**Projekthandbuch**

Eclipse RCP ist ein Standardframework für Geschäftsanwendungen. Mit der neusten Generation E4 wurde Eclipse RCP vollständig modernisiert. Anhand einer wichtigen RCP Applikation der SBB wird eine Migration auf Eclipse E4 exemplarisch durchgeführt und die dabei berücksichtigten Aspekte dargestellt.

|  |  |
| --- | --- |
| Autor: | Mike Rothenbühler |
| Version: | 0.31 |
| Status: | In Arbeit |
| Ablage: | <https://github.com/MikeR13/MAS/blob/master/Deliverables/> |
| Institution: | Hochschule für Technik und Informatik Bern |
| Verteiler: | Brawand Ueli, Hoffmann Marc, Rothenbühler Mike |

**Versionkontrolle**

|  |  |  |  |
| --- | --- | --- | --- |
| **Datum** | **Version** | **Autor** | **Bemerkungen** |
| 03.06.2013 | 0.1 | MIRO | Erster Wurf |
| 07.06.2013 | 0.2 | MIRO | Aspekt 1, Mixing E3 / E4 |
| 17.06.2013 | 0.21 | MIRO | Aspekt 1, Mixing E3 / E4 |
| 19.06.2013 | 0.22 | MIRO | Aspekt 1, Mixing E3 / E4 |
| 25.06.2013 | 0.3 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 28.06.2013 | 0.31 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 30.06.2013 | 0.32 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 02.07.2013 | 0.33 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 03.07.2013 | 0.34 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 05.07.2013 | 0.35 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 08.07.2013 | 0.36 | MIRO | Aspekt 2, Adapters / Dependency Injection |
| 15.07.2013 | 0.4 | MIRO | Aspekt 3 Commands / Handler, Menus, Key Bindings |
| 16.07.2013 | 0.41 | MIRO | Aspekt 3 Commands / Handler, Menus, Key Bindings |

