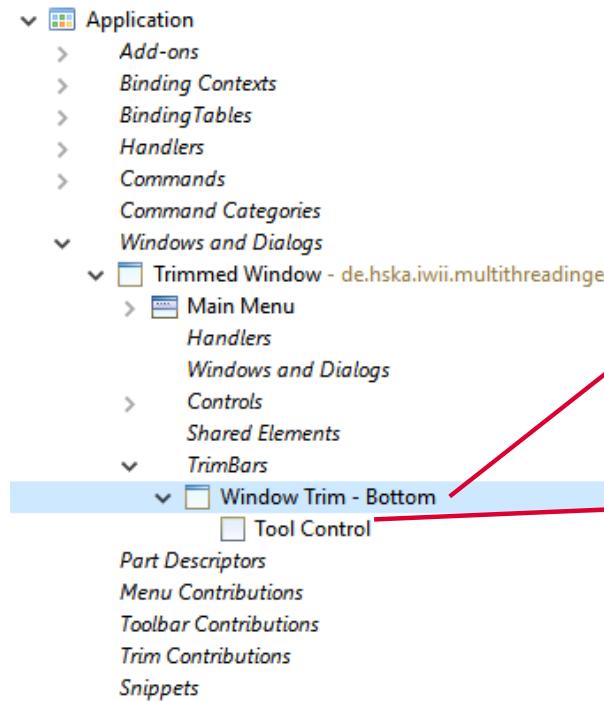


Job läuft, der Monitor
gibt den Fortschritt aus

- Problem: Eine Statuszeile gibt es eigentlich nicht.
- Lösung:
 - ◆ Toolbar ins Anwendungs-Modell eintragen, Position „unten“ festlegen
 - ◆ Widget erstellen, dass **IProgressMonitor** implementiert
 - ◆ dieses Widget in die Toolbar einfügen



- Widget **ProgressToolItem** im Anwendungs-Modell:



The image displays two configuration dialogs. The top dialog is 'Window Trim' with fields for ID (de.hskaiwii.multithreadinge4rcpproject.trimbars.progress), Accessibility Phrase, and Side (set to 'Bottom'). The bottom dialog is 'Tool Control' with fields for ID (de.hskaiwii.multithreadinge4rcpproject.toolcontrol.progress), Accessibility Phrase, Class URI (bundleclass://de.hskaiwii.multithreadinge4rcpproject.toolcontrol.progress), To Be Rendered (checked), and Visible (checked). Red boxes highlight the 'Side' dropdown in the 'Window Trim' dialog and the 'Class URI' field in the 'Tool Control' dialog. Red arrows point from the highlighted areas in the tree view to their corresponding settings in the dialogs.

- Ein **ToolControl** ist ein Widget, dass durch eine eigene Klasse implementiert ist.



- Aufbau des **ProgressToolItems**:



- Läuft kein Job, dann wird das **Composite** verborgen.
- Der Quelltext ist im Projekt und relativ langweilig.

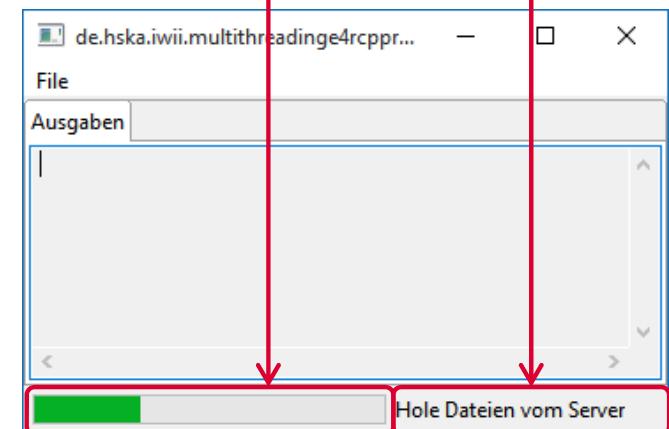


- Erstellung eines **Jobs**:
 - ◆ Klasse erstellen, die von **Job** erbt
 - ◆ Methode **run** überschreiben: In ihr wird die Hintergrundaktivität durchgeführt.
 - ◆ Die **run**-Methode bekommt einen **IProgressMonitor** übergeben, der Informationen über den aktuellen Fortschritt liefert.
 - ◆ Einige Methoden der Job-Klasse:
 - **void setUser(boolean user)**: Eingriff durch den Anwender möglich
 - **void schedule()**: Job starten
 - **void setPriority(int priority)**: Priorität des Jobs festlegen



- Beispiel (Implementierung als anonyme innere Klasse, Teilaufgaben nicht dargestellt):

```
Job job = new Job("Hole Dateien") { → im Beispiel nicht verwendet
    @Override
    protected IStatus run(IProgressMonitor monitor) {
        try {
            // 100 Schritte
            monitor.beginTask("Hole Dateien vom Server", 100);
            for (int i = 0; i < 100; ++i) {
                // immer einen Schritt erledigen
                monitor.worked(1); →
                monitor.subTask("Datei " + i + " von " + 100);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            monitor.setTaskName("Abgebrochen");
            monitor.setCanceled(true);
            return Status.CANCEL_STATUS;
        }
        monitor.done();
        return Status.OK_STATUS;
    }
};
```





- Soll die Aktion durch den Benutzer unterbrechbar sein, dann muss der Abbruch zyklisch abgefragt werden:

```
// ...
monitor.beginTask("Start", 100);
for (int i = 0; i < 100; ++i) {
    // immer einen Schritt erledigen
    monitor.worked(1);
    Thread.sleep(500);

    // Abbrechen der Aktion beachten
    if (monitor.isCanceled()) {
        return Status.CANCEL_STATUS;
    }
// ...
```

- Der Monitor muss also das manuellen Abbrechen irgendwo implementieren, wenn das gewünscht ist.



- Job starten:

```
// Keine Manipulation/kein Abbruch durch den Anwender  
job.setUser(false);  
// Starten  
job.schedule();
```

- Und was passiert bei Beendigung des Programmes, wenn Jobs laufen?

- ◆ Es gibt einen Fehler!
 - ◆ Jobs sollten (müssen) vorher beendet werden.

- Lösung:

- ◆ Jobs mit **cancel** unterbrechen.
 - ◆ Der Job muss auf **cancel** reagieren und sich selbst beenden.
 - ◆ Der Aufrufer von **cancel** wartet auf Beendigung des Jobs.

```
if (job != null && job.getState() != Job.NONE) {  
    job.cancel();  
    try {  
        job.join();  
    } catch (InterruptedException e) {}  
}
```



- Ein Job kann seine Zustandswechsel angemeldeten Beobachtern mitteilen (z.B. wenn abgeholt Daten vorliegen). Beispiel:

```
job.addJobChangeListener(new JobChangeAdapter() {  
    // Job ist fertig, weitere Methoden für andere Zustände  
    // können auch überschrieben werden.  
    @Override  
    public void done(IJobChangeEvent event) {  
        super.done(event);  
        System.out.println("Fertig!");  
    }  
});
```

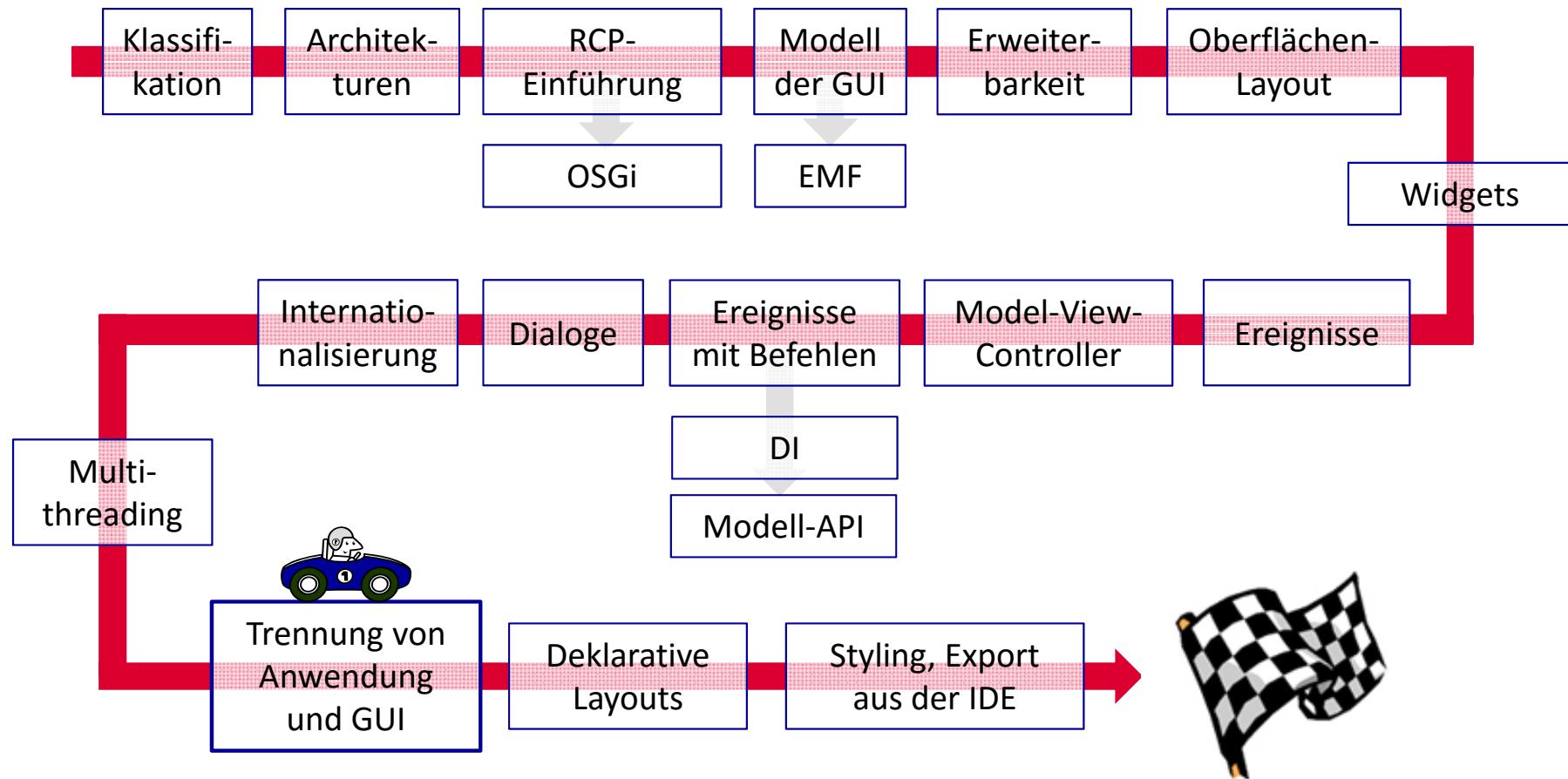
- Achtung: Der Beobachter wird im Kontext des Threads des Jobs aufgerufen und kann somit nicht direkt auf die Widgets zugreifen.



- Wie sieht es mit einem Status-Dialog für Jobs aus?
 - ◆ Es gibt die JFace-Dialog-Klasse **ProgressMonitorDialog**, die einen Standarddialog anzeigt und eine Aktion im Hintergrund durchführt → siehe API
- Fazit:
 - ◆ Job-API manueller Thread-Programmierung vorziehen!
 - ◆ Jobs bieten auch die Benachrichtigung von Beobachtern bei Zustandswechsel.
 - ◆ Jobs müssen beim direkten Zugriff auf Widgets ebenso mit **display.asyncExec** oder **display.syncExec** arbeiten.
- Weiterführende Informationen:
<http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html>
<http://www.vogella.de/articles/EclipseJobs/article.html>

Anbindung an die Geschäftslogik

Übersicht



Anbindung an die Geschäftslogik

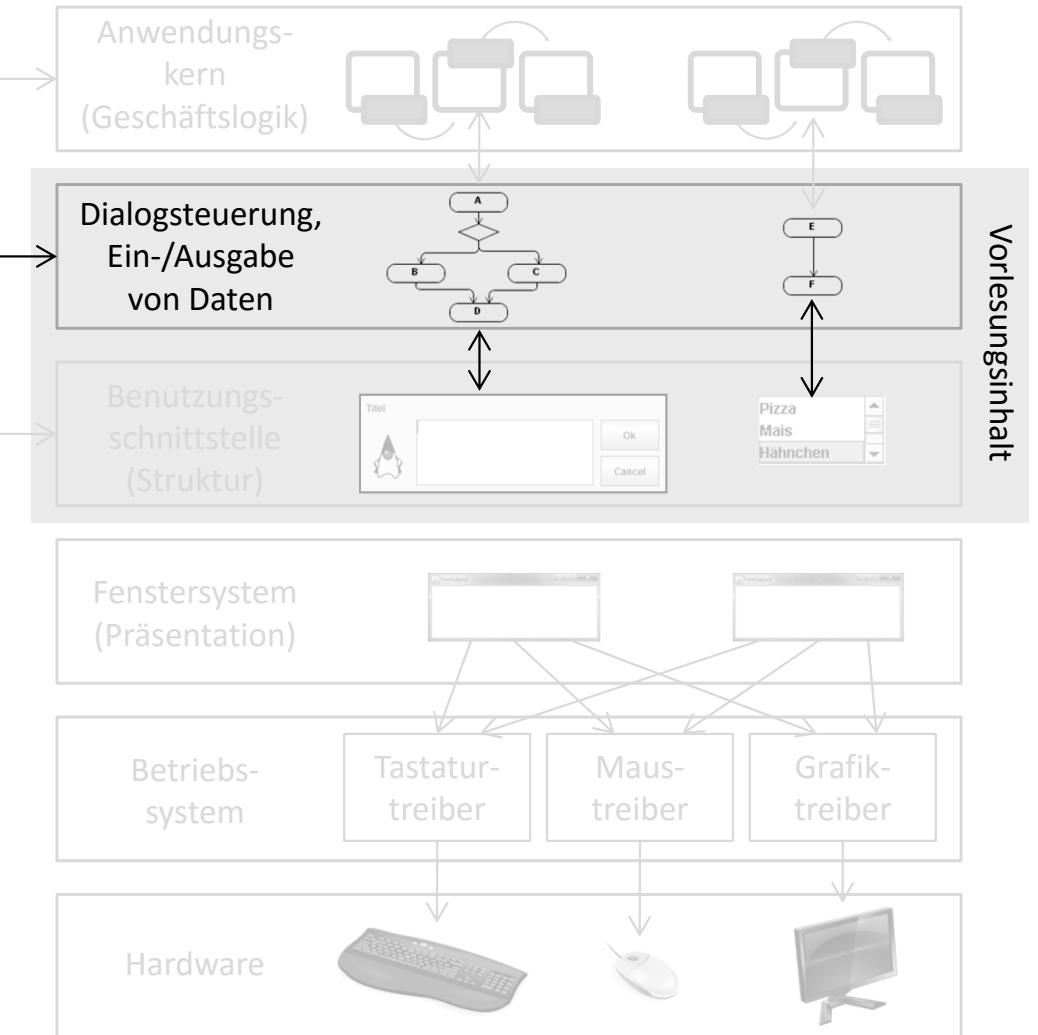
Übersicht



die eigentliche Funktionalität,
von der Dialogsteuerung aufgerufen

kontrolliert Aktionen des Benutzers auf
Plausibilität, steuert die Benutzungs-
schnittstelle und die Geschäftslogik,
synchronisiert Daten in der Oberfläche
und Geschäftslogik

verantwortlich für die Darstellung und
Verwaltung des Inhalts in den Fenstern





- Die Unterteilung in Ein- und Ausgabe von Daten sowie die Dialogsteuerung in den beiden folgenden Abschnitten ist nicht ganz konsistent. Gemeint ist:
 - ◆ Ein- und Ausgabe von Daten: Synchronisation der Daten der Oberfläche mit einem Modell der Anwendung. Ein Synchronisationsereignis kann dabei auch eine Aktion in der Dialogsteuerung bewirken.
 - ◆ Dialogsteuerung:
 - Steuerung wesentlicher Funktionen der Geschäftslogik durch den Anwender bzw. die Oberfläche (Druck auf eine Taste, Menüauswahl, Synchronisationsereignis, Zeitereignis, ...)
 - Änderungen der Oberfläche durch Befehle des Geschäftslogik



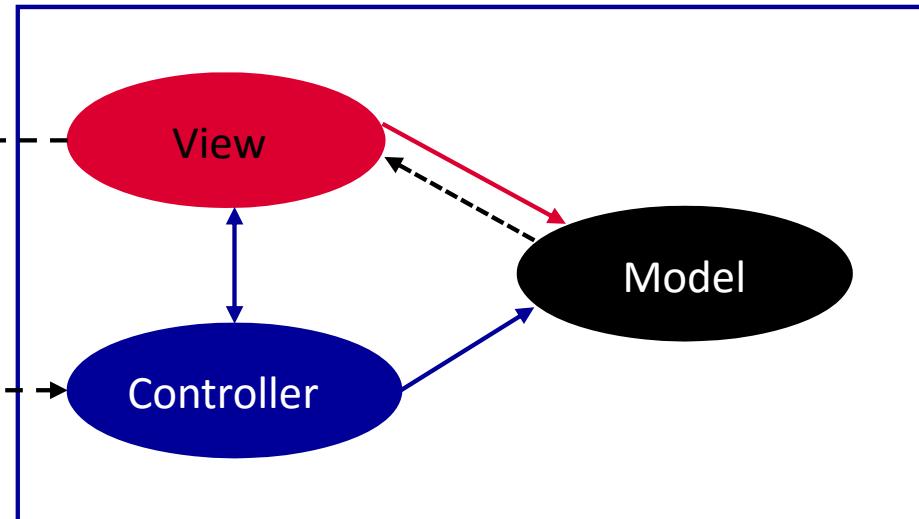
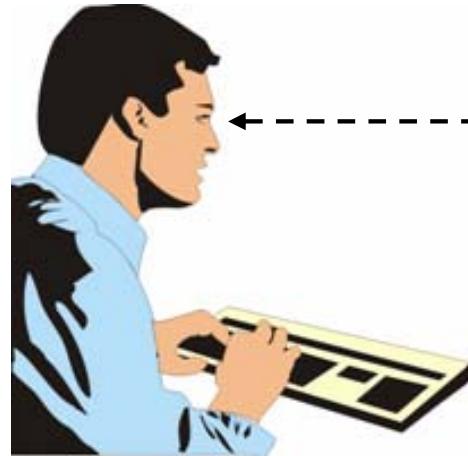
- Bisher wurden die Geschäftslogik und die Oberflächen nicht voneinander getrennt.
- Die Geschäftslogik greift direkt auf die Widgets zu.
- Das Resultat ist eine schlechte Entkopplung:
 - ◆ Eine parallele Konstruktion (durch verschiedene Entwickler) von Anwendungslogik und Oberfläche ist erschwert.
 - ◆ Das unabhängige Testen der Oberfläche bzw. Anwendungslogik ist kaum möglich.
 - ◆ Ein Austausch der Oberfläche bzw. Logik ist extrem aufwändig.
- Ziel dieses Kapitels: Trennung der Daten von Anwendungslogik und Oberfläche durch das so genannte „Data-Binding“.

Anbindung an die Geschäftslogik



Ein- und Ausgabe

- Zur Erinnerung: Struktur des MVC-Ansatzes



- Dialoge dienen dazu,
 - ◆ Objekt-Attribute mit Daten aus Widgets zu füllen
 - ◆ und Objekt-Attribute in Widgets darzustellen.
- Die Kernanwendung interessiert nur die Objekt-Attribute, nicht deren Darstellung.



- Es muss eine Instanz geben, die Objekt-Attribute und Widget-Werte synchronisiert:



- Analog zu einem O/R-Mapper:
 - ◆ Die Kernanwendung sieht die Objekte mit ihren Daten.
 - ◆ Woher die Daten kommen und wie sie abgelegt werden, interessiert (in der Regel) nicht.



- Aufgaben des Controllers (der „Bindung“):
 1. Beobachten der Datenänderungen
 2. Konvertierungen der Daten:
 - Modell zu Darstellung (z.B. **int → String**)
 - Darstellung zu Modell (z.B. **String → int**)
 3. Eingabekontrolle:
 - Ungültige Daten aus den Widgets dürfen nicht in das Modell eingetragen werden.
 - Es müssen Fehlerbeschreibungen generiert werden.
 4. Unterstützung zusammengesetzter Datentypen, Beispiel:
 - Im Model befinden sich Vor- und Nachname des Kunden.
 - In einem Widget werden beide zusammengesetzt angezeigt.
 - Ändert sich eine der beiden Modell-Zeichenketten, dann muss das Widget aktualisiert werden.
 5. Master/Detail-Beziehungen

Anbindung an die Geschäftslogik

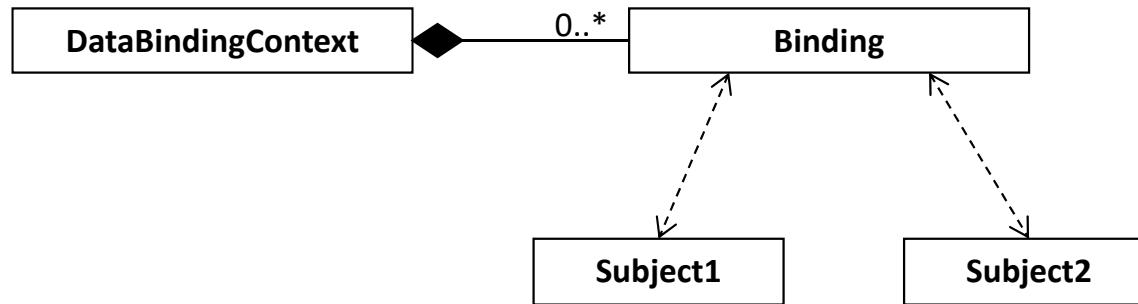
Ein- und Ausgabe



6. Thread-Sicherheit: Ein Objekt-Attribut wird von einem beliebigen Thread geändert.
7. Steuerbarkeit der Synchronisationsstrategie: Wann und in welcher Art und Weise soll synchronisiert werden?
 - Immer synchronisieren
 - Niemals synchronisieren
 - Nach Aufforderung prüfen und synchronisieren
 - Nur Eingabe prüfen



- Klassen für Bindungen:



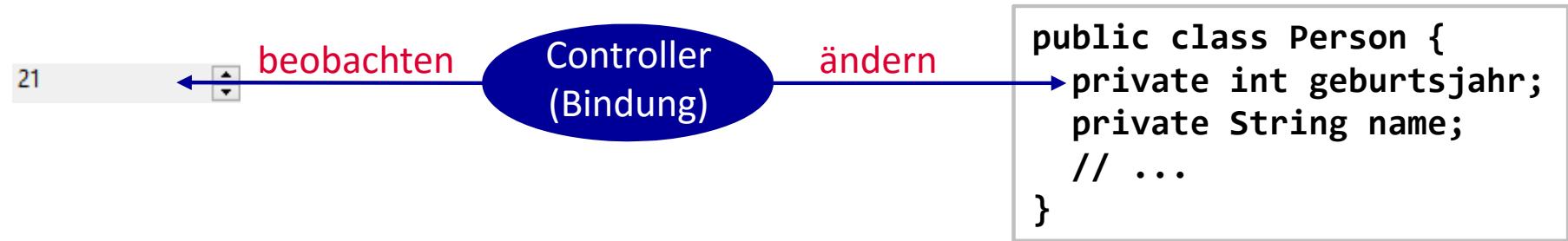
- ◆ Alle zusammengehörigen Bindungen (z.B. aus einem Dialog) werden in einem gemeinsamen Kontext gespeichert.
- ◆ Eine Bindung beobachtet die Änderungen an zwei Objekten und synchronisiert diese.
- ◆ **Subject1** und **Subject2** sind entweder **IObservable**-Klassen oder Java-Beans.



- Die zu überwachenden Klassen des Modells müssen der Java-Beans-Spezifikation unterliegen:
 - ◆ ein parameterloser Konstruktor
 - ◆ Getter- und Setter-Methoden für Attribute, die sich am besten an den Standard zur Namensvergabe halten.
 - ◆ Beispiel:
 - Attribut: **private int age;**
 - Getter: **public int getAge()**
 - Setter: **public void setAge(int age)**
 - Name der Eigenschaft („property“): **age**
 - ◆ Ausnahme **boolean**-Attribut:
 - Attribut: **private boolean retired;**
 - Getter: **public boolean isRetired()**
 - Setter: **public void setRetired(boolean retired)**
 - Name der Eigenschaft („property“): **retired**



- Sollen lediglich die Modelldaten in Widgets geschrieben werden, dann dürfen die Modellobjekte POJOs (Plain Old Java Objects) sein. Die umgekehrte Richtung ist dann nicht möglich.



- Für eine bidirektionale Synchronisation müssen die Modell-Objekte nach jeder überwachbaren Attributänderung ein **PropertyChangeEvent** auslösen:
 - Java unterstützt die Verwaltung der **PropertyChangeListener** durch eine Anzahl von Klassen und Schnittstellen.
 - Die Setter-Methoden müssen einem einheitlichen Aufbau folgen.
- Anmerkung: Es gibt auch eine Lösung ohne die Ereignisse. Dann müssen die Attribute alle Objekte einer der Klassen **WritableValue**, **WritableList**, ... sein → soll hier nicht betrachtet werden.



- Erstellen einer Modell-Klasse, die **PropertyChangeListener** verwaltet und **PropertyChangeEvent**s auslöst:
 - Das Modell muss die Beobachter verwalten (das Abmelden als Beobachter fehlt im Beispiel). Dazu existiert die Klasse **PropertyChangeSupport**:

```
public class CustomerModel {  
    // Liste aller Beobachter am Modell  
    private transient PropertyChangeSupport support =  
        new PropertyChangeSupport(this);  
  
    // Beobachter registriert sich für die Änderungen aller Attribute.  
    public void addPropertyChangeListener(  
        PropertyChangeListener listener) {  
        support.addPropertyChangeListener(listener);  
    }  
  
    // Beobachter registriert sich nur für die Änderungen eines  
    // Attributes.  
    public void addPropertyChangeListener(String propertyName,  
        PropertyChangeListener listener) {  
        support.addPropertyChangeListener(propertyName, listener);  
    }  
}
```



- Aufbau der Setter-Methode, um die Beobachter zu informieren:

```
private int age;  
  
// Setter-Methode, um das Alter zu ändern.  
public void setAge(int age) {  
    support.firePropertyChange("age", this.age, this.age = age);  
    //           ^          ^          ^  
    //PropertyName alter Wert  neuer Wert  
}
```

- Besser: Property-Namen als Konstanten in der Klasse ablegen.
- Die Getter-Methode muss nicht angepasst werden.
- Für Array-Attribute gibt es in der Klasse **PropertyChangeSupport** auch eine Lösung.
- Für allgemeine Bean-Klassen gibt es auch noch **VetoableChangeListener**. Im Gegensatz zu einem **PropertyChangeListener** kann jeder Beobachter ein Veto gegen eine Änderung einlegen → wird für das Data-Binding nicht benötigt.
- Resultat: Durch **PropertyChangeEvent**s informieren Modellklassen über Attributänderungen → saubere Entkopplung über Ereignisse.

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding



- Weiterer Vorteil dieses Modell-Ansatzes: Modell-Objekte können leicht serialisiert werden:
 - ◆ Zur Übertragung zwischen Client und Server
 - ◆ Zur Speicherung im Dateisystem
 - ◆ Als Basis für eine Persistenzlösung
- Die Modell-Klasse muss lediglich die leere Schnittstelle **Serializable** implementieren.

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding

- Modell für das Beispiel (Klasse **Customer**, Ausschnitt):

```
public class Customer {  
    // Möglicher Familienstand  
    public static enum FamilyStatus { UNKNOWN, MARRIED, UNMARRIED }  
  
    // Property-Namen  
    public static final String PROPERTY_FIRST_NAME      = "firstName";  
    public static final String PROPERTY_LAST_NAME       = "lastName";  
    public static final String PROPERTY_CUSTOMER_NUMBER = "customerNumber";  
    public static final String PROPERTY_RETIRED          = "retired";  
    public static final String PROPERTY_FAMILY_STATUS   = "familyStatus";  
    public static final String PROPERTY_CUSTOMER_SINCE  = "customerSince";  
  
    private String      firstName;  
    private String      lastName;  
    private int         customerNumber;  
    private boolean     retired;  
    private FamilyStatus familyStatus = FamilyStatus.UNKNOWN;  
    private Date        customerSince;  
  
    // Getter, Setter mit Auslösung von PropertyChangeEvents,  
    // Standardkonstruktor, PropertyChangeSupport  
}
```

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding



- Darstellung der Beispielanwendung:

Kundenstammdaten

Eingabe der Kundendaten
Geben Sie die Stammdaten des Kunden ein.

Vorname:
Nachname:
Kundennummer: Im Ruhestand:
Familienstand: Kunde seit:
OK

OK **Cancel** **Löschen**

Modell löschen



- Welche Anforderungen gibt es an die Darstellungsseite (Dialog mit den Widgets)?
 - ◆ Keine! Die Widgets können ja schon Ereignisse auslösen.
- Erzeugen des gemeinsamen Kontextes für die Bindungen:
`DataBindingContext context = new DataBindingContext();`

1. Beobachten der Datenänderungen

- Bindungen benötigen bei bidirektionalem Abgleich Beobachter auf „beiden Seiten“ (Widget und Attribut).
- Zur Erzeugung der Beobachter existieren Fabrikmethoden in den folgenden Klassen, die die Verwendung vereinfachen:
 - ◆ **WidgetProperties** für SWT-Widgets
 - ◆ **ViewersProperties** für JFace-Viewer-Klassen
 - ◆ **BeansProperties** für Attribute der Modell-Klasse

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding: Beobachtung von Änderungen

- Anbindung eines SWT-Textfeldes an das Vornamen-Attribut des Kunden:

Vorname:

```
Text firstNameText = new Text(parent, SWT.BORDER);

// Beobachteten Wert für das Textfeld erzeugen,
// eine Benachrichtigung soll nach jedem geänderten Zeichen
// erfolgen. SWT.Modify ist ein SWT-Ereignis-Typ.
// Um erst beim Verlassen des Feldes zu synchronisieren, würde
// SWT.FocusOut verwendet werden.
IObservableValue<String> stringWidgetObservable =
    WidgetProperties.text(SWT.Modify).observe(firstNameText);

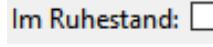
// Beobachteten Wert für das Vornamensattribut des Kunden-Modells
// erzeugen.
IObservableValue<String> stringPropertyObservable =
    BeansProperties.value(Customer.class,
        Customer.PROPERTY_FIRST_NAME).observe(customer);

// Bindung zwischen beiden beobachteten Werten herstellen.
context.bindValue(stringWidgetObservable, stringPropertyObservable);
```

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding: Beobachtung von Änderungen

- Anbindung einer SWT-CheckBox an das Ruhestands-Attribut (Typ **boolean**) des Kunden: 

```
retiredButton = new Button(contents, SWT.CHECK);

// Bei jeder Änderung der Selektion wird synchronisiert.
IObservableValue<boolean> booleanWidgetObservable =
    WidgetProperties.selection().observe(retiredButton);

IObservableValue<boolean> booleanPropertyObservable =
    BeanProperties.value(Customer.class,
        Customer.PROPERTY_RETIRE).observe(customer);
context.bindValue(booleanWidgetObservable, booleanPropertyObservable);
```

- Für Widgets gibt es viele beobachtbare Eigenschaften:
 - Farben, Zeichensatz, Größe und Position, Sichtbarkeit, Freigabezustand (enabled/disabled)
 - Tasturfokus, minimaler, maximaler und aktueller Wert (z.B. bei einem Schieberegler), Text, Selektion
 - Einträge (z.B. in einer Liste)
- Nicht alle Eigenschaften können bei allen Widget-Typen beobachtet werden.



2. Konvertierungen der Daten

- Nicht immer sind der Datentyp im Widget und im Modell identisch, Beispiele:
 - ◆ Im Textfeld wird als Alter ein **String** eingegeben. Im Modell befindet sich ein **int**-Wert.
 - ◆ Im Textfeld wird für ein Datum ein **String** eingegeben. Im Modell befindet sich ein **Date**-Wert.
 - ◆ In einer Combo-Box wird ein Wert selektiert. Das Ergebnis ist ein **int**-Index. Das Modell erwartet aber einen Aufzähltyp, weil die Combo-Box eine Auswahl aus einer Menge von Werten anbietet.
- Einfach Konvertierungen werden automatisch vorgenommen:
 - ◆ zwischen **String** und primitiven Datentypen
 - ◆ zwischen **String** und **Date**:
 - Beim Parsen werden alle möglichen Formatierungen durchprobiert.
 - Zur Ausgabe wird ein vordefiniertes Format verwendet → häufig nicht hilfreich.

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding: Konvertierungen der Daten

- Wie werden eigene Konvertierungen vorgenommen?
 - ◆ Übergabe eines Konverters an die Bindung (Schnittstelle **IConverter** oder abstrakte Klasse **Converter**):
 - ◆ Für jede Richtung (Widget → Modell-Attribut und Modell-Attribut → Widget) kann ein eigener Konverter angegeben werden.
 - ◆ Die Klasse **UpdateValueStrategy** kapselt eine Konvertierung zusammen mit optionalen Validierungen der Werte (kommt gleich).
- Beispiel für das Umwandeln eines **int**-Indexes in einen **enum**-Wert (Auswahl des Familienstandes durch einen **ComboViewer**):

The screenshot shows two states of a JComboBox. On the left, the dropdown is closed, displaying the text "unbekannt". On the right, the dropdown is open, showing a list of items: "unbekannt", "verheiratet", and "unverheiratet". The item "unbekannt" is highlighted with a blue selection bar, indicating it has been selected.

- Und Beispiel für eine Umwandlung zwischen **String** und Datum (**Date**):

The screenshot shows a JTextField containing the text "24.09.15". This represents a string date being converted into a Date object for storage or display.

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding: Konvertierungen der Daten

```
// Combo-Box erzeugen und füllen.  
familyStatusCombo = new ComboViewer(contents, SWT.READ_ONLY);  
familyStatusCombo.setContentProvider(ArrayContentProvider.getInstance());  
familyStatusCombo.setInput(Customer.FamilyStatus.values());  
// ...  
familyStatus.getCombo().select(0);  
  
// Familienstand bei Änderung des selektierten Eintrags übernehmen.  
IViewerObservableValue<Viewer> viewerWidgetObservable =  
    ViewerProperties.singleSelection().observe(familyStatusCombo);  
IObservableValue<Customer.FamilyStatus> enumPropertyObservable =  
    BeanProperties.value(Customer.class,  
        Customer.PROPERTY_FAMILY_STATUS).observe(customer);  
  
// Konvertierung funktioniert bei JFace automatisch.  
// Validierung ist nicht erforderlich, da keine Fehleingaben möglich sind.  
  
context.bindValue(viewerWidgetObservable, enumPropertyObservable);
```



Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding: Konvertierungen der Daten

```
// Kunde seit...
stringWidgetObservable = WidgetProperties.text(SWT.FocusOut)
    .observe(customerSinceText);
stringPropertyObservable = BeanProperties.value(Customer.class,
    Customer.PROPERTY_CUSTOMER_SINCE).observe(customer);

// Konvertierung des eingegebenen Strings in ein Datums-Objekt
UpdateValueStrategy widgetToModelUpdater = new UpdateValueStrategy();
StringToDateConverter widgetToModelConverter
    = new StringToDateConverter(dateFormat);
widgetToModelUpdater.setConverter(widgetToModelConverter);

// Konvertierung des Datums aus dem Modell in einen String
DateToStringConverter dateToStringConverter
    = new DateToStringConverter(dateFormat);
UpdateValueStrategy modelToWidgetUpdater = new UpdateValueStrategy();
modelToWidgetUpdater.setConverter(dateToStringConverter);

context.bindValue(stringWidgetObservable, stringPropertyObservable,
    widgetToModelUpdater, modelToWidgetUpdater);
```



- Aufbau des Konverters für ein Datums-Objekt in einen String:

```
public class DateToStringConverter extends Converter {  
    private DateFormat dateFormat;  
  
    public DateToStringConverter(DateFormat dateFormat) {  
        super(Date.class, String.class);  
        this.dateFormat = dateFormat;  
    }  
  
    @Override  
    public Object convert(Object value) {  
        if (value == null) {  
            return "";  
        }  
        return dateFormat.format((Date) value);  
    }  
}
```



3. Eingabekontrollen

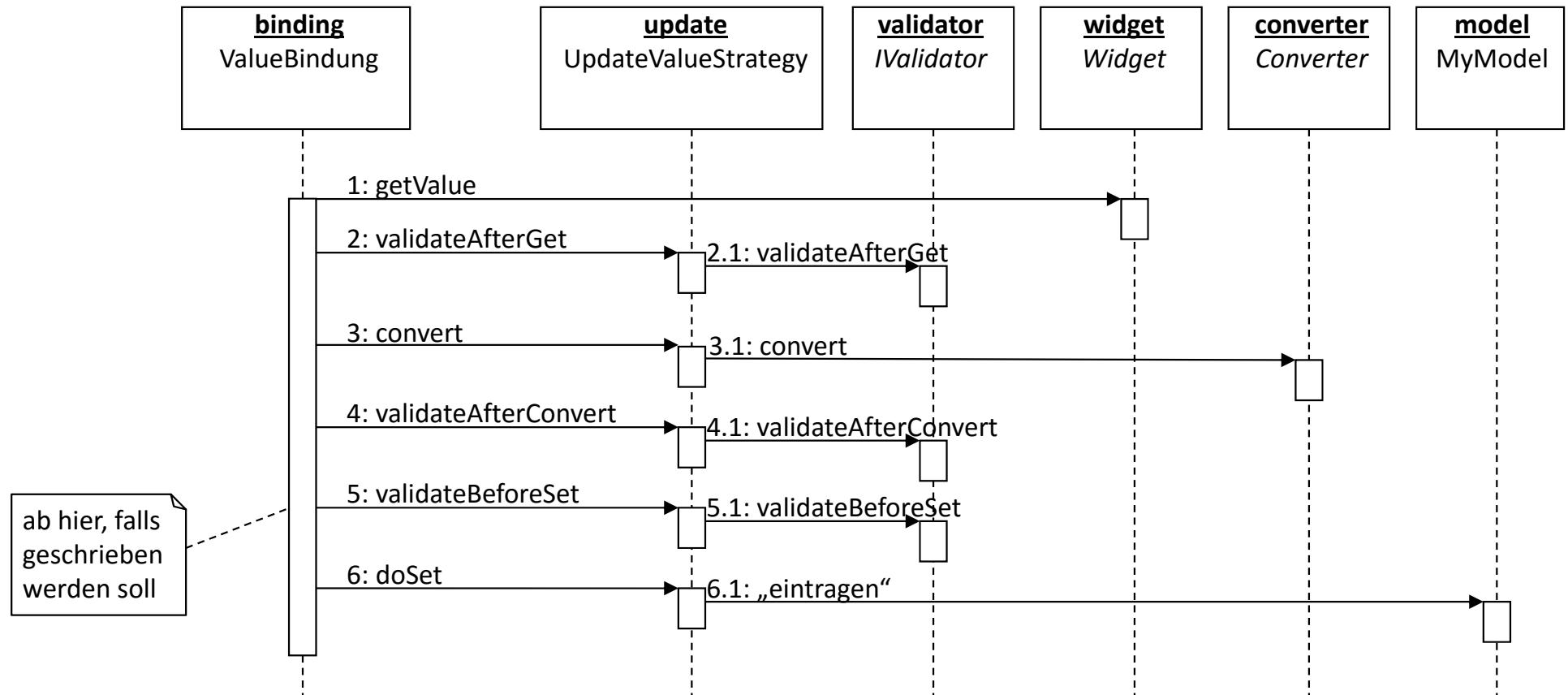
- Aufgaben:
 - ◆ Ungültige Werte dürfen nicht in das Ziel eingetragen werden.
 - ◆ Falsche Eingaben müssen dem Benutzer angezeigt werden.
- Schritte, bis ein Wert von der Quelle in das Ziel übertragen wurde:
 1. Wert aus der Quelle auslesen
 2. Prüfung des ausgelesenen Wertes (Methode **validateAfterGet** des Validierers), häufig für syntaktische Tests
 3. Konvertierung in das neue Format (Methode **convert** des Konverters)
 4. Prüfung des konvertierten Wertes (Methode **validateAfterConvert** des Validierers), häufig für Tests auf Wertebereiche
 5. Prüfung des konvertierten Wertes direkt vor dem Eintragen in das Ziel (Methode **validateBeforeSet** des Validierers), falls geschrieben wird
 6. Eintragen des Wertes in das Ziel
- Schlägt eine Validierung fehl, dann wird die Bearbeitungskette abgebrochen.

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding: Eingabeverifikationen

- Ablauf bei der kompletten Synchronisation (vereinfacht, beispielhaft):





- Erstellen eines Validierers:
 - ◆ Implementieren der Schnittstelle **IValidator**
 - ◆ Implementieren der Methode **public IStatus validate(Object value)**
 - ◆ Die Rückgabewerte der Methode können mit Hilfe einer statischen Methode der Klasse **ValidationStatus** erzeugt werden. Jede Meldung hat einen Schweregrad, anhand dessen später die Ausgaben gefiltert werden können.
 - 0: **ValidationStatus.ok()**, kein Fehler aufgetreten
 - 1: **ValidationStatus.info()**, Hinweis oder Hilfestellung an den Benutzer
 - 2: **ValidationStatus.warning()**, nur eine Warnung, die ignoriert werden kann
 - 4: **ValidationStatus.error()**, Fehler aufgetreten
 - 8: **ValidationStatus.cancel()**, Abbruch der gestarteten Aktion

Anbindung an die Geschäftslogik



Ein- und Ausgabe mit Databinding: Eingabeverifikationen

- Kundenbeispiel:
 - ◆ Die Übernahme eines ungültigen Datums in das Modell soll verhindert werden.
 - ◆ Ein Fehler wird durch ein Symbol direkt am Widget angezeigt.
 - ◆ Der Dialog enthält im unteren Abschnitt eine Aufzählung aller Eingabefehler.

Kundenstammdaten

Eingabe der Kundendaten

Geben Sie die Stammdaten des Kunden ein.