Inhaltsverzeichnis

1. Einleitung 6

1.1. Zweck des Dokumentes 6

1.2. Problemstellung 6

1.3. Randbedingungen 6

1.4. Situationsanalyse 6

1.5. Erbrachte Vorleistung 6

2. Vorbereitung 7

3. Aspekt „Mixing E3 / E4“ 7

3.1. Beschreibung des Aspektes 7

3.1.1. Diskussion der Eclipse RCP 4 Lösung 7

3.1.2. Vergleich mit Eclipse RCP 3 9

3.1.3. Vorteile E4 9

3.1.4. Einschränkungen und Risiken 9

3.1.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 10

3.1.6. Migrationspfade 10

3.2. Konkretes Beispiel RCS 12

3.2.1. Migration 13

3.2.2. Kritische Betrachtung der Möglichkeiten 19

3.2.3. Test 19

4. Aspekt „Adapters / Dependency Injection“ 20

4.1. Beschreibung des Aspektes 20

4.2. Adapters 20

4.2.1. Diskussion der Eclipse RCP 4 Lösung 20

4.2.1. Vergleich mit Eclipse RCP 3 21

4.2.2. Vorteile E4 23

4.2.3. Einschränkungen und Risiken 23

4.2.4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 23

4.3. Adapters: Konkretes Beispiel RCS 23

4.3.1. Migration 23

4.3.2. Test 23

4.4. Dependency Injection 23

4.4.1. Diskussion der Eclipse RCP 4 Lösung 24

4.4.2. Vergleich mit Eclipse RCP 3 32

4.4.3. Vorteile 32

4.4.4. Einschränkungen und Risiken 32

4.4.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 32

4.5. Dependency Injection: Konkretes Beispiel RCS 32

4.5.1. Migration 32

4.5.2. Test 32

5. Aspekt „Commands / Handler, Menus, Key Bindings“ 33

5.1. Beschreibung des Aspektes 33

5.1.1. Diskussion der Eclipse RCP 4 Lösung 33

5.1.2. Vergleich mit Eclipse RCP 3 38

5.1.3. Vorteile E4 38

5.1.4. Einschränkungen und Risiken 38

5.1.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 38

5.2. Diskussion der Eclipse RCP 4 Lösung 39

5.2.1. Vorteile 39

5.2.2. Vergleich mit Eclipse RCP 3 39

5.2.3. Einschränkungen und Risiken 39

5.2.4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 39

5.3. Konkretes Beispiel RCS 39

5.3.1. Definition Abnahmekriterien 39

5.3.2. Migration 39

5.3.3. Test 39

5.4. Beschreibung des Aspektes „Eigene Extension Points / Eigene Services“ 39

5.5. Diskussion der Eclipse RCP 4 Lösung 39

5.5.1. Vorteile 39

5.5.2. Vergleich mit Eclipse RCP 3 39

5.5.3. Einschränkungen und Risiken 39

5.5.4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 39

5.6. Konkretes Beispiel RCS 39

5.6.1. Definition Abnahmekriterien 39

5.6.2. Migration 39

5.6.3. Test 39

5.7. Beschreibung des Aspektes „Application Model vs. Advisors“ 39

5.8. Diskussion der Eclipse RCP 4 Lösung 39

5.8.1. Vorteile 39

5.8.2. Vergleich mit Eclipse RCP 3 39

5.8.3. Einschränkungen und Risiken 39

5.8.4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 39

5.9. Konkretes Beispiel RCS 39

5.9.1. Definition Abnahmekriterien 39

5.9.2. Migration 40

5.9.3. Test 40

5.10. 40

Reflexion 40

1. Einleitung
   1. Zweck des Dokumentes

Dieses Projekthandbuch informiert die an „Migration von Eclipse 3.x nach Eclipse 4“ beteiligten Parteien über die getätigten Migrationen von den ausgewählten Aspekten. Im Folgenden werden die Begriffe Eclipse 3.x RCP und E3 sowie Eclipse 4.x RCP und E4 identisch behandelt. Es wird also immer die Rede von Eclipse RCP und nicht der IDE sein.

* 1. Problemstellung

Da sich mit der Version 4 einiges an Eclipse RCP geändert hat ist eine Migration nicht einfach so zu bewerkstelligen. Es gibt aus der Community (noch) nicht viele Berichte zu gelungenen Migrationen, geschweige denn eine Anleitung wie eine solche Migration erfolgreich durchgeführt werden kann.

Es sollen Erkenntnisse gewonnen werden, wie eine erfolgreiche Migration durchgeführt werden kann, ohne dass die bestehende Applikation in den Punkten

* Funktionalität
* Performance
* Stabilität
* Usability
* Look and Feel

negativ beeinflusst wird. Die Arbeit an der bestehenden Applikation soll auch während der Migrationszeit möglich sein. Hierfür müssen Lösungen erarbeitet werden.

Mit den Erfahrungen und Ergebnissen aus der Master Thesis soll eine Migration auch für grosse Projekte relativ einfach möglich sein.

* 1. Randbedingungen

Das Projekt wird im Rahmen der Semesterarbeit und der Diplomarbeit durchgeführt. Dafür gelten die an der HTI üblichen Bedingungen.

* 1. Situationsanalyse

Um die erarbeiteten Migrationsvorschläge in der Praxis zu verifizieren, wird die Migration des RCP-Clients exemplarisch durchgeführt.

Bei dem RCS Client handelt es sich um die grösste Eclipse RCP Applikation der SBB mit folgenden Kennzahlen:

* 70 Plugins
* über 10 Hauptfenster
* Dutzende von Dialogen
* 270‘000 Zeilen Code

RCS wird zur Disposition des Zugverkehrs auf dem gesamten Streckennetz der SBB verwendet.

* 1. Erbrachte Vorleistung

Das Knowhow über Eclipse RCP 4 wurde bereits vor Projektstart aufgebaut.

1. Vorbereitung

Im Vorfeld wurde Know

1. Aspekt „Mixing E3 / E4“
   1. Beschreibung des Aspektes

In dieser Iteration soll geprüft werden welche Migrationsmöglichkeiten grundsätzlich existieren. Können Eclipse RCP 3 und Eclipse RCP 4 Komponenten im selben Projekt gleichzeitig nebeneinander im Einsatz sein? Um einen Kontext zu schaffen werden die beiden Versionen 3.x und 4.x kurz vorgestellt und einander gegenübergestellt.

* + 1. Diskussion der Eclipse RCP 4 Lösung



Abbildung Architektur E4

E4 läuft auf einer Java Virtual Machine (Fussnote), die Java Version sollte mindestens 6 sein.

Eclipse 4 RCP besteht aus den Komponenten:

|  |  |  |
| --- | --- | --- |
| **Komponente** | | **Beschreibung** |
| Eclipse 4.x Application Platform | Modeled UI | Das Application Model ist ein Konzept von E4. Die komplette Anwendung befindet sich auch als abstraktes Modell im Speicher. |
| CSS | Cascading Style Sheets erlaubt es Benutzeroberflächen optisch zu verändern ohne Code anzupassen. Es können Schriften, Farben und weitere Teile verändert werden. |
| Dependency Injection | Mit Dependency Injection werden Abhängigkeiten von Objekten zu anderen Objekten vom Container augelöst und gesetzt. Dadurch wird der Code des Objektes unabhängig von seiner Umgebung. Solche Klassen lassen sich um ein Vielfaches einfacher Unittesten als Objekte die sämtliche Referenzen selbst erzeugen. |
| Application Services | Services die von E4 zur Verfügung gestellt werden. Das sind zum Beispiel Services für das Logging, Eventhandling und Zugriffe auf das Application Model. |
| Equinox | OSGi Implementation von Eclipse. Die Equinox Runtime stellt das Framework, um modulare Eclipse Applikationen laufen zu lassen, zur Verfügung |
| EMF Core | Das Eclipse Modeling Framework ist ein quelloffenes Java-Framework zur automatisierten Erzeugung von Quelltext anhand von strukturierten Modellen (link to wiki) |
| SWT/ JFace | SWT ist die Standard User Interface Komponentenbibliothek von Eclipse. JFace stellt praktische APIs oberhalb von SWT zur Verfügung. |
| Workbench | | Im Prinzip ein Adapter der Aufrufe an die e4-Bundles weiterleitet. Die Kompatibilitätsschicht übersetzt API-Aufrufe aus E3 in die neue Welt, also E4. |
| JDT | | Die Java Development Tools sind eine Reihe von Plug-ins für die Entwicklungsumgebung Eclipse (link to wiki) |
| PDE | | Die Plugin Development Tools helfen bei der Entwicklung von Plugins. |
| Restliche Bestandteile der Plattform | | Dazu gehören Plugins wie Text, Team, Debug uvm… |

* + 1. Vergleich mit Eclipse RCP 3



Abbildung Architektur E3

Grundsätzlich sieht die Architektur von E3 ähnlich aus wie die von E4. Auch hier sind SWT und JFace im Einsatz. Equinox steht als OSGi Implementation zur Verfügung. Auch JDT, PDE und die restlichen Bestandteile der Plattform stehen zur Verfügung.

Der Unterschied bei E4 ist die Implementation der der Workbench (also das org.eclipse.org.workbench Plugin) und die neuen Technologien auf der diese neue Implementation basiert. Die Vorteile dieser neuen Technologien sind unten aufgeführt.