Vorname:	Holger		
Nachname:	Vogelsang		
Kundennummer:	1234	Im Ruhestand:	<input type="checkbox"/>
Familienstand:	unbekannt	Kunde seit:	eee

Ungültiges Datum

OK Cancel Löschen

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding: Eingabeverifikationen



- Quelltext, etwas länglich:

```
Text customerSinceText = new Text(contents, SWT.BORDER);

// "dekorieren", damit ein Fehlersymbol am Textfeld ein- und ausgeblendet
// werden kann.
customerSinceDecorator = createDecorator(customerSinceText,
                                         "Ung\u00fcltiges Datum");

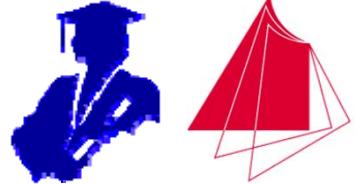
// Text\u00e4nderung beim Verlassen des Textfeldes melden.
stringWidgetObservable = WidgetProperties.text(SWT.FocusOut)
                           .observe(customerSinceText);

// Modell\u00e4nderung beobachten
stringPropertyObservable = BeanProperties.value(Customer.class,
                                                 Customer.PROPERTY_CUSTOMER_SINCE).observe(customer);

// UpdateStrategy-Objekt bef\u00fcllen, zuerst den Konverter eintragen
// (eigene Konvertierungsklasse).
widgetToModelUpdater = new UpdateValueStrategy();
StringToDateConverter widgetToModelConverter =
    new StringToDateConverter(dateFormat);
widgetToModelUpdater.setConverter(widgetToModelConverter);
```

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding: Eingabeverifikationen



```
// Validierer erzeugen: Eigene Klasse, die den Konvertierer verwendet,  
// um auf ungültige Eingaben zu prüfen. Der Konvertierer meldet in  
// diesem Fall einen Fehler durch Auslösen einer Ausnahme.  
// Leere Datumsangaben sind zulässig (true).  
StringToDateValidator stringToDateValidator =  
    new StringToDateValidator(widgetToModelConverter, true, null,  
        "Ung\u00fcrliges Datum");  
stringToDateValidator.setControlDecoration(customerSinceDecorator);  
widgetToModelUpdater.setAfterGetValidator(stringToDateValidator);  
  
// Der Rückweg vom Modell zum Widget funktioniert genauso.
```



- Aufbau des Validierers für die Umwandlung eines String-Objektes in ein Datums-Objekt (Klasse **StringToDateValidator**, hier ohne das Ein- und Ausblenden des Fehlersymbols des Dekorierers):

```
public class StringToDateValidator implements IValidator {  
    private StringToDateConverter converter;  
    private boolean allowEmptyDate;  
    private String emptyDateMessage;  
    private String illegalDateMessage;  
  
    public StringToDateValidator(StringToDateConverter converter,  
                                boolean allowEmptyDate,  
                                String emptyDateMessage,  
                                String illegalDateMessage) {  
        this.converter = converter;  
        this.allowEmptyDate = allowEmptyDate;  
        this.emptyDateMessage = emptyDateMessage;  
        this非法DateMessage = illegalDateMessage;  
    }  
}
```

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding: Eingabevalidierungen



```
// Test darauf, ob das Datum (Übergabeparameter value) gültig ist.  
@Override  
public IStatus validate(Object value) {  
    boolean ok = false;  
    boolean empty = false;  
    try {  
        // Versuche, zu konvertieren  
        empty = converter.convert(value) == null;  
        ok = !empty || allowEmptyDate;  
    }  
    // fehlgeschlagen, ok bleibt false  
    catch (IllegalArgumentException ex) {  
    }  
  
    if (ok) {  
        return ValidationStatus.ok();  
    }  
  
    return empty ? ValidationStatus.error(emptyDateMessage) :  
        ValidationStatus.error(illegalDateMessage);  
}  
}  
}
```



- Ausgabe des schwersten Fehlers eines Dialogs:

```
Label errorLabel = new Label(contents, SWT.NONE);
errorLabel.setText("");

// Alle Fehlermeldungen aggregieren und den schlimmsten Fehler im
// Fehler-Label ausgeben
IObservableValue<String> errorLabelObservable = WidgetProperties.text()
    .observe(errorLabel);

// AggregateValidationStatus ermittelt die Meldung mit dem höchsten
// Fehler und erzeugt daraus eine Zeichenketten, deren aktueller
// Inhalt an den Text des Labels gebunden wird. Mit Übergabe der
// Konstanten AggregateValidationStatus.MERGED würde eine Liste aller
// Fehler zurück gegeben werden.
context.bindValue(errorLabelObservable,
    new AggregateValidationStatus(context.getBindings(),
        AggregateValidationStatus.MAX_SEVERITY));
```



- Was passiert, wenn die Gültigkeit einer Eingabe von anderen Quellen abhängt?
 - ◆ Beispiel: Das Einstellungsdatum einer Person muss größer als sein Geburtsdatum sein.
 - ◆ Lösung: Die Klasse **MultiValidator** unterstützt genau solche Einsatzgebiete.
- Wie kann dem Anwender eine Hilfestellung bei der Eingabe gegeben werden?
 - ◆ Der **ControlDecorater** kann **ProposalProvider** verwalten.
 - ◆ Diese werden auf Tastendruck eingeblendet und erlauben die Auswahl eines Wertes aus einer vorgegebenen Anzahl (wie in der IDE bei der Methodenauswahl nach Eingabe eines Punktes nach einer Referenz).

The screenshot shows an IDE interface with code completion proposals for a 'Text' control. The proposals listed are:

- setText(String string) : void - Text - 70%
- addModifyListener(ModifyListener listener) : void -
- setEnabled(boolean enabled) : void - Control - 13%
- getText() : String - Text - 7%
- setFocus() : boolean - Control - 6%
- setBackground(Color color) : void - Control - 5%
- setFont(Font font) : void - Text - 5%
- handle : long - Control
- setLayoutData(Object layoutData) : void - Control -

Below the proposals, a tooltip provides detailed information about the `setText` method:

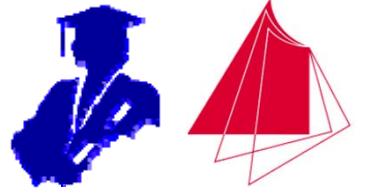
Sets the contents of the receiver to the given string. If the receiver has style SINGLE and the argument contains multiple lines of text, the result of this operation is undefined and may vary from platform to platform.

Note: If control characters like '\n', '\t' etc. are used in the string, then the behavior is platform dependent.

Parameters:
string the new text

Throws:
[IllegalArgumentException](#) -

Press 'Tab' from proposal table or click for focus



4. Unterstützung zusammengesetzter Datentypen

- Aufgabe:
 - ◆ Automatisches Zusammensetzen mehrerer einzelner Quelldaten zu einem Zieldatenwert
 - ◆ Soll hier nicht näher betrachtet werden → siehe Klasse **ComputedValue**

5. Master/Detail-Beziehungen

- Aufgabe:
 - ◆ In einer Tabelle wird eine Zeile selektiert (Master).
 - ◆ Der Inhalt der Tabellenzeile wird aus dem Modell in einen Dialog kopiert und dort angezeigt.
 - ◆ Alle Dialogeingaben werden mit der Modellzeile synchronisiert.
 - ◆ → soll hier nicht betrachtet werden



6. Thread-Sicherheit

- Problem:
 - ◆ Widgets werden im UI-Thread gefüllt → Das Schreiben in das Modell erfolgt somit auch im UI-Thread.
 - ◆ Modelle können durch die Kernanwendung in einem anderen Thread gefüllt werden.
 - ◆ Wie soll hier Thread-Sicherheit hergestellt werden?
- Lösung:
 - ◆ Die Bindungen stellen selbst eine Thread-Sicherheit her.
 - ◆ Dazu wurde das Konzept des „Realms“ (Königreich, Bereich) eingeführt:
 - Beobachtbare Werte werden einem Bereich zugeordnet.
 - Es handelt sich um einen Thread (oder seltener eine Sperre), der für den serialisierten Zugriff auf die Werte zuständig ist.
 - Also: Jeder Zugriff auf einen beobachteten Wert erfolgt innerhalb des Bereiches, zu dem er gehört.

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding: Multithreading



- Vereinfacht beschrieben: Alle Zugriffe aus anderen Threads werden durch **realm.asyncExec(Runnable run)** im selben Thread ausgeführt.
- Die Standardimplementierung verwendet intern **Display.asyncExec(Runnable run)**.
- ◆ Was passiert beim Aufruf einer Setter-Methode des Modells in einem anderen Thread?
 - Die **PropertyChangeEvent**s werden in dem fremden Thread ausgeführt.
 - Die Änderungen an den beobachteten Werten erfolgt aber automatisch im richtigen Bereich.
- Bei RCP-Anwendungen erhält jedes Plug-in automatisch seinen Bereich.

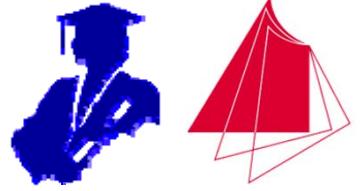


7. Synchronisationsstrategien

- Aufgabe:
 - ◆ Wann wird eine Synchronisation durchgeführt?
 - ◆ In gewissen Grenzen konnte das schon durch die Art des Ereignisses (**SWT.Modify**, **SWT.FocusOut**) gesteuert werden.
- Die **UpdateValueStrategy** (gilt genauso für **UpdateListStrategy** und **UpdateSetStrategy**) einer Bindung kann auch selbst anhand von Regeln entscheiden, wann (und ob überhaupt) synchronisiert wird. Dazu kann sie eine „Policy“ (Strategie) übergeben bekommen:
 - ◆ **POLICY_UPDATE**: Standardeinstellung, in der immer automatisch synchronisiert wird.
 - ◆ **POLICY_NEVER**: Änderungen am Quellobjekt werden ignoriert und das Ziel nie aktualisiert. So kann eine Bindung unidirektional ausgelegt werden (eine Strategie der Bindung synchronisiert nie, die andere dagegen schon). Das kann auch erforderlich sein, wenn ein Widget keine Ereignisse unterstützt und lediglich Daten anzeigen soll.



- ◆ **POLICY_ON_REQUEST**: Änderungen am Quellobjekt werden ignoriert und das Ziel nur dann aktualisiert, wenn eine der beiden Methoden
 - **updateModels()**: Widget-Werte in das Modell eintragen
 - **updateTargets()**: Modell-Werte in die Widgets eintragenauf dem **DataBindingContext** aufgerufen wird. Einsatzgebiete:
 - Ein Dialog prüft spät und schreibt erst beim Klick auf die OK-Taste mit **updateModels()** in das Modell.
 - Ein Dialog macht Änderungen mit **updateTargets()** rückgängig, indem beim Druck auf die Abbruch-Taste die Widget-Werte durch die Werte des Modells überschrieben werden.
- ◆ **POLICY_CONVERT**: Das Quellobjekt wird immer beobachtet, die Validierungen und Konvertierungen ausgeführt. Die Synchronisation erfolgt wie bei **POLICY_ON_REQUEST** manuell durch Aufrufe von **updateModels()** bzw. **updateTargets()**. Einsatzgebiete:
 - Die Eingaben werden ständig geprüft und Fehler angezeigt.
 - Änderungen werden erst z.B. beim Druck auf OK übernommen.



- Verwendung der Strategie durch Übergabe im Konstruktor (zur Erinnerung: die **UpdateValueStrategy** fasst auch die Konvertierungen und Validierungen zusammen für eine Synchronisationsrichtung):

```
UpdateValueStrategy widgetToModel =  
    new UpdateValueStrategy(UpdateValueStrategy.POLICY_NEVER);
```

Weitere Informationen unter http://wiki.eclipse.org/index.php/JFace_Data_Binding

Anbindung an die Geschäftslogik

Ein- und Ausgabe mit Databinding: Synchronisationsstrategien



- Beispiel aus der Bonusaufgabe:

Proxy-Einstellungen

Proxy-Einstellungen für den Netzwerkzugriff

ⓘ Innerhalb der Hochschule muss der Proxy-Server verwendet werden.

HTTP-Proxy: verwenden

Proxyname:

Proxyport:

Benutzername:

Passwort:

OK Cancel

Je nach Selektionsstatus der Taste (Checkbox) werden die Textfelder sowie deren Label gesperrt oder freigegeben,

Proxy-Einstellungen

Proxy-Einstellungen für den Netzwerkzugriff

ⓘ Innerhalb der Hochschule muss der Proxy-Server verwendet werden.

HTTP-Proxy: verwenden

Proxyname:

Proxyport:

Benutzername:

Passwort:

OK Cancel



- Variante 1 (ohne Data-Binding):

```
Button button = new Button(contents, SWT.CHECK);
button.setText(Messages.ConfigurationDialog_Apply);
button.setSelection(true);
builder.add(button, cc.xy(4, 2));
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        boolean enabled = (((Button) event.widget).getSelection());
        proxyNameLabel.setEnabled(enabled);
        proxyNameText.setEnabled(enabled);
        proxyPortLabel.setEnabled(enabled);
        proxyPortText.setEnabled(enabled);
        userNameLabel.setEnabled(enabled);
        userNameText.setEnabled(enabled);
        passwortLabel.setEnabled(enabled);
        passwortText.setEnabled(enabled);
    }
});
```

- Wichtig: Der Initialzustand muss stimmen → Die Taste ist selektiert und alle Felder sind freigegeben.
- Problem: Bei programmgesteuertem Ändern der Taste tritt kein Ereignis auf.



- Variante 2 mit bidirektonaler Synchronisation zwischen Selektions- und Freigabezustand (unidirektonal wäre hier etwas eleganter → siehe Eclipse-Projekt auf der Homepage):

```
private void createBidirectionalBindings() {  
    context = new DataBindingContext();  
    createSingleBinding(proxyButton, new Control[]{ proxyNameLabel,  
        proxyNameText, proxyPortLabel, proxyPortText,  
        userNameLabel, userNameText, passwortLabel, passwortText} );  
}  
  
private void createSingleBinding(Button source, Control[] destinations) {  
    IobservableValue<Boolean> sourceObserv =  
        WidgetProperties.selection().observe(source);  
  
    for (Control dest: destinations) {  
        IobservableValue<Boolean> destObserv =  
            WidgetProperties.enabled().observe(dest);  
        context.bindValue(destObserv, sourceObserv);  
    }  
}
```

- Bei programmgesteuertem Ändern der Taste tritt immer noch kein Ereignis auf.



- Inhalt des Kapitels:
 - ◆ Wie können entkoppelte Objekte überhaupt miteinander kommunizieren?
 - ◆ Wie kann die Oberfläche Aktionen in der Geschäftslogik auslösen?
 - ◆ Wie kann die Geschäftslogik die Oberfläche ändern?
 - ◆ Wie können diese beiden Benachrichtigungen der Schichten untereinander sauber entkoppelt werden?
 - ◆ Statt „Dialogsteuerung“ wird auch der Begriff „Ablaufsteuerung“ verwendet.



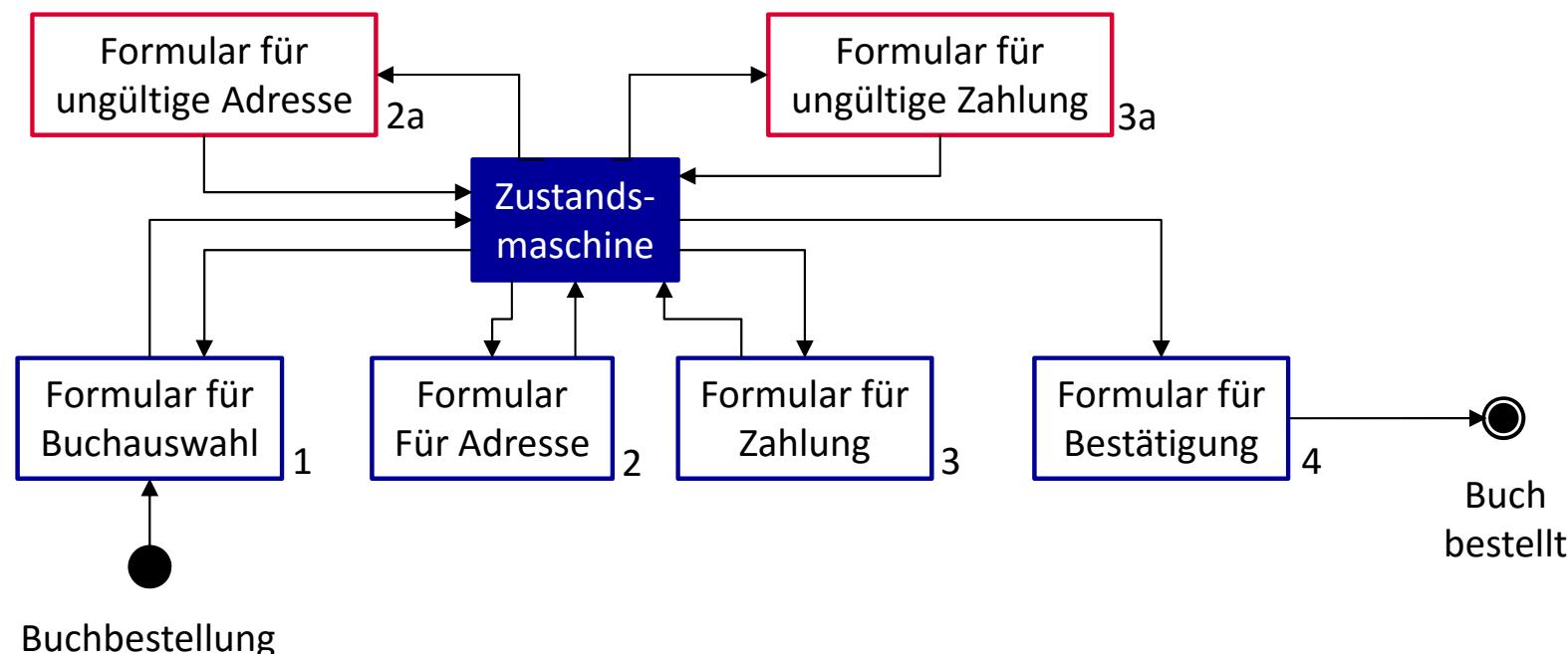
- Bisherige Lösung durch das Beobachtermuster:
 - ◆ Es entkoppelt Quelle und Ziel einer Nachricht durch „Broadcast“ der Nachricht.
 - ◆ Es erlaubt den Austausch von Quelle und Ziel. Beide müssen sich nicht direkt kennen.
 - ◆ Der Code ist leicht wiederverwendbar.
 - ◆ Es ist auf vielen Gebieten einsetzbar und leicht zu verstehen.
- Problem gelöst?
 - ◆ Nein: Es ist teilweise schwierig, den kompletten Kontrollfluss im Code zu erkennen → nur durch Simulation (Ablauf des Programms) nachvollziehbar.

Anbindung an die Geschäftslogik

Dialogsteuerung durch einen Zustandsautomaten



- In formularbasierten Anwendungen sind sehr leicht Zustände identifizierbar:
 - Wenn Formular 1 korrekt ausgefüllt wurde, dann soll Formular 2 angezeigt werden.
 - Bei einer fehlerhaften Eingabe bleibt Formular 1 angezeigt, und es werden Meldungen eingeblendet.
 - ...



Anbindung an die Geschäftslogik

Dialogsteuerung durch einen Zustandsautomaten



- Implementierung des Automaten:
 - ◆ durch eine Zustandstabelle → wird schnell unübersichtlich
 - ◆ „manuell“ durch Programmcode → häufig flexibler
- Anmerkung: Im MVC-Ansatz kann der Controller auch durch einen Automaten beschrieben werden.

Anbindung an die Geschäftslogik

Dialogsteuerung durch einen Ereignis-Vermittler



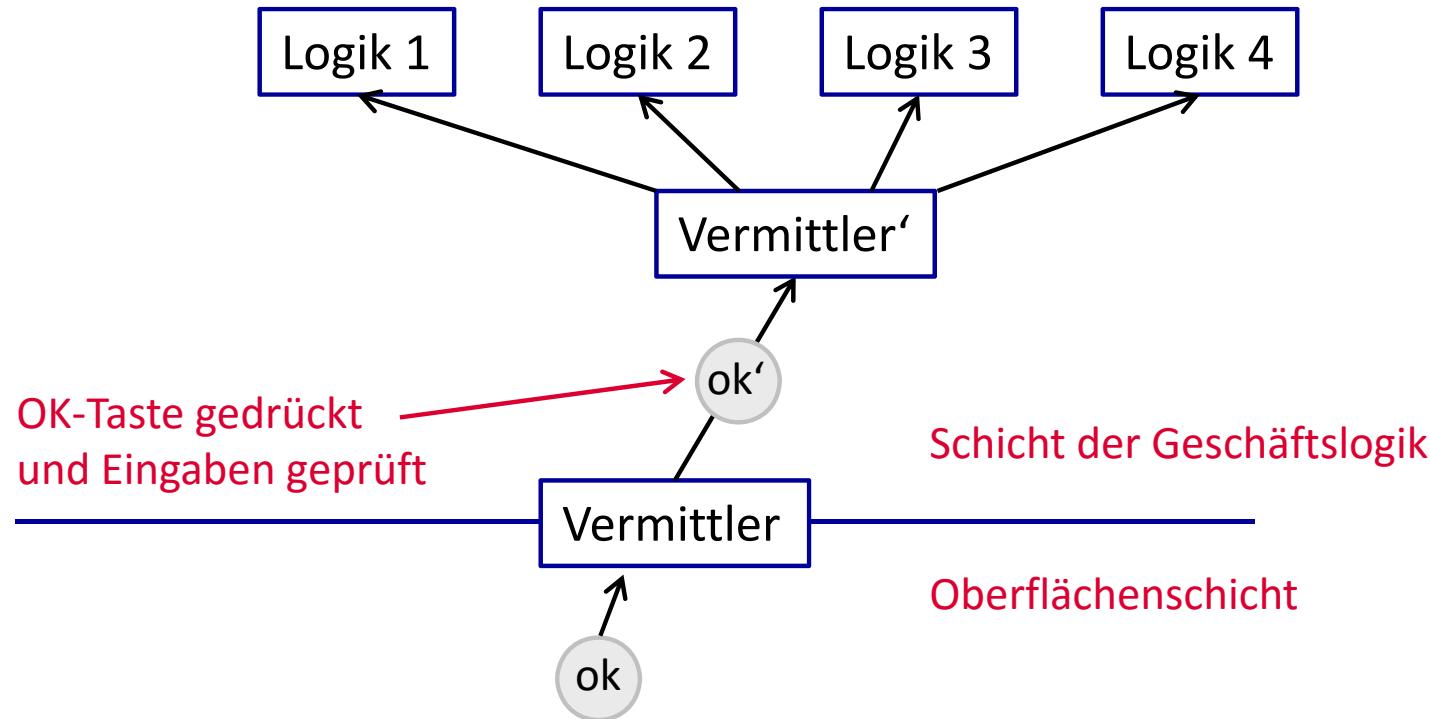
- Funktionsweise eines Ereignis-Vermittlers („Event Arbitrator“):
 - ◆ Er beobachtet viele verschiedene Quellen, indem er sich dort als Beobachter anmeldet.
 - ◆ Er leitet bestimmte Ereignisse an ihm bekannte Ziele weiter. Er kennt also die konkreten Ziele der Ereignisse.
 - ◆ Er kann Ereignisse manipulieren und weiterleiten.
 - ◆ Er kann Ereignisse in bestimmten Situationen unterdrücken (z.B. Debug-Ereignisse).
 - ◆ Er kann Ereignisse aggregieren und daraus andere Ereignisse auslösen (z.B. aus einer Folge von Ereignissen einer niederen Stufe ein Ereignis einer höheren Stufe erzeugen). Beispiel:
 - niedrige Stufe: OK-Taste gedrückt
 - höhere Stufe: OK-Taste gedrückt und alle Eingaben im Dialog gültig
 - ◆ Ziel: Zentralisierung der Ereignisbehandlung.



- Was hat das mit der Entkopplung von Geschäftslogik und Oberfläche zu tun?
- Ein besonderes Einsatzgebiet des Vermittlers:
 - ◆ Weitergabe von Ereignissen über Hierarchieebenen hinweg:
 - Er „übersetzt“ Oberflächenereignisse in Ereignisse für die Geschäftslogik (niedere Ebene → höhere Ebene).
 - Hier kennt er die Ziele der Ereignisse nicht. Die Ziele registrieren sich als Beobachter am Vermittler.
 - ◆ Der Vermittler ist der „Master“, die an ihm registrierten Beobachter sind die „Slaves“.
 - ◆ Wichtig: Zyklen sind nicht erlaubt → azyklischer Graph von Beobachtern.

Anbindung an die Geschäftslogik

Dialogsteuerung durch einen Ereignis-Vermittler



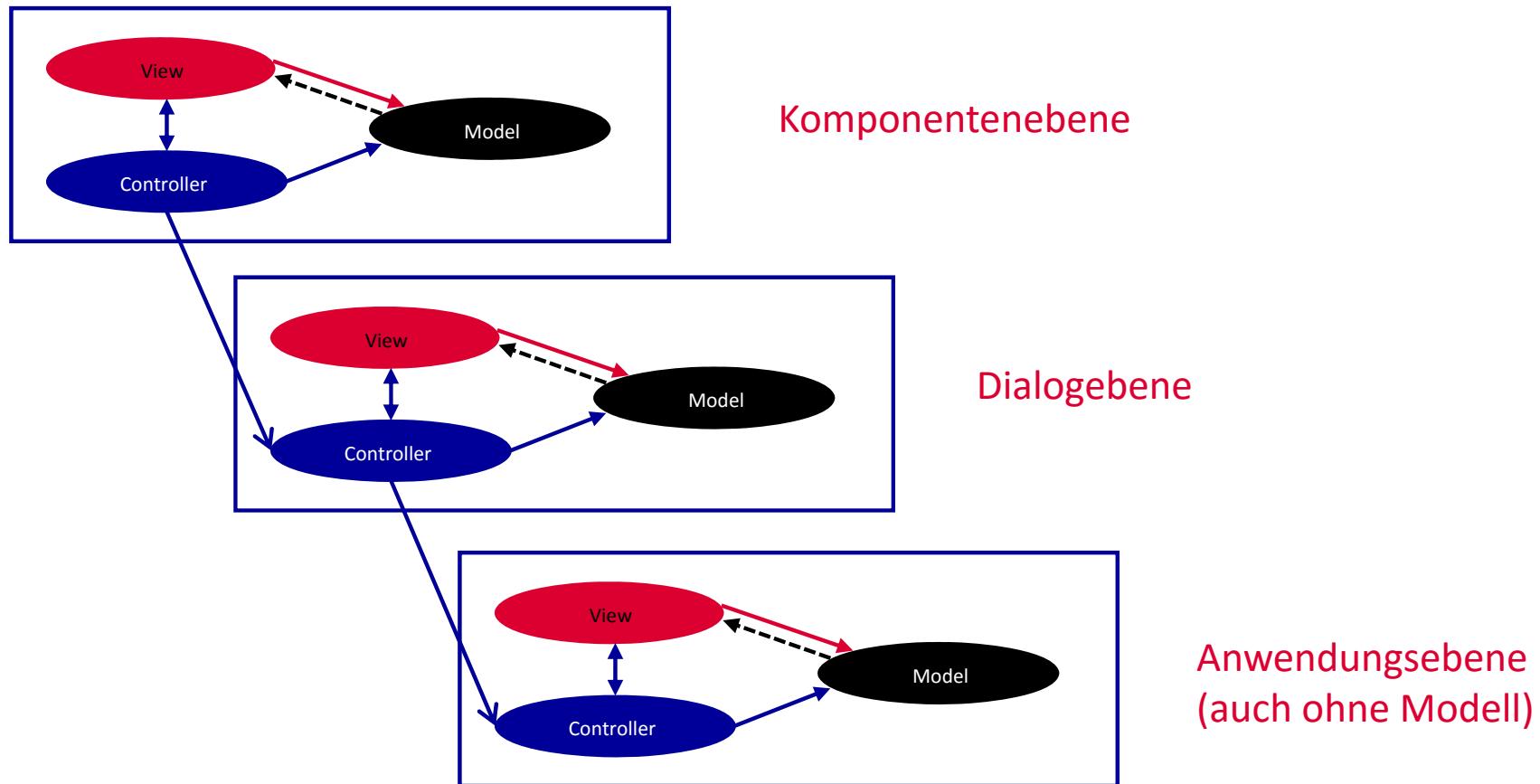
- Die Logik kann manuell ausprogrammiert oder z.B. durch Zustandsautomaten beschrieben werden.

Anbindung an die Geschäftslogik

Dialogsteuerung durch einen Ereignis-Vermittler



- Der Vermittler-Ansatz kann auch im H-MVC-Muster verwendet werden (Hierarchical Model View Controller):



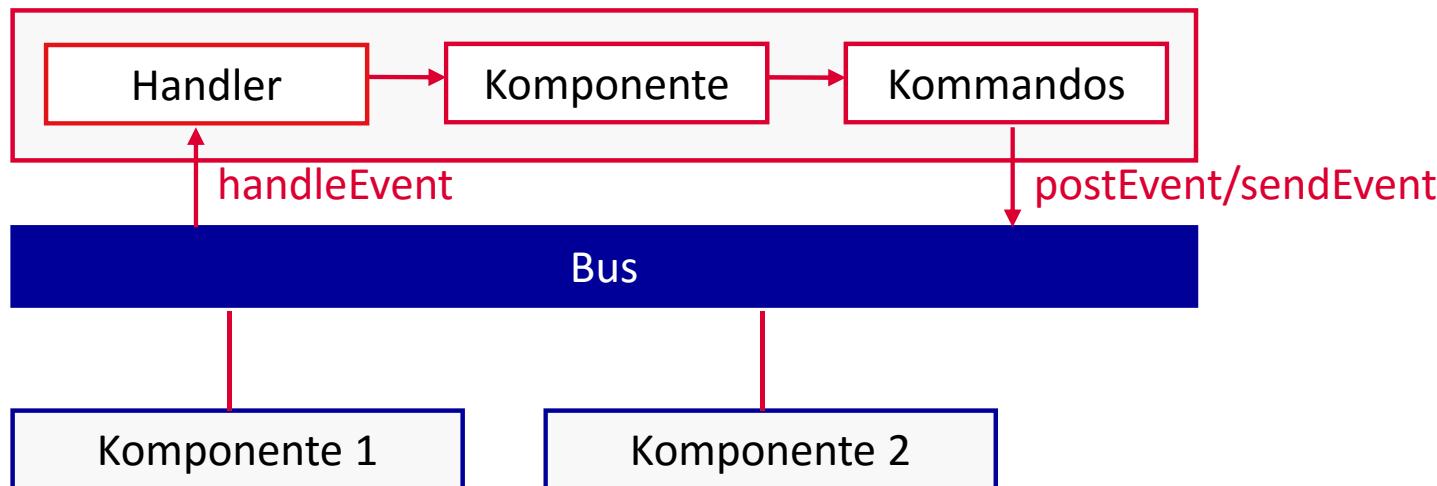
- Ein Controller gibt „übersetzte“ Ereignisse an die höhere Ebene weiter.

Anbindung an die Geschäftslogik

Dialogsteuerung durch Nachrichten auf einem Bus



- In der Elektrotechnik findet die Kommunikation häufig in Form von Nachrichten auf einem Bus statt (PCI, ...).
- Hier: Übernahme der Idee zur Verknüpfung von Java-Komponenten.
- Schon bekannt: siehe Einführung zu OSGi („OSGi: Modularisierung mit Declarative Services“)



- Die Ereignisbehandlung kann manuell ausprogrammiert oder z.B. durch Zustandsautomaten beschrieben werden.



- Eclipse 4 „verpackt“ den Event-Admin-Service aus OSGi für eine einfachere Verwendbarkeit. Die Klasse **IEventBroker** besitzt zwei Sende-Methoden:
 - ◆ **void send(String topic, Object data)**: versendet die Daten **data** zum Ereignis **topic** synchron:
 - Der Aufruf erfolgt im Thread des Senders.
 - Der Sender arbeitet erst dann weiter, wenn die Nachricht zugestellt ist.
 - ◆ **void post(String topic, Object data)**: versendet die Daten **data** zum Ereignis **topic** asynchron:
 - Der Aufruf erfolgt in einem eigenen Thread des Dienstes.
 - Der Sender arbeitet sofort weiter, auch wenn die Nachricht noch nicht zugestellt ist.
- Für beide gilt:
 - ◆ Die Daten **data** sollten sich während des Sendens nicht ändern, da sie nicht kopiert werden.
 - ◆ Der **IEventBroker** wird per Dependency Injection übergeben.
 - ◆ Soll **data** mehr als ein Objekt beinhalten, dann wird häufig eine **Map** verwendet.



- Der Nachrichtenempfang kann auf zwei Arten erfolgen:
 - ◆ Per Dependency Injection (einfacher): Es wird eine Methode für eine beliebige Klasse geschrieben, die auf Nachrichten bestimmter Type wartet, Beispiel:
 - **public void copy(@EventTopic(topic) Object data)**: Wird immer dann aufgerufen, wenn eine Nachricht des Typs **topic** verschickt wurde.
 - **data** ist der gesendete Datentyp. Es darf auch die konkrete Klasse des Nachrichteninhalt (z.B. **Map**) angegeben werden.
 - **topic**: Typ der Nachricht, wie vom Sender vergeben oder mit Wildcard versehen (siehe OSGi-Event-Admin)
 - **@EventTopic**: Die Nachricht wird im Thread des Senders bzw. des Nachrichtendienstes zugestellt.
 - **@UIEventTopic**: Die Nachricht wird immer im UI-Thread zugestellt.
 - ◆ Durch direktes Registrieren als Beobachter am Dienst (**subscribe**-Methode des **IEventBroker**) → soll hier nicht näher betrachtet werden.

Anbindung an die Geschäftslogik

Dialogsteuerung durch Nachrichten auf einem Bus



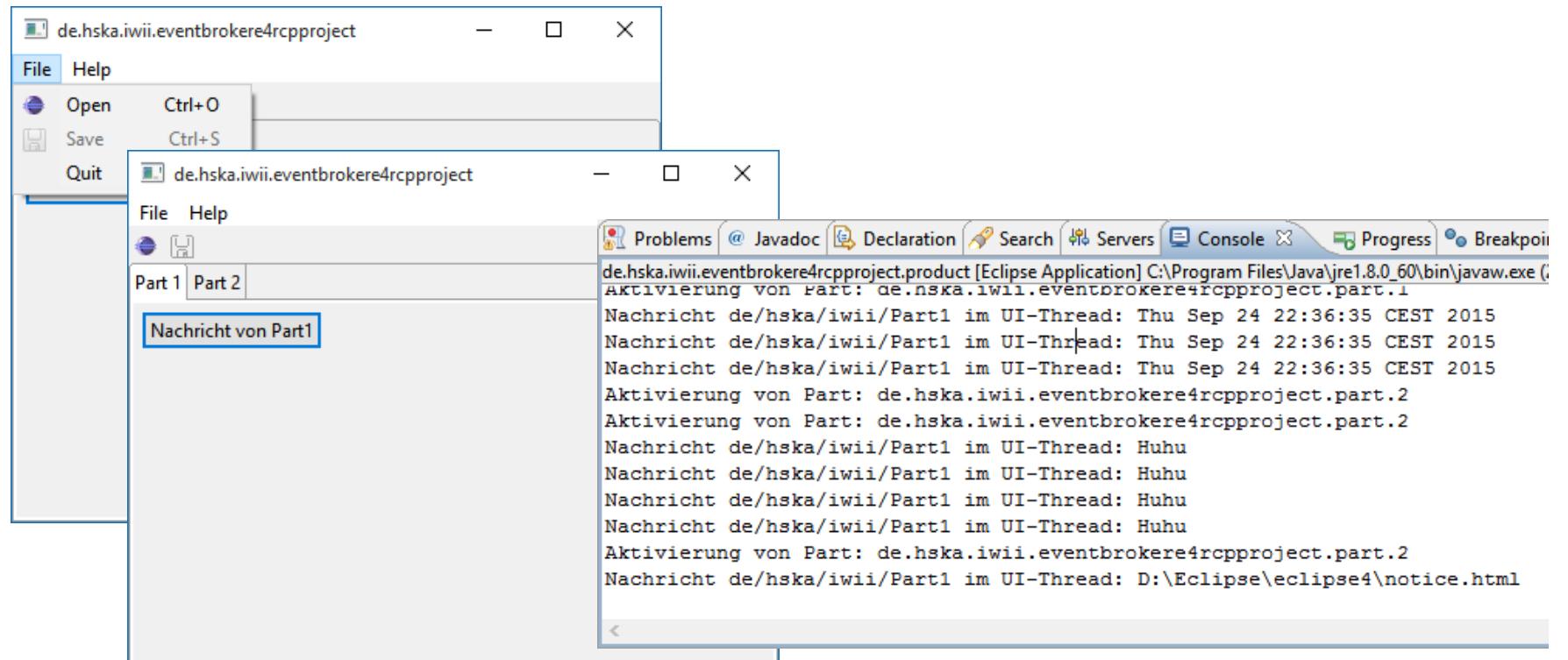
- Es lassen sich auch alle internen Ereignisse beobachten:
 - ◆ Deren Typen (**topic**) sind als Konstanten in der Schnittstelle **UIEvents** definiert.
 - ◆ Beispiel: **UIEvents.UILifeCycle.ACTIVATE** beobachtet die Aktivierung eines Parts.
- Wichtig: **org.eclipse.e4.core.services.events** als Paket importieren (nicht Abhängigkeit zu dessen Plug-in herstellen)

Anbindung an die Geschäftslogik

Dialogsteuerung durch Nachrichten auf einem Bus



- Beispiel (**de.hska.iwii.eventbrokere4rcpproject**):
 - ◆ Jeder Part hat eine Taste, die beim Anklicken Nachrichten "**de/hska/iwii/Part1**" bzw. "**de/hska/iwii/Part2**" verschickt.
 - ◆ Das Handler für das Öffnen einer Datei verschickt eine Nachricht "**de/hska/iwii/FileOpen**" mit dem Pfad der geöffneten Datei.



Anbindung an die Geschäftslogik

Dialogsteuerung durch Nachrichten auf einem Bus



- Versand der Nachrichten, Part 1:

```
public class Part1 {  
    @Inject  
    private IEventBroker eventBroker;  
    private Button btn;  
  
    @Inject  
    public void createContents(Composite parent) {  
        parent.setLayout(new GridLayout(1, false));  
        btn = new Button(parent, SWT.PUSH);  
        btn.setText("Nachricht von Part1");  
        btn.addSelectionListener(new SelectionAdapter() {  
            @Override public void widgetSelected(SelectionEvent e) {  
                eventBroker.post("de/hska/iwii/Part1", new Date());  
            }  
        });  
    }  
    @Focus  
    public void setFocus() {  
        btn.setFocus();  
    }  
}
```

Anbindung an die Geschäftslogik

Dialogsteuerung durch Nachrichten auf einem Bus



- Versand der Nachricht durch den Handler zum Öffnen einer Datei:

```
public class OpenHandler {  
    @Execute  
    public void execute(IEclipseContext context,  
                        @Named(IServiceConstants.ACTIVE_SHELL) Shell shell,  
                        IEventBroker eventBroker) {  
        FileDialog dialog = new FileDialog(shell);  
        String path = dialog.open();  
        if (path != null) {  
            eventBroker.post("de/hska/iwii/FileOpen", path);  
        }  
    }  
}
```

- Als **topic** werden also immer Pfade unterhalb von "**de/hska/iwii/**" verwendet.

Anbindung an die Geschäftslogik

Dialogsteuerung durch Nachrichten auf einem Bus



- Empfang der Nachrichten:

```
@Creatable
public class Receiver {

    // Empfang der eigenen Nachrichten
    @Inject
    @Optional
    private void receive(@UIEventTopic("de/hska/iwii/*") Object message) {
        System.out.println("Nachricht de/hska/iwii/ im UI-Thread: " + message);
    }

    // Empfang interner Nachrichten, hier das Aktivieren eines Parts
    @Inject
    @Optional
    private void receiveInternal(@UIEventTopic(UIEvents.UILifeCycle.ACTIVATE)
                                 Event event) {
        MPart modelPart = (MPart) event.getProperty(UIEvents.EventTags.ELEMENT);
        System.out.println("Aktivierung von Part: " + modelPart.getElementId());
    }
}
```



- Anmerkungen:
 - ◆ **@Creatable** bewirkt, dass ein Objekt der Receiver-Klasse per **@Inject** in andere Objekte injiziert werden kann → das Receiver-Objekt wird automatisch erzeugt.
 - ◆ **@Inject @Optional** sorgt beim Erzeugen des Receivers dafür, dass die Methoden nicht aufgerufen werden, wenn die Parameterobjekte nicht zur Verfügung stehen.
- Erzeugen des Receivers z.B. im Lifecycle-Objekt:

```
public class LifecycleManager {  
    @Inject  
    private Receiver receiver;  
  
    // ...  
}
```

- Achtung: Wird der Receiver manuell per **new** erzeugt, dann wird Dependency Injection nicht unterstützt!

Anbindung an die Geschäftslogik

Dialogsteuerung durch externe Skripte



- Kein direkt neuer Ansatz, lediglich eine Anwendung der bisherigen Architekturen
- Was passiert, wenn das Verhalten der Anwendung konfigurierbar sein soll (durch den Anwender oder einen Berater in einer Firma)?
 - ◆ Anpassung des Quelltextes → keine gute Idee
 - ◆ Besser:
 - Auslagerung der Dialogsteuerung oder einzelner Funktionen in Skripte
 - Aufrufe der Skripte durch die Anwendung
 - Die Skripte können auf Objekte der Anwendung zugreifen.
 - Gut geeignet sind Skriptsprache, die im Falle einer Java-Anwendung auch in der JVM laufen (JavaScript, Scala, Groovy, JRuby, ...).
 - ◆ Das Skript darf natürlich auch einen Zustandsautomaten implementieren...

Anbindung an die Geschäftslogik

Dialogsteuerung durch externe Skripte



- Beispiel der IDS GmbH:

The screenshot illustrates the integration of business logic into a dialog interface. It shows three main components:

- Background_menu.pfr:** A configuration window showing a grid-based layout with various buttons and fields. A specific button labeled "Ausführen" is highlighted with a yellow oval.
- command_dialog.pfr:** A script editor window containing Groovy code. The code handles button logic, specifically enabling or disabling the "Ausführen" button based on user input and button presses. A red arrow points from the "Ausführen" button in the configuration window to this script.
- Eigenschaften (Properties) Window:** A panel showing the properties of a selected element. The "Name" field is set to "commandBtn". The "Skripte" (Scripts) section is expanded, showing a "Selected" closure. This closure contains code to import the necessary API and handle button events like "Mouse Down", "Mouse Up", and "Focus In". A red box highlights this "Selected" closure.

```

background_menu.pfr
proFrame_initscript_7.groovy
command_dialog.pfr
commandBtn_selectedscript_6.groovy

1 import de.ids.acos.cc_api.util
2
3 def bits
4
5 if (overrideChkBox.getSelection())
6     // User wants to override
7     bits = new EventActionBits()
8
9 else
10    // User doesn't want to ov
11    bits = new EventActionBits()
12
13 // Send the new value to the s
14 if (pulseEdit.isVisible()) {
15     def pulse = new Scanner(pu
16     actionExecuter.executeWPul
17 } else {
18     actionExecuter.execute(tar
19 }
20
21 statusbar.setText(messages.get(
22
23
24 // Disable the buttons. Either
25 // valueChangedClosure when th

```

```

182
183 setButtonsEnabled = {isEnabled ->
184     onBtn.setEnabled(isEnabled)
185     offBtn.setEnabled(isEnabled)
186     enableCommandBtn(false)
187 }
188
189 // A closure that enables the command button depending on the isCommandBtnEnabled
190 // variable and the validity of the pulse edit entry field.
191 // It is called when the 'on', 'off' or 'command' button is pressed or the content
192 // of the pulse entry field changes.
193 def isCommandBtnEnabled = false
194 enableCommandBtn = {isEnabled = null ->
195     if (isEnabled != null)
196         isCommandBtnEnabled = isEnabled
197
198     commandBtn.setEnabled(isCommandBtnEnabled && pulseEdit.isValid())
199 }

```

Anbindung an die Geschäftslogik

Dialogsteuerung durch externe Skripte



The screenshot illustrates the integration of business logic into dialog control through external scripts. It shows four windows:

- background_menu.pfr**: A configuration window for a menu frame.
- proFrame_initscript_7.groovy**: A Groovy script containing logic for enabling buttons based on command availability and pulse edit validity.
- command_dialog.pfr**: A configuration window for a command dialog, showing a grid with buttons for 'on', 'off', 'Ausführen' (Execute), and 'Abbrech' (Cancel).
- cancelBtn_selectedscript_8.groovy**: A Groovy script containing the command to close the frame.