* + 1. Vorteile E4

Um zu verstehen was E4 für Vorteile bringt ist es sinnvoll die Nachteile von E3 aufzuführen:

* E3 ist sehr komplex, es gibt bei verschiedenen Punkten mehrere Wege die ans Ziel führen
* Viele API’s
* Viele Singletons
* Aufgrund von Abwärtskompabilität gibt es viel Legacy Code

E4 bietet ein moderneres Programmiermodell an als sein Vorgänger. Die vielfältigen APIs aus E3 wurden deutlich reduziert und vereinheitlicht. Die über das gesamte API verteilten Singletons wurden entfernt.

Neben der Vereinfachung wurden moderne Konzepte wie Dependency Injection und Declarative Styling eingeführt.

Die Implementation von Rich Client Applikation in Eclipse RCP wird mit der Version 4 flexibler und deutlich vereinfacht. Die Produktivität der Programmierer steigt, die Testbarkeit und die Wartung der Applikationen wird erleichtert.

* + 1. Einschränkungen und Risiken

Es gibt Berichte darüber, dass E4 in der momentan aktuellen Version 4.2 noch ziemlich unstabil und wenig performant ist. Ende diesen Monats (Juni 2013) kommt der Main Release von Kepler, also der Version 4.3, raus. Mit diesem Release sollen viele Performanceverbesserungen und eine grössere Stabilität daherkommen.

Die heute angebotenen API’s sind nicht unbedingt final und könnten sich in der Zukunft noch ändern. Es kann also sein, dass wer heute alles von E3 auf E4 migriert schon bald wieder Anpassungen tätigen muss.

* + 1. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3

Wie bereits oben erwähnt ist E4 mit der Version 4.2 qualitativ noch nicht auf dem Stand auf dem es sein sollte. Mit 4.3 sollen viele Verbesserungen kommen.

In Sachen Testbarkeit spricht alles für E4, der neue Ansatz mit den POJOs bringt bereits eine bessere Testbarkeit mit sich. Auch die Entfernung der Singleton‘s bringt eine grosse Verbesserung in Sachen Testbarkeit. Für Unit Test können nun die benötigten Services relativ einfach gemockt werden. Es kann ausschliesslich die Funktionalität einer Klasse bzw. Komponente getestet werden.

* + 1. Migrationspfade

http://eclipsesource.com/blogs/2012/06/18/migrating-from-eclipse-3-x-to-eclipse-4-e4/

Option 1: Compatibility Layer einsetzen

Der Compability Layer ermöglicht es E3 Applikationen ohne Codeanpassungen auf der Eclipse 4 Plattform zu laufen. Wenn man nicht auf E4 migriert so bleibt man kompatibel mit 3.x.

Um die Migration zu erleichtern bietet der Compability Layer 3.x Workbench APIs an und übersetzt alle Aufrufe in das Programmiermodell von E4. Im Hintergrund wird transparent ein Application Model erstellt. Eclipse 3.x Applikationen sollten jedoch keine interne Workbench APIs benutzen um mit dem Compability Layer zu funktionieren.

Die folgende Abbildung zeigt wie der Compability Layer auf der 4.0 Workbench aufsetzt.



Abbildung Compability Layer E3

Eine E3 Applikation die man auf dem E4 Compability Layer laufen lassen möchte benötigt zusätzlich die folgenden Plugins:

* org.eclipse.equinox.ds
* org.eclipse.equinox.event
* org.eclipse.equinox.util
* org.eclipse.e4.ui.workbench.addons.swt

Mit dem Compability Layer Ansatz alleine kann man aber nicht von den neuen Konzepten von E4 (Dependency Injection und Annotationen) Gebrauch machen. CSS funktioniert mit diesem Ansatz.

Option 2: Eine reine Eclipse RCP 4 Applikation

Hier sind alle Plugins reine E4 Plugins. Diese Option wird nicht näher betrachtet und kommentiert, da ja eine bestehende Eclipse RCP 3.x Applikation migriert und nicht neu geschrieben werden soll.

Option 3 Compatibility Layer und Eclipse RCP 4 Plugins

Hier werden neue Plugins in Eclipse RCP 4 Manier geschrieben, die alten werden auf Eclipse 3.x belassen und laufen koexistent auf dem Compability Layer.