Annotations with arrows show the flow of logic from the UI elements in the configuration windows to the corresponding script code. For example, the 'Ausführen' button in the dialog window is linked to the 'execute' method in the Groovy script.

```
background_menu.pfr
proFrame_initscript_7.groovy
command_dialog.pfr
cancelBtn_selectedscript_8.groovy
```

Anbindung an die Geschäftslogik

Dialogsteuerung durch externe Skripte



- Funktionsweise der Anwendung:
 - ◆ Es gibt in der Geschäftslogik vordefinierte Funktionen.
 - ◆ Die Oberfläche kann für einen Kunden angepasst oder komplett neu erstellt werden.
 - ◆ Jedes Widget, das aus einem Skript heraus angesprochen werden kann, erhält einen eindeutigen Namen.
 - ◆ Die Skriptsprache ist Groovy.
- Die Dialogsteuerung erhält hier direkt die Widget-Ereignisse.
- Sie übernimmt auch die Ankopplung der Geschäftslogik an Oberfläche.



- Im einfachsten Fall werden kleine Code-Schnipsel ohne vorherige Compilierung ausgeführt (ohne Fehlerbehandlung):

```
// Manager, der die Skript-Engines verwaltet
ScriptEngineManager manager = new ScriptEngineManager();
// Skript-Engine, die die Skripte ausführt.
ScriptEngine scriptEngine = manager.getEngineByName("javascript");
// Für das Skript einige Variablen vordefinieren.
engine.put("user", userText); // Name, Wert
engine.put("password", passwordText);
engine.put("okButton", button);
// Skript aufrufen.
URL jsFile = this.getClass().getResource("buttoncontrol.js")
engine.eval(new InputStreamReader(jsFile.openStream()));
```

Inhalt des Skriptes:

```
okButton.enabled = (user.text != "" && password.text != "")
```

Nachteil: Beim ersten Ausführen des Skriptes (und nach jeder Änderung) wird es neu übersetzt → blockiert die Interaktion kurzeitig.

Anbindung an die Geschäftslogik

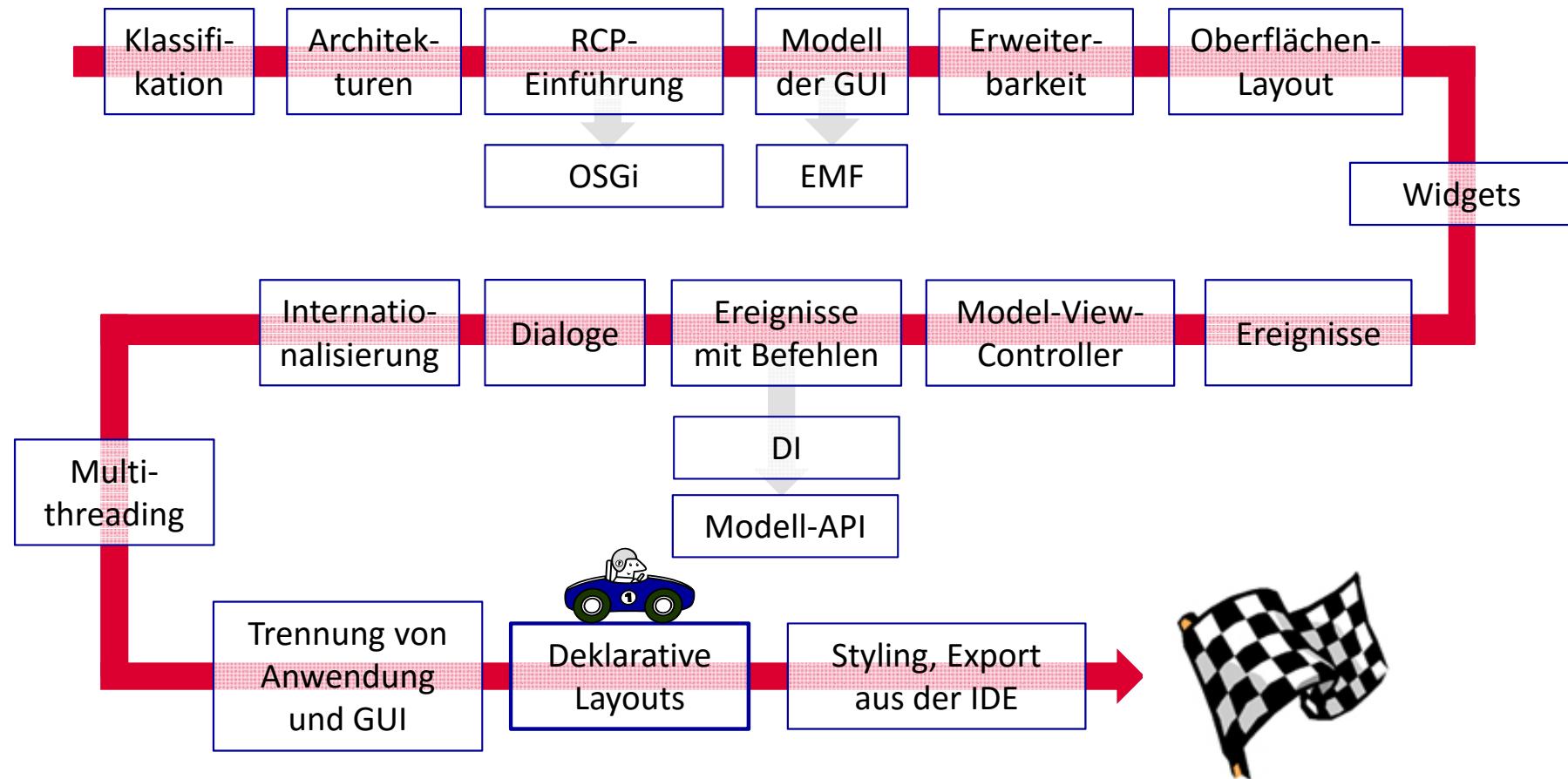
Dialogsteuerung durch externe Skripte



- Viele weitere Hinweise zur Architektur von Anwendungen mit Oberflächen finden sich im Buch von Mauro Marinilli (siehe Literaturliste).

Deklarative Beschreibungen

Einführung

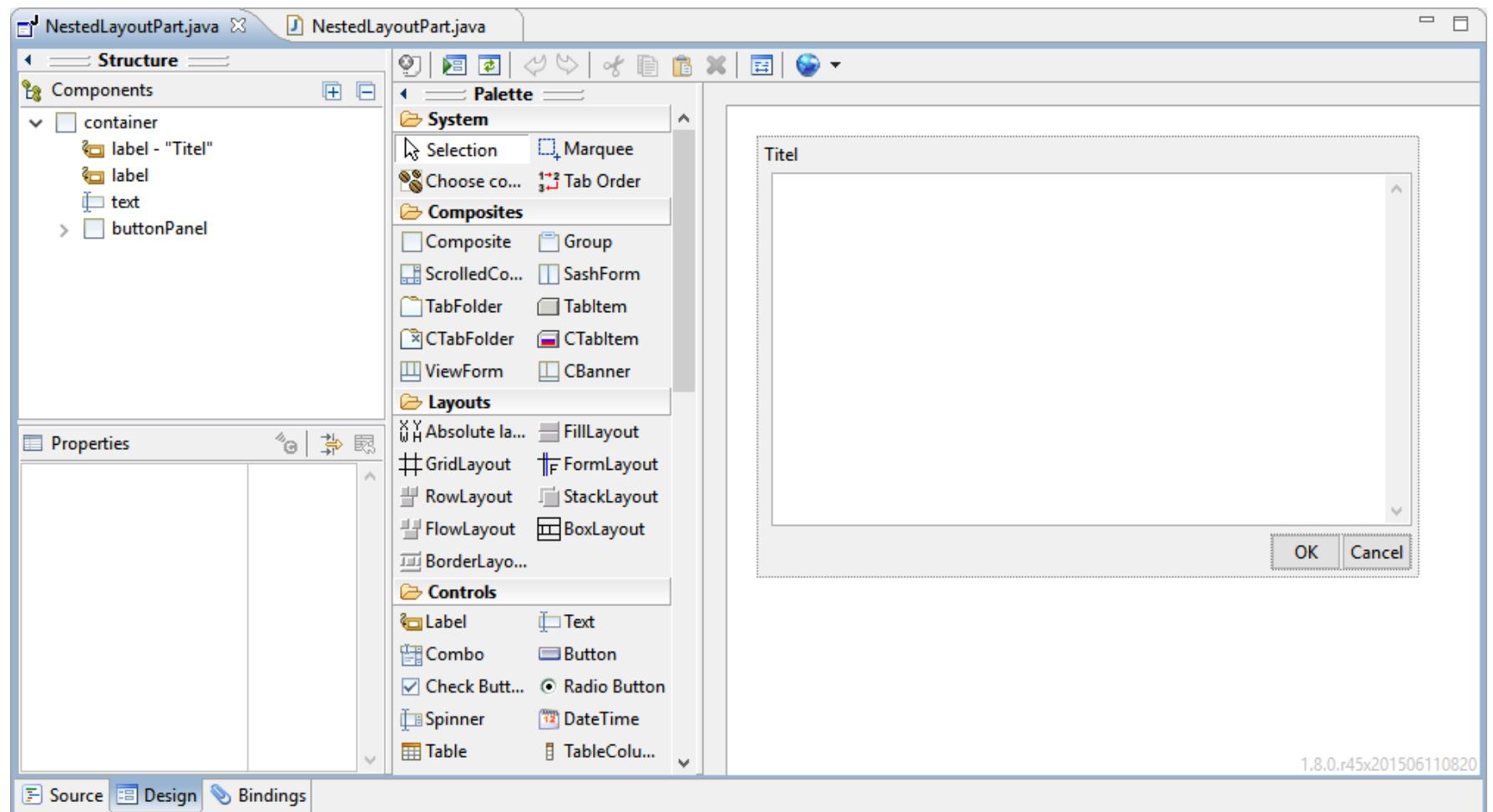


Deklarative Beschreibungen

Einführung



- Bisherige Erstellung grafischer Benutzeroberflächen:
 - ◆ Interaktiv in einem GUI-Editor

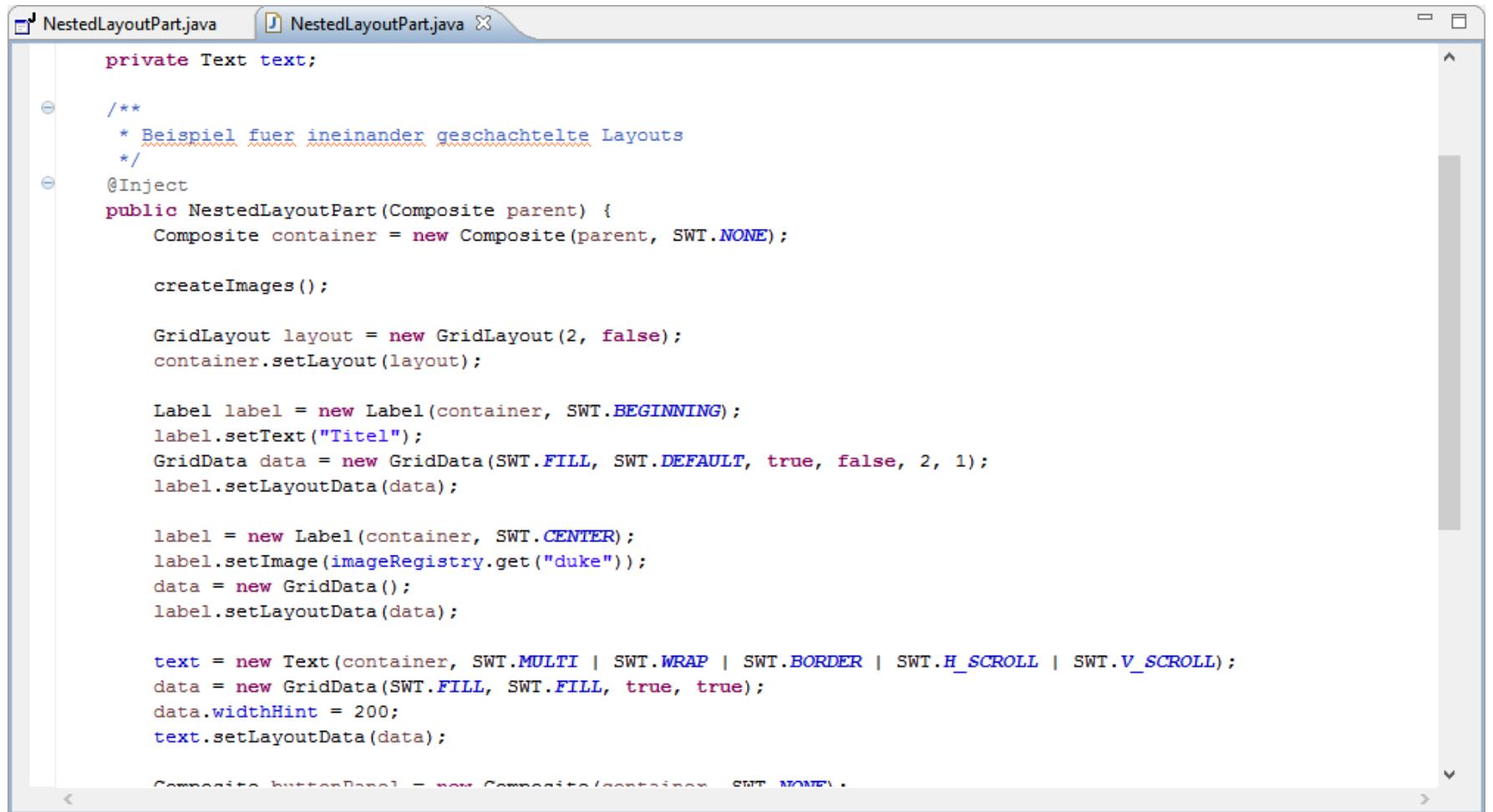


Deklarative Beschreibungen

Einführung



- ◆ Manuell programmiert



The screenshot shows a Java code editor window with two tabs: "NestedLayoutPart.java" and "NestedLayoutPart.java X". The code in the editor is as follows:

```
private Text text;

/**
 * Beispiel fuer ineinander geschachtelte Layouts
 */
@Inject
public NestedLayoutPart(Composite parent) {
    Composite container = new Composite(parent, SWT.NONE);

    createImages();

    GridLayout layout = new GridLayout(2, false);
    container.setLayout(layout);

    Label label = new Label(container, SWT.BEGINNING);
    label.setText("Titel");
    GridData data = new GridData(SWT.FILL, SWT.DEFAULT, true, false, 2, 1);
    label.setLayoutData(data);

    label = new Label(container, SWT.CENTER);
    label.setImage(imageRegistry.get("duke"));
    data = new GridData();
    label.setLayoutData(data);

    text = new Text(container, SWT.MULTI | SWT.WRAP | SWT.BORDER | SWT.H_SCROLL | SWT.V_SCROLL);
    data = new GridData(SWT.FILL, SWT.FILL, true, true);
    data.widthHint = 200;
    text.setLayoutData(data);

    Composite buttonPanel = new Composite(container, SWT.NONE);
```

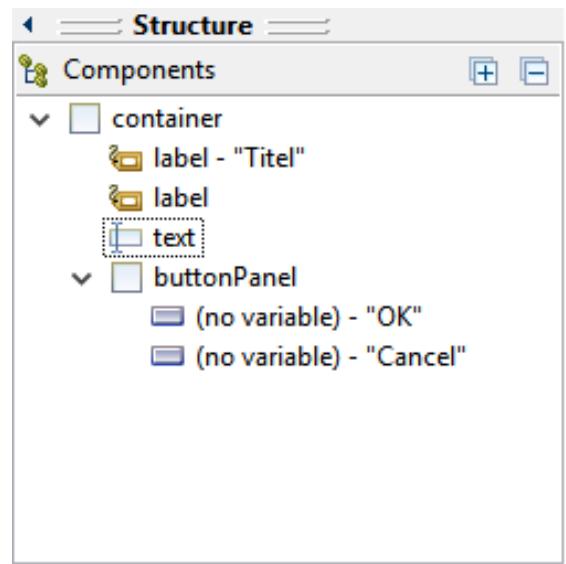
- ◆ Auch die Bindungen (das Data-Binding) wird manuell durchgeführt.

Deklarative Beschreibungen



Einführung

- Ergebnis: In beiden Fällen entsteht Quelltext, der einen Baum aus Widgets zur Laufzeit erzeugt.



Deklarative Beschreibungen

Einführung



- Wie sieht es deklarativ aus?

- ◆ (X)HTML:

The screenshot shows a code editor window with the title "Quelltext von: http://www.iwi.hs-karlsruhe.de/Intranetaccess/info/bulletinboard/INFB ...". The menu bar includes "Datei", "Bearbeiten", "Ansicht", and "Hilfe". The code itself is a fragment of an HTML menu. It starts with a closing

 tag at line 106, followed by a closing li tag at line 107. A comment "!-- Anleitungen -->" is at line 108. Line 109 contains an - tag with an anchor () pointing to "/Intranetaccess/info/faqs/INFB" with the text "FAQs". This is followed by a nested
 tag with the attribute "role='menu'" (line 110). The menu items are listed from line 111 to 126, each containing an anchor () with various href values like "/Intranetaccess/info/faqs/INFB", "/Intranetaccess/info/workflow/", and so on. The menu ends with another - tag with "role='presentation' class='divider'" at line 126.

```
106      </ul>
107    </li>
108    <!-- Anleitungen -->
109    <li><a href="/Intranetaccess/info/faqs/INFB">FAQs</a>
110    <ul role="menu">
111      <li><a href="/Intranetaccess/info/faqs/INFB">FAQs</a>
112      </li>
113      <li role="presentation" class="divider"></li>
114      <li><a href="/Intranetaccess/info/workflow/">Workflow</a>
115      </li>
116      <li><a href="/Intranetaccess/info/workflow/">Workflow</a>
117      </li>
118      <li><a href="/Intranetaccess/info/workflow/">Workflow</a>
119      </li>
120      <li><a href="/Intranetaccess/info/workflow/">Workflow</a>
121      </li>
122      <li><a href="/Intranetaccess/info/workflow/">Workflow</a>
123      </li>
124      <li><a href="/Intranetaccess/info/workflow/">Workflow</a>
125      </li>
126    <li role="presentation" class="divider"></li>
```

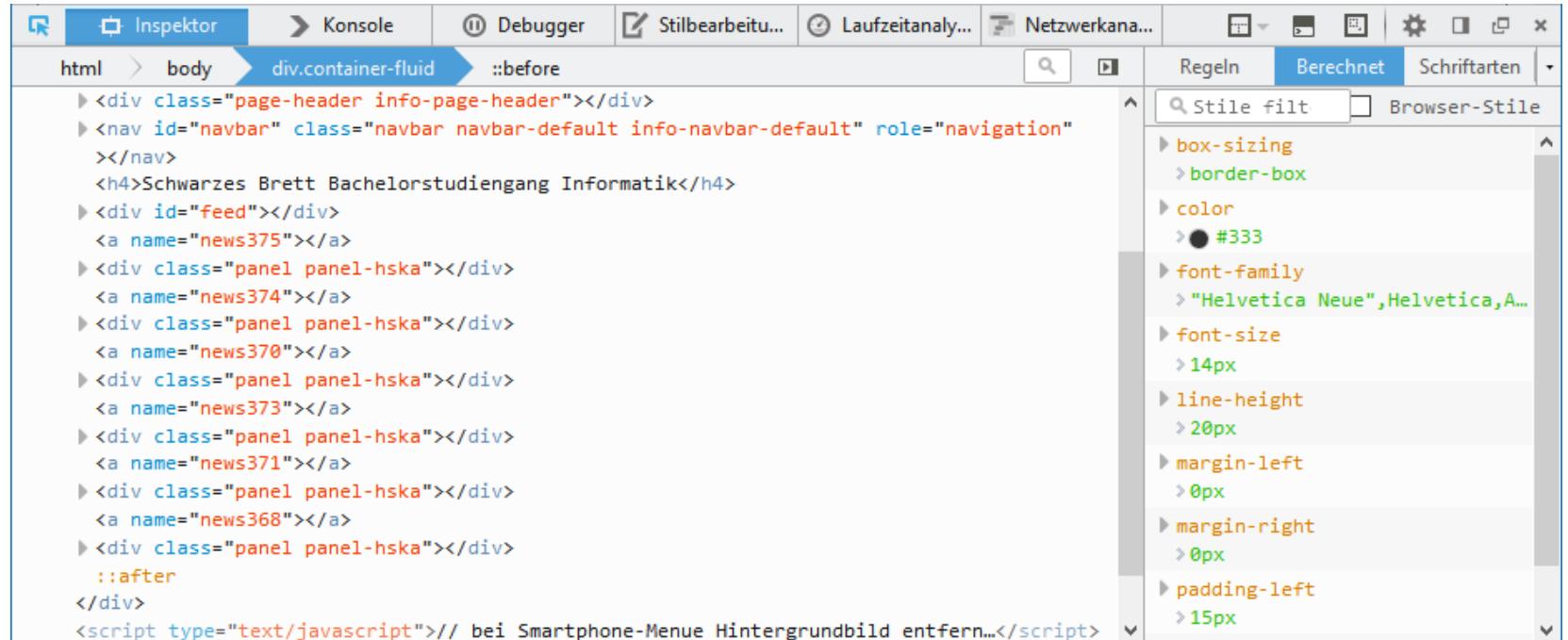
- ◆ Vorteile:
 - Auch von „Laien“ leichter zu erstellen.
 - Weniger Code.

Deklarative Beschreibungen

Einführung



- DOM-Baum bei Auszeichnungssprachen im Quelltext sichtbar!



The screenshot shows the Firefox Developer Tools interface with the 'Inspektor' (Inspector) tab selected. The left pane displays the DOM tree, starting from the root 'html' element and navigating through 'body', 'div.container-fluid', and '::before'. The right pane shows the 'Computed' tab of the style editor, listing various CSS properties and their values for the selected element.

Property	Value
color	#333
font-family	"Helvetica Neue", Helvetica, A...
font-size	14px
line-height	20px
margin-left	0px
margin-right	0px
padding-left	15px

- Bisher: DOM nicht aus dem Quelltext ersichtlich.



- Ziele:
 - ◆ Deklarative Erstellung von Oberflächen sowie deren Bindungen an die Geschäftslogik auch für Fat-Client-Anwendungen.
 - ◆ Trennung von deklarativem Oberflächenentwurf und Programmlogik.
- Wo wird dieser Ansatz schon verwendet (Auswahl)?
 - ◆ Microsoft: XAML als XML-basierte Beschreibungssprache für .NET-Anwendungen (auf WPF-Basis)
 - ◆ Qt:
 - Eine XML-Beschreibung wird auch durch den interaktiven GUI-Editor erzeugt.
 - Die Beschreibung wird zur Laufzeit eingelesen, um daraus die Oberfläche zu erzeugen.
 - ◆ Swing (diverse Toolkits, nicht sehr weit verbreitet)

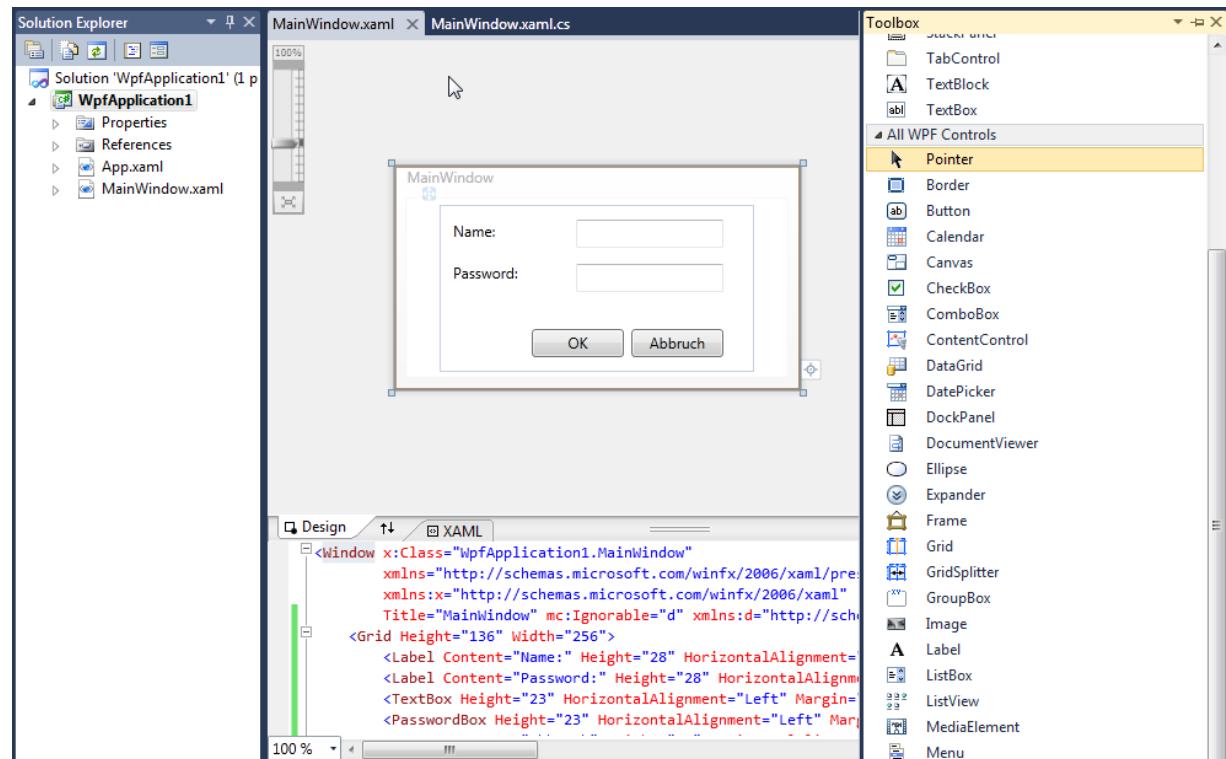
Deklarative Beschreibungen

Einführung



XAML

- XAML ist eine generische Markup-Sprache von Microsoft.
- Sie dient im Zusammenhang mit WPF (Windows Presentation Foundation) dazu, Oberflächen für .NET-Anwendungen deklarativ zu erstellen. Beispiel in Visual-Studio 2010:



Deklarative Beschreibungen



Einführung

- Das Ergebnis der interaktiven Konstruktion ist die XML-Datei mit der Oberflächen-Beschreibung (Kopf der XML-Datei zu verkürzt):

```
<Window .... Height="183" Width="331">
    <Grid Height="136" Width="256">
        <Label Content="Name:" Height="28" HorizontalAlignment="Left"
            Margin="6,9,0,0" Name="label1" VerticalAlignment="Top" Width="120" />
        <Label Content="Password:" Height="28" HorizontalAlignment="Left"
            Margin="6,43,0,0" Name="label2" VerticalAlignment="Top"
            Width="120" />
        <TextBox Height="23" HorizontalAlignment="Left" Margin="111,12,0,0"
            Name="textBox1" VerticalAlignment="Top" Width="120" />
        <PasswordBox Height="23" HorizontalAlignment="Left" Margin="111,48,0,0"
            Name="passwordBox1" VerticalAlignment="Top" Width="120" />
        <Button Content="Abbruch" Height="23" HorizontalAlignment="Left"
            Margin="156,101,0,0" Name="button1" VerticalAlignment="Top"
            Width="75" />
        <Button Content="OK" Height="23" HorizontalAlignment="Left"
            Margin="75,101,0,0" Name="button2" VerticalAlignment="Top"
            Width="75" />
    </Grid>
</Window>
```



- Warum ist XAML für Java-Entwickler interessant?
- Mit eFace gibt es eine Java-Implementierung, die XAML-Dateien einlesen und daraus Oberflächen für verschiedene Zielplattformen erzeugen kann: SWT/JFace, Swing, HTML
- Nähere Informationen: <http://www.soyatec.com/eface/>

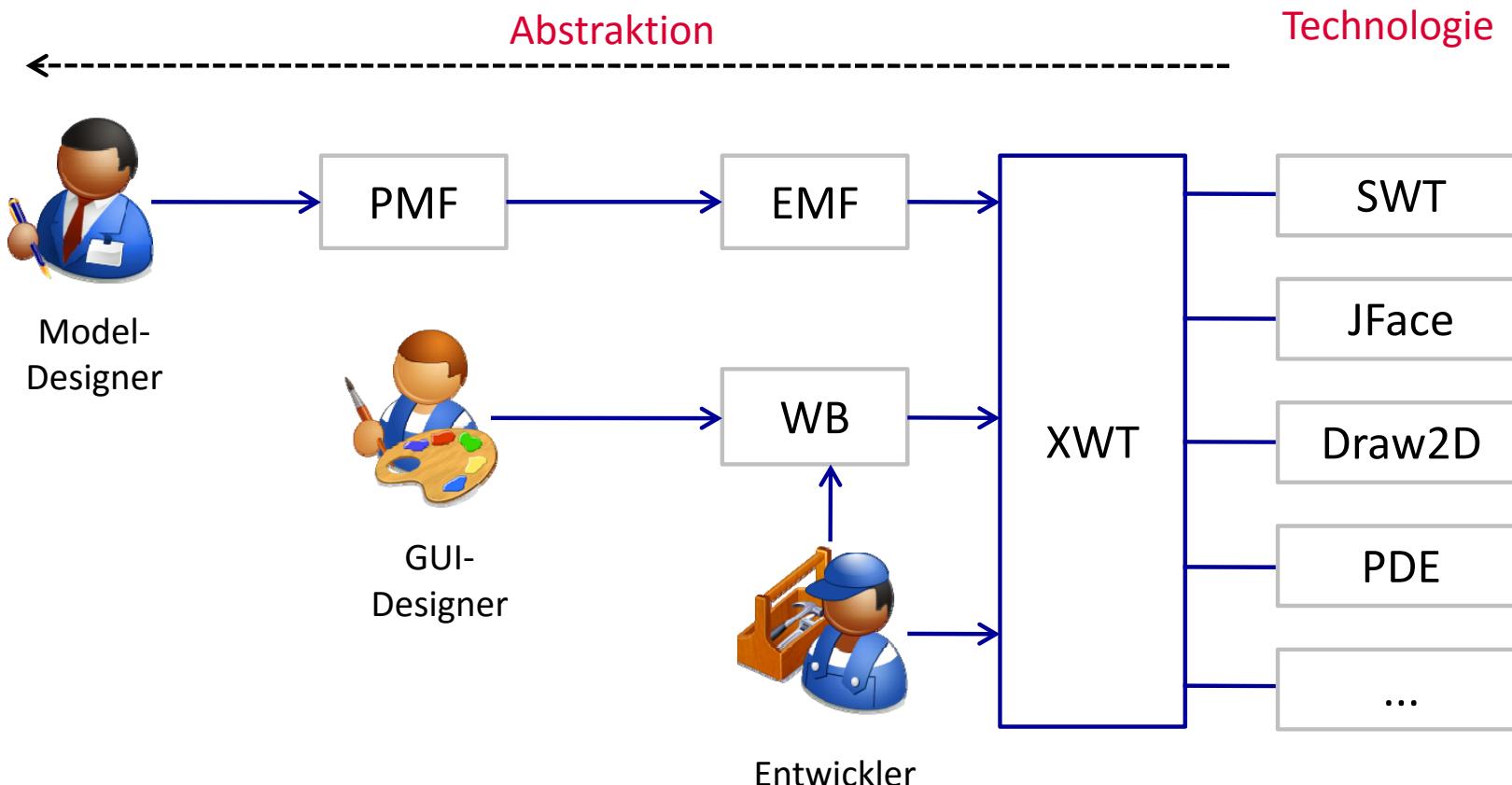
Deklarative Beschreibungen



XWT

XWT (eclipse XML Window Toolkit) für SWT/JFace-Oberflächen

XWT ist ein eFace-Klon speziell für Eclipse-Anwendungen.



[<http://dev.eclipse.org/blogs/yvesyang/files/2008/11/position.png>]

Deklarative Beschreibungen

XWT



- Prinzipieller Aufbau:



deklarative Beschreibung
der Oberfläche mit XML

Ereignisbehandlung und
Bean für das Data-Binding



Deklarative Beschreibungen



XWT

- Beispiel (von <http://wiki.eclipse.org/E4/XWT>)

```
<Shell xmlns="http://www.eclipse.org/xwt/presentation"
       xmlns:x="http://www.eclipse.org/xwt"
       x:Class="de.hska.iwii.xwte4rcpproject.xwteventdemo.EventHandler">
  <Shell.layout>
    <GridLayout/>
  </Shell.layout>
  <Button text="Click Me!" SelectionEvent="clickButton">
  </Button>
</Shell>
```

↑
Codebehind-Klasse

Entspricht dem Quelltext:

```
Shell parent = new Shell();
parent.setLayout(new GridLayout());
Button button = new Button(parent, SWT.NONE);
button.setText("Clicked!");
// Selection-Listener der Klasse EventHandler an der Taste button
// registrieren, der die Methode clickButton auruft.
```

Deklarative Beschreibungen

XWT



Einlesen der Deklaration und Aufbau der Oberfläche:

```
Shell parent = (Shell) XWT.load(parent, file);
```

Beispiel, wird manuell im Code erstellt und automatisch an die Taste in der Deklaration gebunden:

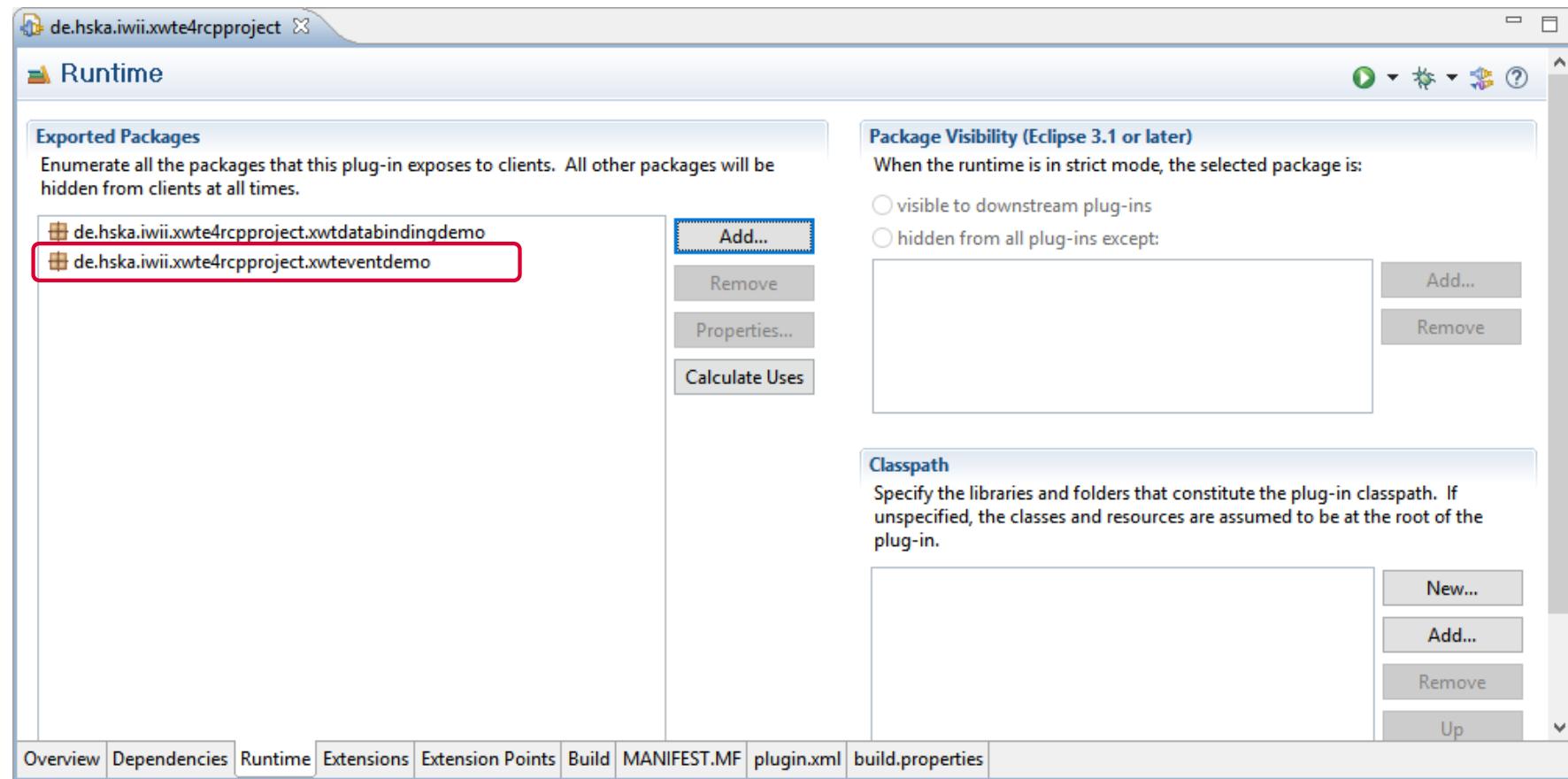
```
package de.hska.iwii.xwte4rcpproject.xwteventdemo;  
import org.eclipse.swt.Event;  
import org.eclipse.swt.Button;  
  
public class EventHandler {  
    protected void clickButton(Event event) {  
        Button button = (Button )event.widget;  
        button.setText("Clicked!");  
    }  
}
```

Deklarative Beschreibungen

XWT



- Achtung: Das Paket, in dem der **EventHandler** liegt, muss in der Datei **plugin.xml** exportiert werden.



Deklarative Beschreibungen



XWT: Data-Binding

- Das Data-Binding verwendet dieselbe Codebehind-Klasse wie für die Ereignisbehandlung. Beispiel:

Kundennummer:	31.415.926	Familienstand:	UNKNOWN
Vorname:	Holger	Nachname:	Vogelsang
Ruhestand:	<input type="checkbox"/>		

- Das Bean ist dasselbe wie im Abschnitt zum Data-Binding.
- Anbindung an das Bean (Ausschnitt nur für das erste Textfeld):

```
<Shell xmlns="http://www.eclipse.org/xwt/presentation"
       xmlns:x="http://www.eclipse.org/xwt"
       xmlns:j="clr-namespace:java.lang"
       x:Class="de.hska.iwii.xwte4rcpproject.xwtdatabindingdemo.Customer"
       x:text="Kunden-Dialog">
  <Shell.layout>
    <GridLayout numColumns="4" />
  </Shell.layout>
```

↑
Codebehind-Klasse

Deklarative Beschreibungen



XWT: Data-Binding

```
<Label x:text="Kundennummer:"/>
<!-- Textfeld mit Bindung an das Attribut customerNumber
     des Beans -->
<Text x:Name="customerNumber" x:Style="Border"
      text="{Binding path=customerNumber}"> ← Bindung
  <Text.layoutData>
    <GridData grabExcessHorizontalSpace="true"
              horizontalAlignment="GridData.FILL" widthHint="100"/>
  </Text.layoutData>
</Text>
```

- Bei geschachtelten Beans dürfen auch Pfade angegeben werden:
{Binding path=customerAddress.city.zipcode}
- Es sind auch Bindungen zwischen zwei Widgets oder Bindungen zu JFace-Klassen erlaubt.
- Aus hier muss die Codebehind-Klasse in der **plugin.xml** exportiert werden.