Es gibt 3 Arten Eclipse RCP 4 Plugins im Compability Layer zu integrieren.

1. Prozessoren und Fragmente: Um dem Application Model – welches vom Compability Layer erstellt wird- Elemente hinzuzufügen werden Prozessoren und Fragmente benutzt. Hier gibt es aber aktuell noch Timing Probleme (Bug Link: <https://bugs.eclipse.org/bugs/show_bug.cgi?id=376486>), denn wenn die Prozessoren und Fragmente verarbeitet werden hat der Compability Layer das Application Model noch nicht komplett erstellt. Diese Option mag für Handles und View funktionieren aber nicht für Editor‘s.
2. LegacyIDE.xml: Das Application Model, das vom Compability Layer erstellt wird, wird kopiert und als Application Model registriert. Diesem Application Model können nun neue Eclipse RCP 4 Komponenten hinzugefügt werden. Das Model XMI-File – konkret: LegacyIDE.e4xmi - kann aus dem Plugin org.eclipse.platformherauskopiert werden. Plugins die nun Das Application Model erweitern wollen können dies über
3. 3.x e4-Bridge: Die 3.x e4-Bridge von Tom Schindl ermöglicht es Views und Editoren in Eclipse RCP 3.x wie auch 4 zu benutzen. Um diese Bridge einzusetzen wird das Plugin org.eclipse.e4.tools.compat benötigt. Dieses Plugin stellt als Basisklasse einen Wrapper um Eclipse RCPS 4 POJOs zur Verfügung, diese Klasse DIViewPart erbt vom Eclipse 3 RCP ViewPart. Eine vollständige Anleitung ist unter <http://eclipsesource.com/blogs/2012/06/18/migrating-from-eclipse-3-x-to-eclipse-4-e4/> zu finden.

Diese Möglichkeiten werden im Folgenden konkret in der Applikation RCS umgesetzt.

* 1. Konkretes Beispiel RCS

Die Option 1und 2 kommen für RCS nicht in Frage. Option 1 wäre auschliesslich eine E3 Applikation die dank dem Compability Layer auf E4 läuft. Mit diesem Ansatz können wir aber nicht von den neuen Konzepten von E4 profitieren. Option 2 bedingt ein komplettes Neuschreiben der Applikation. Auch diese Möglichkeit steht für RCS ausser Frage. Bleibt also „nur“ noch Option 3: Compatibility Layer und Eclipse RCP 4 Plugins. Die 1. Möglichkeit der Option 3 wird auch ausser Acht gelassen, da diese

Um eine Idee zu erhalten wie die oben aufgeführten 3 Möglichkeiten (2. Möglichkeit einmal mit Prozessor und einmal mit Fragment, 3. Möglichkeit 3.x e4-Bridge) zu bewerten sind habe ich mich entschlossen jede Option einmal auszuprobieren. Für die Migration habe ich einen möglichst einfachen ViewPart ausgewählt und diesen angepasst. Die Migrationsanleitungen und auch die Bewertung der jeweiligen Möglichkeit sind weiter unten aufgeführt.

Der einfache ViewPart sieht in der E3 Version folgendermassen aus. Der Einfachheit halber wurden die referenzierten Klassen, diverse Methoden, die Imports und die Kommentare weggelassen.

package ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider;

public class ZugnummerRendererView extends ViewPart {

private GraphViewer graphViewer;

@Override

public void createPartControl(Composite parent) {

final Composite composite = new Composite(parent, SWT.NONE);

composite.setLayout(new FillLayout());

graphViewer = new GraphViewer(composite);

graphViewer.setContentProvider(new ArrayContentProvider());

graphViewer.setInput(new Object[] { new Object() });

graphViewer.setViewPort(new WorldRectangle(0, 0, 100, 100));

graphViewer.setShapeProvider(new IShapeProviderImplementation());

}

@Override

public void setFocus() {

graphViewer.getControl().setFocus();

}

}

Abbildung Klasse ZugnummerRendererView E3 Stil

* + 1. Migration

Für alle drei Möglichkeiten wird der einfache ViewPart genau gleich angepasst. Er wird zum einfachen POJO welches über Dependency Injection das Parent Composite injiziert bekommt. Das Ganze sieht dann so aus:

package ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider;

….