Deklarative Beschreibungen



XWT: Data-Binding

- Verwendung der XWT-Deklaration im Quelltext:

```
Customer customer = new Customer();
// Bean füllen...
customer.setCustomerNumber(31415926);
customer.setLastName("Vogelsang");
customer.setFirstName("Holger");
customer.setRetired(true);
customer.setFamilyStatus(Customer.FamilyStatus.MARRIED);

// XWT-Deklaration laden, Codebehind-Objekt übergeben
shell = (Shell) XWT.load(null,
    this.getClass().getResource("/XWTDatabindingDemo.xwt"), customer);
// Fenster anzeigen
shell.open();
```



- Internationalisierung wird natürlich auch unterstützt.
 - Dazu kann der Wizard für „Externalize Strings“ eingesetzt werden (wenn er denn funktioniert...).
1. Die Texte werden in die Ressource-Datei **messages** ausgelagert (Ausschnitt):

```
Text_FamilyStatus = Familienstand:  
Text_FirstName = Vorname:
```

2. Es wird die Zugriffsklasse **Messages** generiert (unvollständig):

```
public class Messages extends NLS {  
    private static final String BUNDLE_NAME = "messages";  
    public static String Text_FamilyStatus;  
    public static String Text_FirstName;  
    static {  
        NLS.initializeMessages(BUNDLE_NAME, Messages.class);  
    }  
}
```

Deklarative Beschreibungen



XWT: Internationalisierung

- Dann kann in der XWT-Datei auf die Schlüssel zugegriffen werden:

```
<Shell xmlns="http://www.eclipse.org/xwt/presentation"
    xmlns:x="http://www.eclipse.org/xwt"
    xmlns:j="clr-namespace:java.lang"
    xmlns:d="clr-namespace:de.hska.iwii.xwte4rcpproject.xwt.databindingdemo"
    x:Class="de.hska.iwii.xwte4rcpproject.xwt.databindingdemo.Customer"
    x:text="{x:Static d:Messages.Text_ShellTitle}">

    <Shell.layout>
        <GridLayout numColumns="4" />
    </Shell.layout>

    <Label x:text="{x:Static d:Messages.Text_CustomerNumber}"/>
```

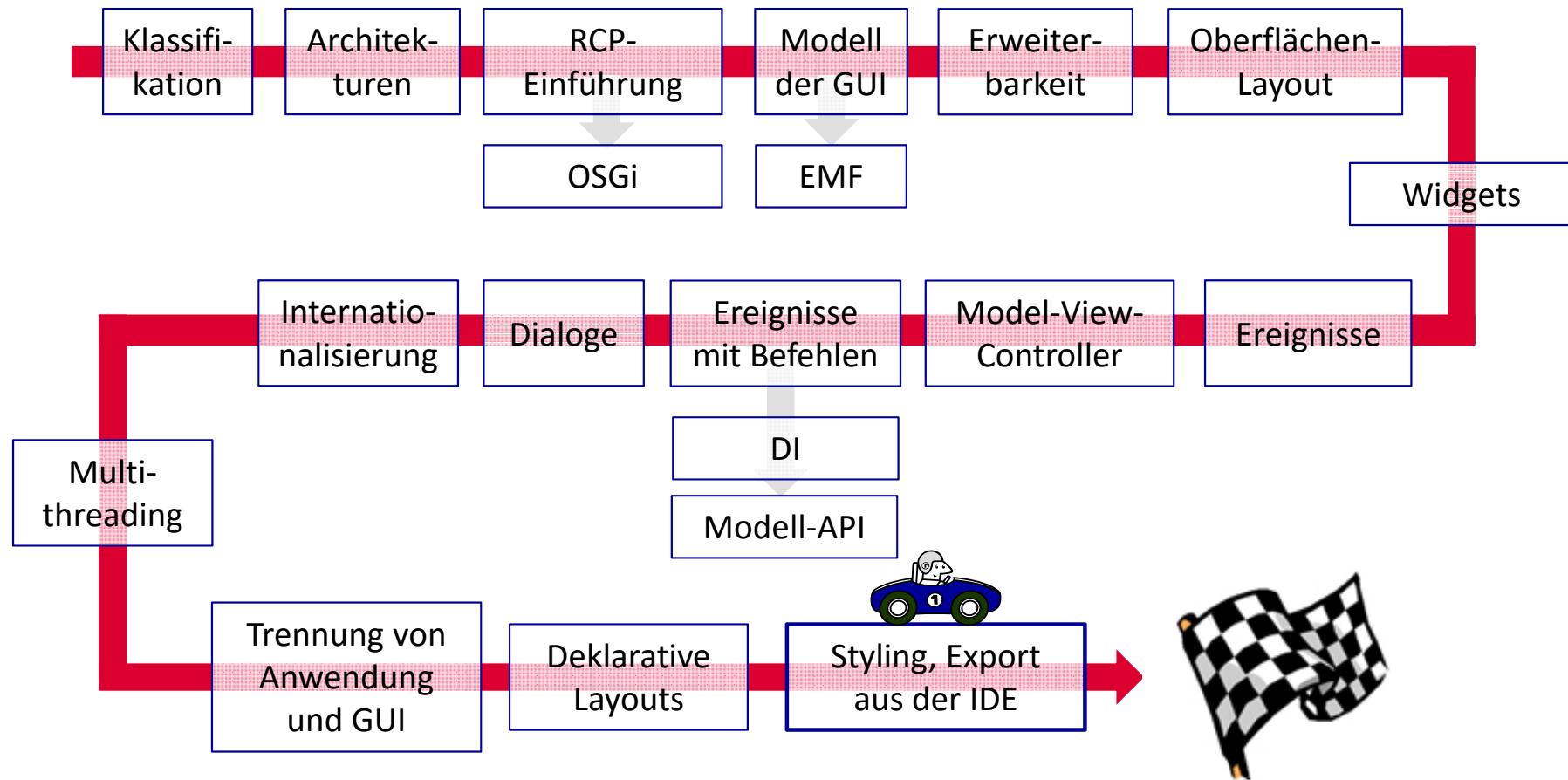
- **x:Static**: Statische Ressource im Namensraum x
- **d:Messages.Text_ShellTitle**:
 - ◆ **Messages**: Name der Zugriffsklasse
 - ◆ **d**: Namesraum der Zugriffsklasse
 - ◆ **Text_ShellTitle**: Schlüssel, dessen String verwendet werden soll



- Der WindowBuilder kann statt Java-Code auch XWL-Dateien erzeugen → kein manuelles Basteln am Code erforderlich.
- Weitere Literatur:
 - ◆ <http://wiki.eclipse.org/E4/XWT>
 - ◆ Eclipse-Magazin 2/2010, Seite 19-24 (inkl. Erklärungen zum Data-Binding)
- → siehe Hilfe zu XWT in Eclipse (noch recht wenig hilfreich...)
- Oberflächen lassen sich teilweise auch aus Modellen erzeugen:
 - ◆ Projekt Review: <http://www.redview.org/>
 - ◆ Soll hier nicht näher betrachtet werden.

Styling und Export

Splash-Screen und Styling mit CSS





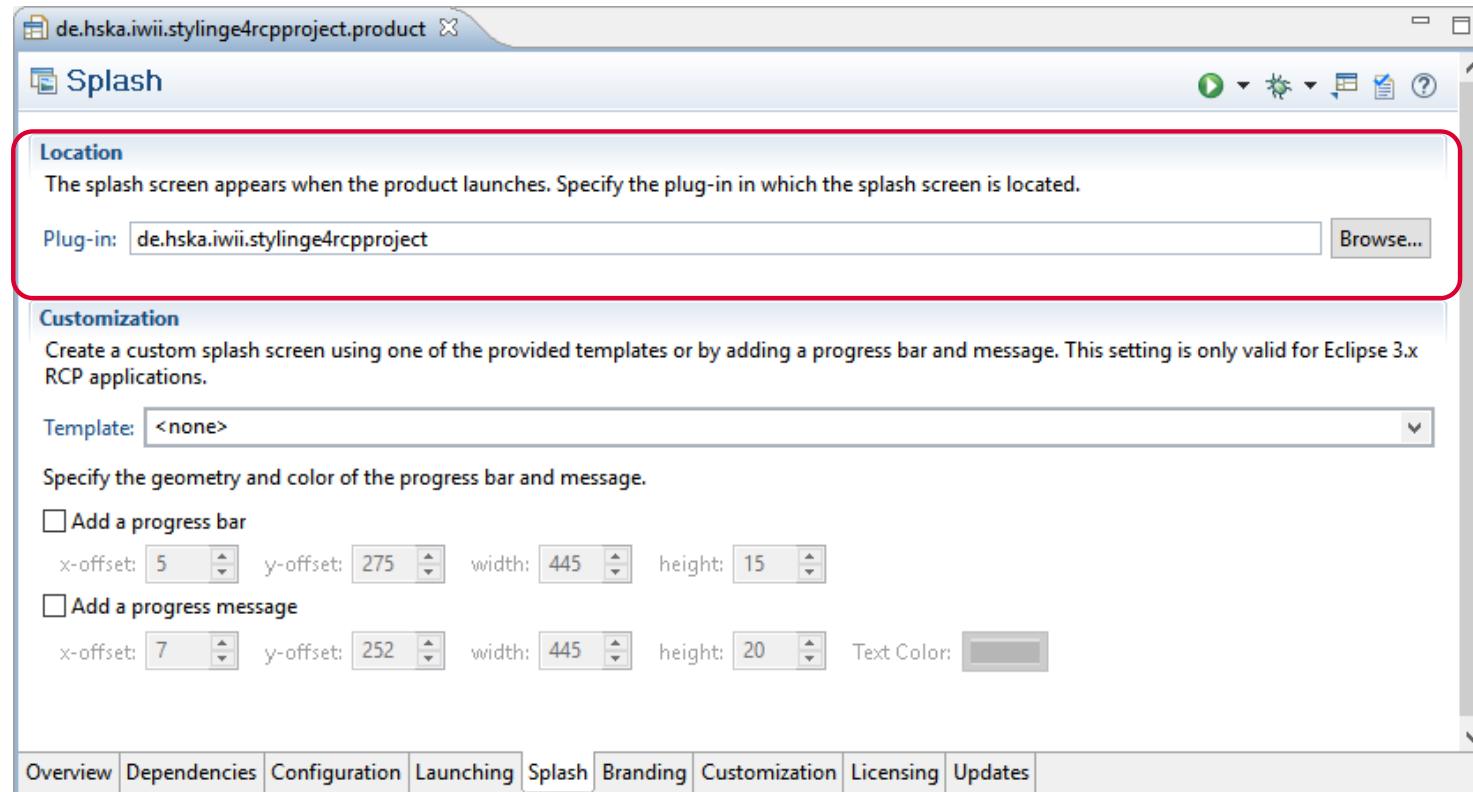
- Vor der Auslieferung an den Kunden soll die Optik eventuell noch angepasst werden:
 - ◆ Splash-Screen („Startbildschirm“)
 - ◆ Farben und Zeichensätze des Programms
 - ◆ Icons (Fenster und für den nativen Starter)
 - ◆ Copyright-Hinweise
- Splash-Screen:
 - ◆ Die Meldung wird während des Programmstarts angezeigt.
 - ◆ Es handelt sich um eine BMP-Datei mit dem Namen **splash.bmp**, die im Hauptverzeichnis eines Plug-ins liegen muss.
 - ◆ Das Plug-in mit der Datei wird im Produkt eingestellt.

Styling und Export

Splash-Screen und Styling mit CSS



- Auswahl des Plug-ins:



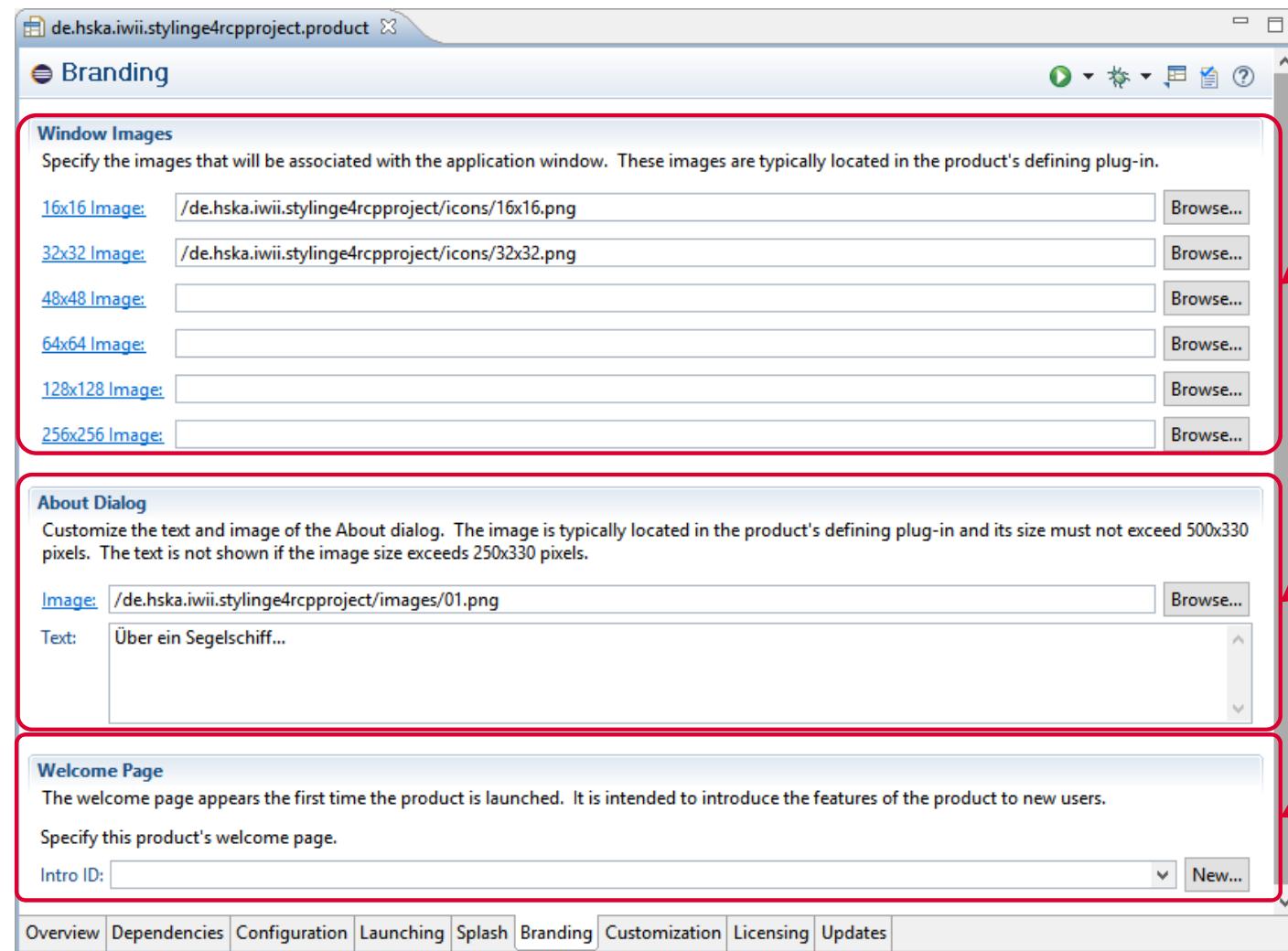
- Zusätzlich können auch noch ein Fortschrittsbalken u.Ä. angezeigt werden → soll hier nicht besprochen werden.

Styling und Export

Splash-Screen und Styling mit CSS



- Unter „Branding“ lassen sich diverse Icons und andere Informationen ablegen:



Fensterdekorationen

Anpassung des
„About“-Dialogs

Ansicht, wenn das
Programm das erste
Mal gestartet wurde

Styling und Export

Splash-Screen und Styling mit CSS



Execution Environment
Specify the execution environment of the product. The respective default JRE will be bundled with the product.

linux | macosx | solaris | win32

Execution Environment: Environments...

Bundle JRE for this environment with the product.

Program Launcher
Customize the executable that is used to launch the product.

Launcher Name:

Customizing the launcher icon varies per platform.

linux | macosx | solaris | win32

Specify 7 separate BMP images:

16x16 (8-bit): Browse...
16x16 (32-bit): Browse...
32x32 (8-bit): Browse...
32x32 (32-bit): Browse...
48x48 (8-bit): Browse...
48x48 (32-bit): Browse...
256x256 (32-bit): Browse...

Use a single ICO file containing the 7 images:
File: Browse...

Launching Arguments
Specify the program and VM arguments to launch the product with.
Platform-specific arguments should be entered on their respective tabs.

All Platforms | linux | macosx | solaris | win32

Set arguments for: All Architectures

Program Arguments:
-clearPersistedState

VM Arguments:

Complete Arguments Preview:
Program Arguments:
-clearPersistedState

Overview | Dependencies | Configuration | Launching | Splash | Branding | Customization | Licensing | Updates

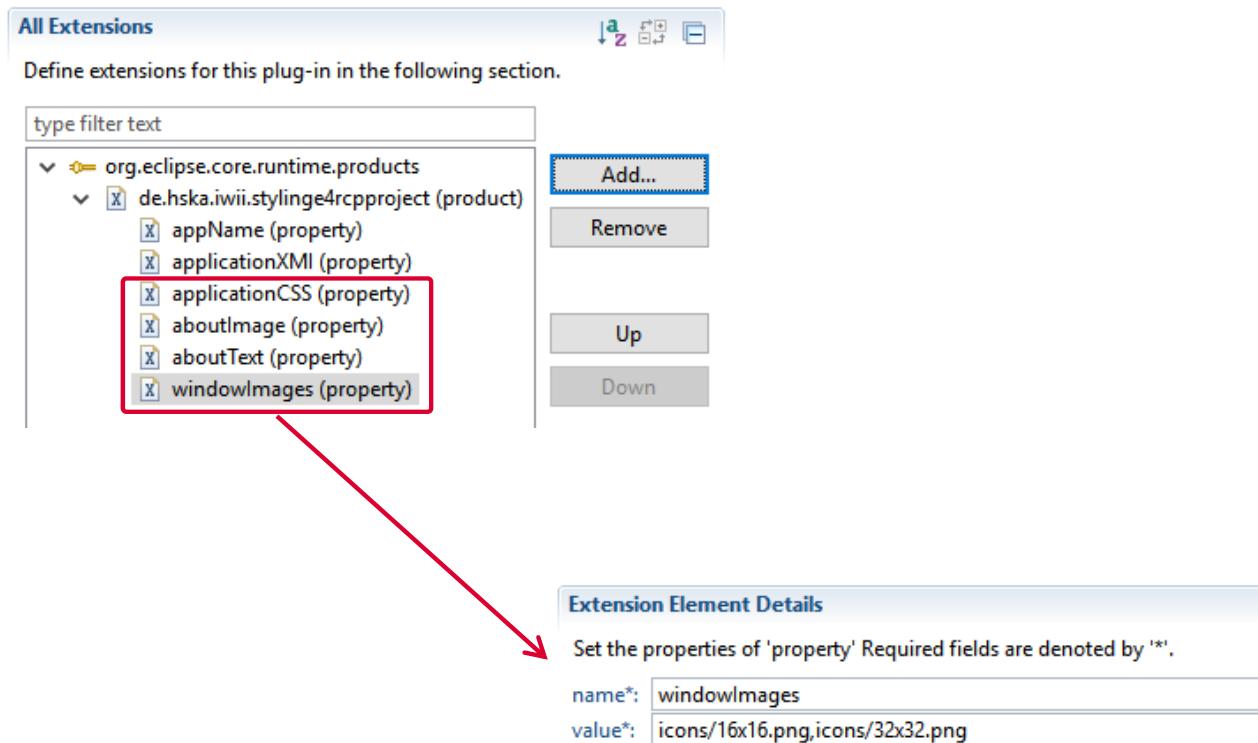
Icons für den nativen
Programmstarter
(plattformabhängig)

Styling und Export

Splash-Screen und Styling mit CSS



- Diese Informationen werden automatisch in die Datei **plugin.xml** als Eigenschaften des Produktes übernommen:



All Extensions

Define extensions for this plug-in in the following section.

type filter text

org.eclipse.core.runtime.products

de.haska.iwii.styling4rcpproject (product)

- appName (property)
- applicationXMI (property)
- applicationCSS (property)**
- aboutImage (property)
- aboutText (property)
- windowImages (property)

Add... Remove Up Down

Extension Element Details

Set the properties of 'property' Required fields are denoted by '*'.

name*: windowImages

value*: icons/16x16.png,icons/32x32.png



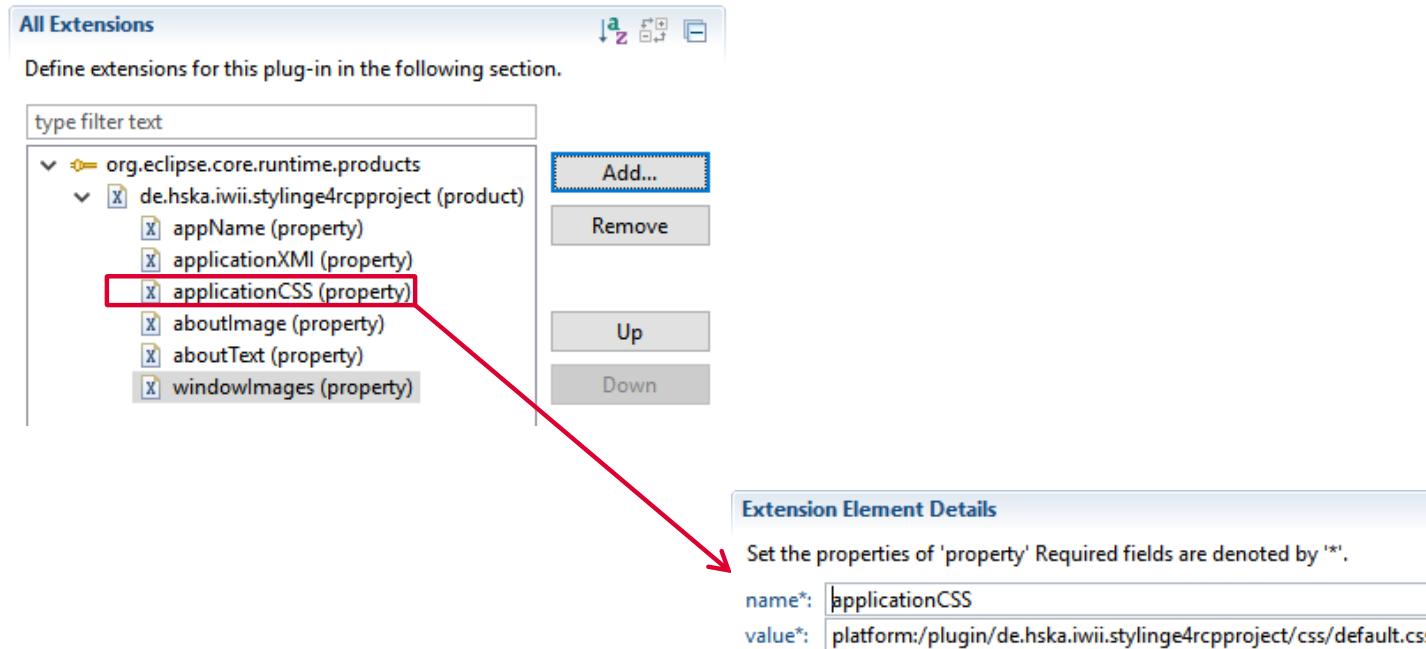
- Wichtig ist auch die optische Gestaltung des Programms mit CSS-Dateien.
- Hier lassen sich einige Eigenschaften der Widgets einstellen:
 - ◆ Vorder- und Hintergrundfarben, Hintergrundbilder
 - ◆ Zeichensätze
 - ◆ Ränder im die Widgets
 - ◆ Icons
 - ◆ ...
- Leider können je nach Fenstersystem nicht alle Eigenschaften verändert werden:
 - ◆ Windows: Farben in den Tasten (Buttons) sind fest.
 - ◆ Menüzeilen
 - ◆ ...
- Das Styling findet in einer oder mehreren CSS-Dateien statt. Verwaltung:
 - ◆ statisch in der Datei **plugin.xml**
 - ◆ dynamisch durch einen Theme-Manager verwaltet, der Wechsel der CSS-Dateien zur Laufzeit unterstützt → soll hier nicht betrachtet werden

Styling und Export

Splash-Screen und Styling mit CSS



- Auswahl der CSS-Datei in der Datei `plugin.xml`:





- Aufbau der CSS-Datei (Dokumentation ist bescheiden...):

- ◆ Selektor für ein SWT-Widget, Beispiel:

```
Label {  
    color: #dc0031;  
    font-weight: bold;  
    background: transparent;  
}
```

- Dadurch werden alle Label, die keine eigenen Regeln besitzen, so formatiert.
 - Der Name des Selektors entspricht dem SWT-Klassennamen.

- ◆ Selektor für die Klasse der Elemente, die durch das Anwendungs-Modell definiert werden, Beispiel:

```
.MTrimmedWindow {  
    background-color: radial #000098 white #000098 50% 100%;  
}
```

- Die Regel bezieht sich auf alle Fenster.
 - Der Name des Selektors entspricht dem des Modellelementes mit einem vorangestellten Punkt.



- Selektor für eine selbstdefinierte Klasse von Elementen, Beispiel:

```
.OK {  
    font-weight: bold;  
    color: #dc0031;  
    font-size: 20px;  
}
```

- Es werden alle Widgets, die der Klasse **OK** angehören, so formatiert.
- Der Name des Selektors entspricht dem Namen der Klasse mit einem vorangestellten Punkt.
- Allen Widgets, die zu dieser Klasse gehören sollen, muss der Klassename zugewiesen werden, Beispiel:

```
Button btn = new Button(buttonPanel, SWT.PUSH);  
btn.setText("OK");  
stylingEngine.setClassname(btn, "OK");
```

- **stylingEngine** ist eine Referenz auf ein **IStylingEngine**-Objekt, das per Dependency Injection übergeben wird.



- Selektor für ein Widget mit einer festen ID, Beispiel:

```
#TitleName {  
    color: #dc0031;  
    font-weight: bold;  
    font-size: 20px;  
}
```

- Nur das Widget mit der ID **TitleName** wird ausgewählt.
- Der Selektorname entspricht dem der ID mit vorangestelltem **#**.
- Die ID muss dem Widget zugewiesen werden, Beispiel:

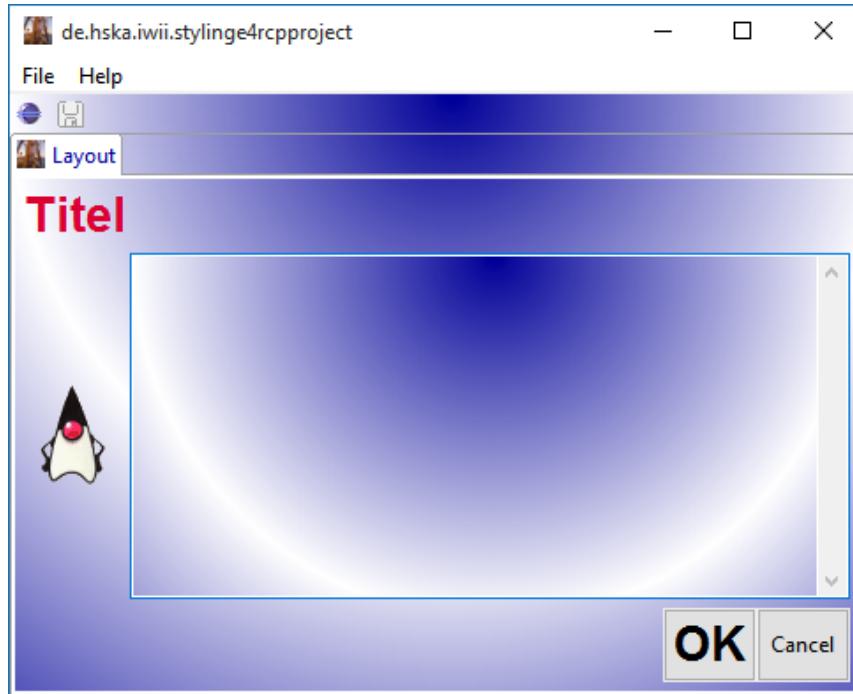
```
Label label = new Label(container, SWT.BEGINNING);  
label.setText("Titel");  
stylingEngine.setId(label, "TitleName");
```
- **stylingEngine** ist auch hier wieder ein per DI übergebenes Objekt, dessen Klasse **IStylingEngine** implementiert.

Styling und Export

Splash-Screen und Styling mit CSS



- Beispiel, wirklich unschön (Projekt **stylinge4rcpproject**):





- Welche Attribute unterstützen die einzelnen Widgets?
 - ◆ für SWT-Widgets: siehe http://wiki.eclipse.org/E4/CSS/SWT_Mapping
 - ◆ für Modell-Elemente: siehe SWT-Widgets, die die Modell-Elemente rendern (wenn SWT verwendet wird)
 - ◆ Hier gibt es noch eine ganz gute Beschreibung:
<http://wiki.eclipse.org/Eclipse4/RCP/CSS>
- Für Neugierige gibt es den CSS-Spy. Mit ihm lassen sich die CSS-Eigenschaften laufender Anwendungen anzeigen.
 - ◆ Zunächst sollte er installiert werden (siehe Links am Anfang des Dokuments).
 - ◆ Dann sollte ihm eine bisher unbenutzte Tastenkombination zum Starten zugewiesen werden. Wählen Sie in der IDE **Window → Preferences → General → Keys** aus und geben Sie im Suchfeld **CSS Spy** ein.
 - ◆ Die Tastenkombination **ALT-SHIFT-F12** ist unter Windows in der Vorlesungskonfiguration frei.

Styling und Export

Splash-Screen und Styling mit CSS



- Der CSS-Spy zeigt sehr schön die benutzen Attribute an (hier in der IDE selbst):

The screenshot shows the Eclipse CSS Spy tool interface. At the top, there's a toolbar with various icons and a tab labeled 'CSS'. Below the toolbar is a 'CSS Selector' input field and a 'All shells' checkbox. The main area is divided into three sections: 'Widget', 'CSS Class', and 'CSS Id'. The 'Widget' section shows a tree view of UI components under 'Shell (org.eclipse.swt.widgets)'. The 'CSS Class' section lists classes like 'MTrimmedWindow', 'MPartSashContainer', 'MPerspectiveStack', 'MPerspective', and 'MPartStack'. The 'CSS Id' section lists IDs such as 'IDEWindow', 'PerspectiveStack', and 'org-eclipse-jdt-ui-JavaPerspective'. In the bottom left, there's a 'CSS Properties' table with rows for 'background' and 'background-color'. In the bottom right, there's a 'CSS Rules' panel containing the CSS properties: 'margin-top: 0.0px;' and 'margin-bottom: 0.0px;'. A checkbox 'Show unset properties' is checked, and a button 'Show CSS fragment' is visible.

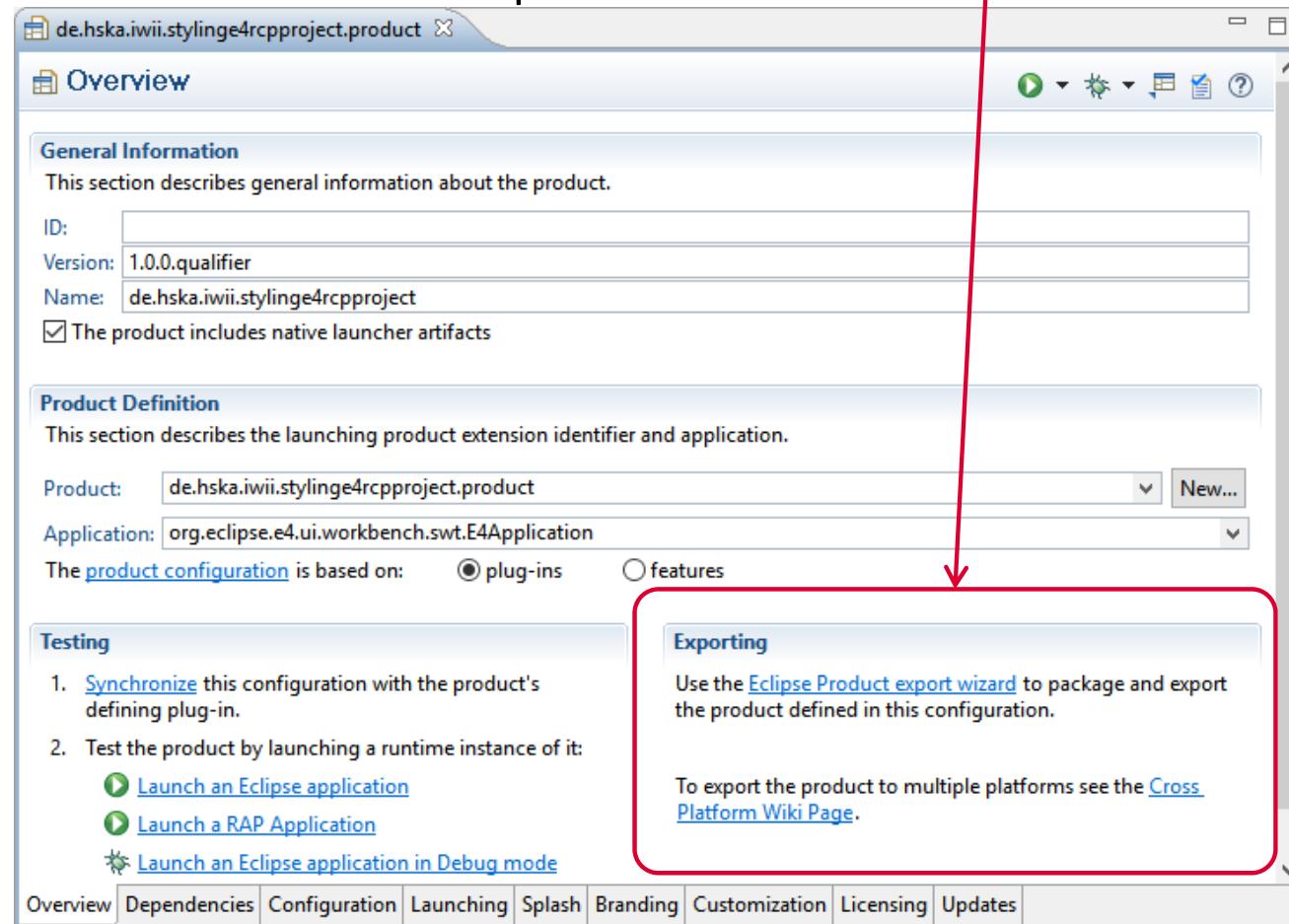
- Die Eigenschaften lassen sich auch direkt ändern.
- Update Site: <http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools/>

Styling und Export



Export als ein lauffähiges Programm

- Ein Produkt kann mit **Eclipse Product export Wizard** aus der geöffneten Produktdatei heraus exportiert werden:

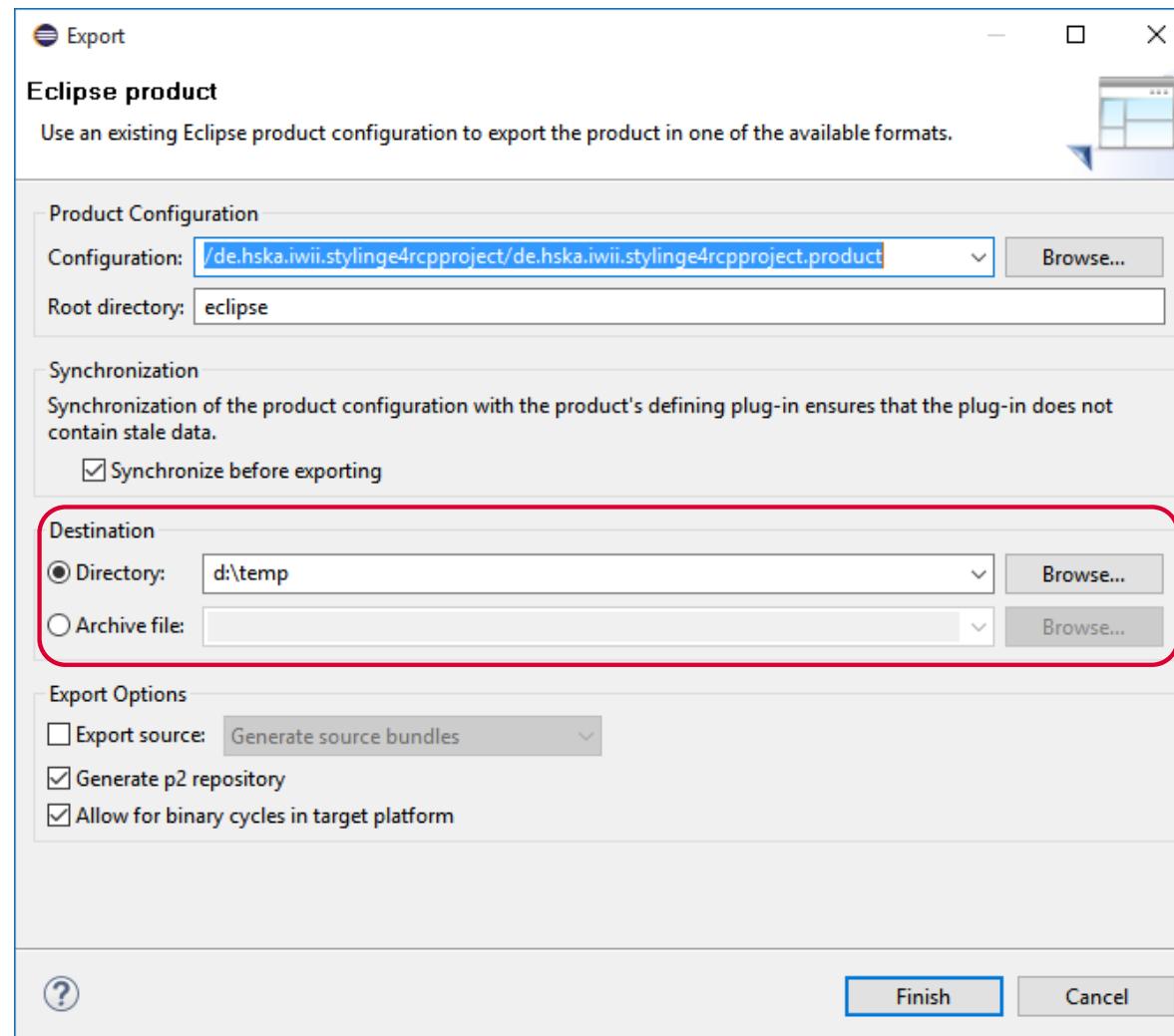


Styling und Export



Export als ein lauffähiges Programm

- Zielverzeichnis angeben oder gleich in eine jar-Datei exportieren:



Styling und Export



Export als ein lauffähiges Programm

- Es entsteht im angegebenen Verzeichnis ein Baum mit den generierten Dateien. Die Struktur entspricht der der IDE.

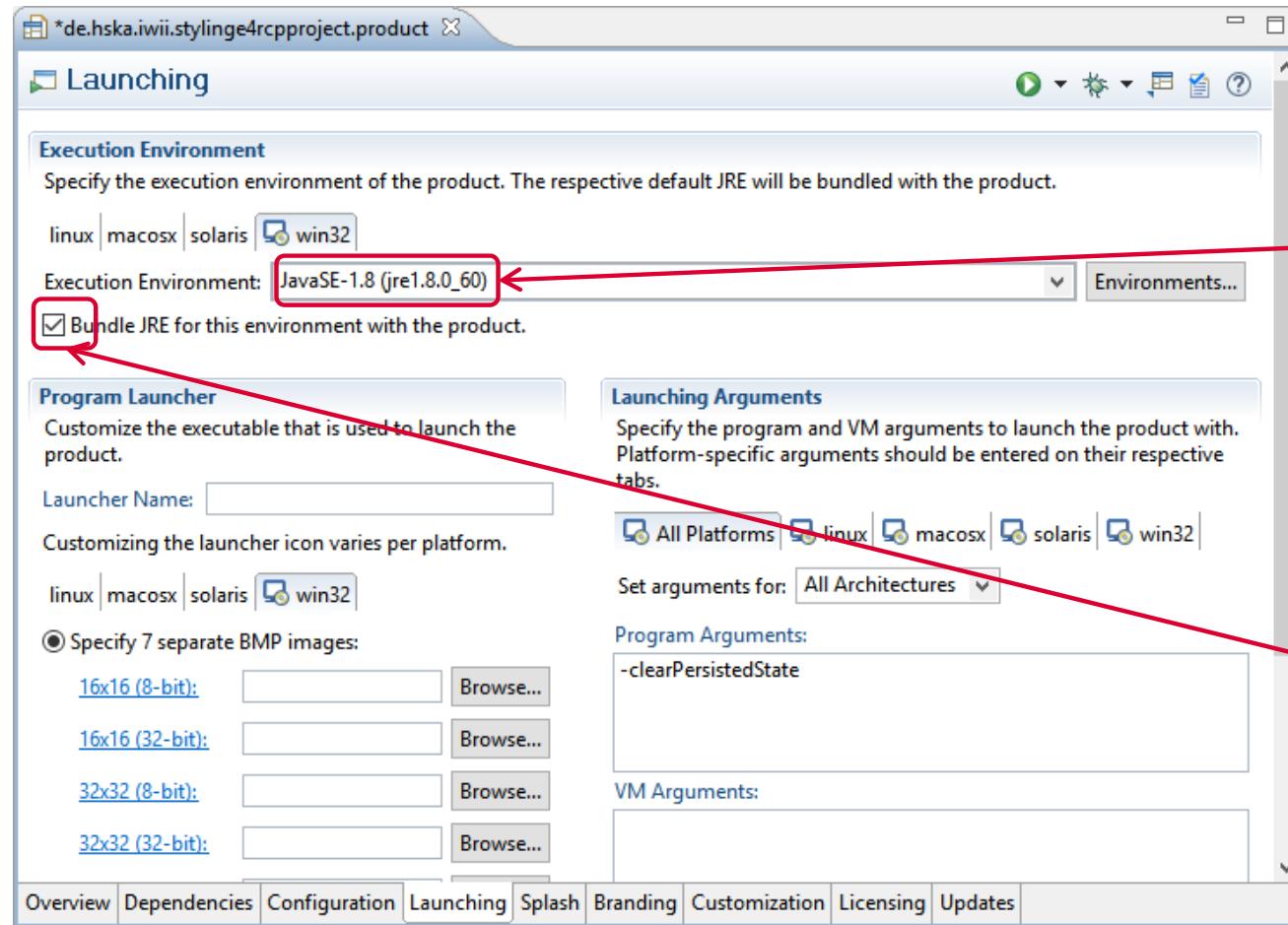
Name	Änderungsdatum	Typ	Größe
configuration	24.09.2015 23:09	Dateiordner	
p2	24.09.2015 23:09	Dateiordner	
plugins	24.09.2015 23:09	Dateiordner	
artifacts.xml	24.09.2015 23:09	XML-Dokument	15 KB
eclipse.exe	25.06.2015 16:52	Anwendung	313 KB
eclipse.ini	24.09.2015 23:09	Konfigurationsein...	1 KB

Styling und Export



Export als ein lauffähiges Programm

- Es kann sogar das benötigte JRE mit exportiert werden:



Laufzeitumgebung auswählen

JRE exportieren



- Einige Punkte, die die Benutzerfreundlichkeit steigern, werden nicht betrachtet:
 - ◆ Undo/Redo: für **TextViewer** siehe **org.eclipse.jface.text.IUndoManager**, eine allgemeine Lösung existiert nicht
 - ◆ Drag&Drop: siehe Paket **org.eclipse.swt.dnd**
 - ◆ Erleichterungen für sehgeschädigte Anwender (Accessibility): siehe Paket **org.eclipse.swt.accessibility**
 - ◆ Online-Hilfe: nicht direkt auf Basis von SWT und JFace unterstützt, aber Bestandteil der Rich Client Platform
 - ◆ Verwaltung von Eigenschaften, Konfigurationen
 - ◆ Features, Updates von Features (Features fassen mehrere Plug-ins zu einer Einheit zusammen)
 - ◆ „Branding“: Splash-Screen, Anpassung des Aussehens (Look&Feel, siehe http://wiki.eclipse.org/RCP_Custom_Look_and_Feel)
 - ◆ System-Tray, siehe:
<http://www.vogella.de/articles/EclipseRCP/article.html#systemtray>



- Was wird in Anwendungen noch benötigt?
 - ◆ Direktes Zeichnen: siehe Paket **org.eclipse.swt.graphics**
 - ◆ Drucken: siehe Paket **org.eclipse.swt.printing**
 - ◆ Einbettung von Windows-Komponenten bei Windows-Anwendungen: siehe Paket **org.eclipse.swt.ole.win32**
 - ◆ Einbettung von AWT/Swing-Komponenten:
 - siehe Paket **org.eclipse.swt.awt**, siehe auch Erklärungen unter <http://www.eclipse.org/articles/article.php?file=Article-Swing-SWT-Integration/index.html> und unter <http://www.eclipsezone.com/eclipse/forums/t45697.html>
z.B. VLC als Widget mit Hilfe von vlcj
<http://www.capricasoftware.co.uk/vlcj/index.php>
 - Alternativ: Darstellung der Swing-Komponenten mittels SWT:
<http://swingwt.sourceforge.net/>
 - ◆ „Ribbons“ unter <http://hexapixel.com/projects/ribbon> bzw.
<http://www.snakedj.ch/2009/01/30/swt-ribbon/>



- ◆ Bau einer Webanwendung mit
 - der Remote Application Platform (RAP, siehe <http://www.eclipse.org/rap/>)
 - Vaaclipse (verwendet Vaadin für die Widgets und zum Rendern des Modells, siehe <http://semanticsoft.github.io/vaacclipse/>)

Ende