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus;

……

public class ZugnummerRendererView {

private GraphViewer graphViewer;

@PostConstruct

public void createPartControl(Composite parent) {

final Composite composite = new Composite(parent, SWT.NONE);

composite.setLayout(new FillLayout());

graphViewer = new GraphViewer(composite);

graphViewer.setContentProvider(new ArrayContentProvider());

graphViewer.setInput(new Object[] { new Object() });

graphViewer.setViewPort(new WorldRectangle(0, 0, 100, 100));

graphViewer.setShapeProvider(new IShapeProviderImplementation());

}

@Focus

public void setFocus() {

graphViewer.getControl().setFocus();

}

}

Abbildung Klasse ZugnummerRendererView E4 Stil

Wir sehen, dass die Klasse jetzt von keiner anderen Klasse mehr erbt. Die Methoden createPartControlundsetFocus die vorher diejenigen des ViewParts überschrieben haben sind jetzt mit Annotationen versehen. Es sind dies @PostConstruct *und* @Focus*.* Mit @PostContruct annotierte Methoden werden unmittelbar nach dem Erzeugen der Instanz aufgerufen (Vor dem Schliessen eines Parts werden Methoden die mit @PreDestroy annotiert sind aufgerufen). Mit @Focus annotierte Methoden werden aufgerufen wenn ein Fokus-Ereignis eingetreten ist. Auf weitere Möglichkeiten von Annotationen und Dependency Injection soll bei der Bearbeitung des nächsten Aspektes eingegangen werden.

Fragment und Processor

Die zwei ersten Möglichkeiten – Fragment und Prozessor - basieren beide auf dem LegacyIDE.e4xmi Ansatz. Die gemeinsamen Schritte werden im Folgenden beschrieben.

Entfernen ViewPart aus plugin.xml

Damit unserer View nicht mehr als E3 View angesehen wird müssen wir diese aus dem *plugin.xml* entfernen. Das kann folgendermassen aussehen:

<extension point="org.eclipse.ui.views">  
 <!—view class="ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider.ZugnummerRendererView"  
            id="ch.sbb.rcsd.client.zwl.ui.debug.ZugnummerRendererView" name="Zugnummern"  
           restorable="true">  
      </view-->  
</extension>

plugin.xml entferne aus *org.eclipse.ui.views*

Kopieren der LegacyIDE.e4xmi Datei ins application Plugin

Hierzu wird das oben genannte File aus dem Plugin org.eclipse.platform herauskopiert. In ein paar Dokumentationen wird erwähnt, dass das File im Pluginorg.eclipse.ui.workbench zu finden sei. Ich habe jedoch bei der 4.2.1 und 4.3 Version die LegacyIDE.e4xmi Datei im org.eclipse.platformgefunden.

Das File kopieren wir nun in das Application Plugin, es kommt auf die gleiche Stufe wie das *plugin.xml*. Bei RCS ist das das Plugin ch.sbb.rcsd.client.application.

Anpassen der application Plugin *plugin.xml* Datei

Um der Applikation mitzuteilen, dass sie jetzt die LegacyIDE.e4xmi als Application Definition nehmen soll ist ein Eintrag im plugin.xml vonnöten. Der Eintrag sieht folgendermassen aus:

<extension id="product" point="org.eclipse.core.runtime.products">  
 <product application="ch.sbb.rcsd.client.application.application" name="RCS-Disposition Client">   
   
 <property name="appName" value="RCS-Disposition Client"></property>  
 ………..  
 ………..  
 <property name="**applicationXMI**" value="**ch.sbb.rcsd.client.application/LegacyIDE.e4xmi**"></property>

</product>  
  </extension>

plugin.xml mit dem applicationXMI Eintrag

Nur Fragment Ansatz

Einträge im MANIFEST.MF

Es müssen die folgenden Dependencies (Require-Bundle:) im Manifest des ZugnummerRendererView Plugin eingetragen werden:

org.eclipse.e4.core.di,  
org.eclipse.e4.core.contexts,  
org.eclipse.e4.ui.model.workbench,  
javax.inject,  
org.eclipse.e4.ui.di,  
javax.annotation

Erstellen fragment.e4xmi Datei im *ZugnummerRendererView* Plugin

Im Plugin wo die Klasse ZugnummerRendererView definiert ist muss eine fragment.e4xmi Datei erstellt werden*.* Dasist in diesem Fall das Plugin ch.sbb.rcsd.client.zwl*.* In unserem Beispiel fügen wir dem Application Model als child ein neues TrimmedWindow hinzu



Abbildung fragment.e4xmi add TrimmedWindow

Um dies zu erreichen muss als ElementId die id unserer Applikation eingetragen werden. Da die Id der application im LegacyIDE.e4xmi nicht angepasst wurde lautet die id org.eclipse.e4.legacy.ide.application*.*

Die application bieten nun als Feature (kann man sich als Field vorstellen, was es übrigens im ApplicationModel dann auch ist) children an. Wir wählen das aus. Unten - bei der Combo wo zuerst *Addon* selektiert ist - können wir nun TrimmedWindow auswählen und dieses hinzufügen.

Jetzt kann man diesem TrimmedWindow alles was man so gerne hätte dazu definieren. Wir beschränken uns auf die Controls und fügen – mittels Add child - einen PartStack hinzu. Dem PartStack fügen wir als Child einen Part hinzu.



Abbildung fragment.e4xmi add Part

Dem Part muss jetzt noch mitgeteilt werden wo unser Part den wir hier jetzt darstellen wollen zu finden ist. Dies geschieht über die sogenannte contributionURI. In der Abbildung ist dies die Class URI. Der Pfad der Klasse muss glücklicherweise nicht abgetippt werden. Die Klasse kann über den Find.. Button ausgewählt werden.



Abbildung fragment.e4xmi define contributionURI

Plugin.xml dieses Projektes anpassen

Zu guter Letzt müssen wir noch das plugin.xml unseres ZugnummerRendererView Plugins angepasst werden. Wir teilen mit diesem Schritt der Applikation mit, dass sie erweitert wird und zwar um die Inhalte der fragment.e4xmi Datei. Dies geschieht folgendermassen:

<extension id="ch.sbb.rcsd.client.zwl.fragment" point="org.eclipse.e4.workbench.model">  
 <fragment uri="fragment.e4xmi"> </fragment>  
 </extension>

plugin.xml mit dem fragment.e4xmi Eintrag

Wie oben ersichtlich ist es der Extension point org.eclipse.e4.workbench.modelder uns das ermöglicht. Wir teilen ausschliesslich den Namen des Fragmentes mit.

Nur Processor Ansatz

Einträge im MANIFEST.MF

Es müssen die folgenden Dependencies (Require-Bundle:) im Manifest des ZugnummerRendererView Plugin eingetragen werden:

org.eclipse.e4.core.di,  
org.eclipse.e4.core.contexts,  
org.eclipse.e4.ui.model.workbench,  
javax.inject,  
org.eclipse.e4.ui.di,  
javax.annotation,  
org.eclipse.e4.ui.model.workbench

Processor erstellen

Wir erstellen einen Processor der uns unserer TrimmedWindow programmatisch zusammenstellt. Wir bauen das TrimmedWindow ähnlich wie auf wie wir es über den Fragment Weg gemacht haben.

package ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider;

import javax.inject.Inject;

import org.eclipse.e4.ui.di.Focus;

import org.eclipse.e4.core.di.annotations.Execute;

import org.eclipse.e4.ui.model.application.MApplication;

import org.eclipse.e4.ui.model.application.ui.basic.MBasicFactory;

import org.eclipse.e4.ui.model.application.ui.basic.MPart;

import org.eclipse.e4.ui.model.application.ui.basic.MPartSashContainer;

import org.eclipse.e4.ui.model.application.ui.basic.MTrimmedWindow;

public class ZugnummerRendererViewPartProcessor {

@Inject

protected MApplication app;

@Execute

public void execute() {

MTrimmedWindow window = MBasicFactory.INSTANCE.createTrimmedWindow();

window.setElementId(id);

MPartSashContainer partSashContainer = MBasicFactory.INSTANCE.createPartSashContainer();

window.getChildren().add(partSashContainer);

String partId = baseId + "part"; //$NON-NLS-1$

MPart part = MBasicFactory.INSTANCE.createPart();

part.setElementId(partId);

part.setContributionURI("bundleclass://ch.sbb.rcsd.client.zwl.ui/ch.sbb.rcsd.client.zwl.ui.“ +

„internal.shapeprovider.ZugnummerRendererView");

partSashContainer.getChildren().add(part);

app.getChildren().add(window)

}

}

Abbildung Klasse ZugnummerRendererView E4 Stil

Ein Processor muss eine Methode anbieten die mit @Execute annotiert ist. Diese Methode wird vom Eclipse Framework aufgerufen um die gewünschte Aktion durchzuführen.

In unserem Fall lassen wir uns die Application (MApplication) mittels Dependency Injection setzen. Dies instruieren wir über die Annotation @Inject.

Der Processor fügt dem TrimmedWindow ein PartSashContainer hinzu. Dem PartSashContainerwird ein *Part* hinzugefügt und dem *Part* wird über die contributionURI der Pfad zu unserer Klasse mitgeteilt.

Plugin.xml dieses Projektes anpassen

Zu guter Letzt müssen wir noch das plugin.xml unseres ZugnummerRendererView Plugins angepasst werden. Wir teilen mit diesem Schritt der Applikation mit, dass sie erweitert wird und zwar um die Teile die vom Processor erstellt werden. Dies geschieht folgendermassen:

<extension id="ch.sbb.rcsd.client.zwl.fragment" point="org.eclipse.e4.workbench.model">  
 <processor beforefragment="false"   
 class="ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider.ZugnummerRendererViewPartProcessor">  
 </ processor>  
 </extension>

plugin.xml mit dem processor Eintrag

3.x e4-Bridge

Einträge im MANIFEST.MF

Es müssen die folgenden Dependencies (Require-Bundle:) im Manifest des ZugnummerRendererView Plugin eingetragen werden:

org.eclipse.e4.core.di,  
org.eclipse.e4.core.contexts,  
org.eclipse.e4.tools.compat,  
org.eclipse.e4.ui.di

ViewPart von DIViewPart erben lassen und ViewPart zu POJO machen

Mit diesem Ansatz werden die Views auch zu POJOs. Sie werden aber von einem Objekt dessen Klasse von DIViewPart erbt gewrappt. Dies sieht konkret folgendermassen aus:

package ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider;

import org.eclipse.e4.tools.compat.parts.DIViewPart;

public class ZugnummerRendererView extends DIViewPart<ZugnummerRendererViewWrapped> {

public ZugnummerRendererView()

{

super(ZugnummerRendererViewWrapped.class);

}

}

Abbildung Klasse ZugnummerRendererView Vererbung von DIViewPart

Die Klasse ZugnummerRendererViewWrapped entspricht zu 100% (ausser dem Namen) unserer Klasse ZugnummerRendererView von oben.

package ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider;

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus

public class ZugnummerRendererViewWrapped {

private GraphViewer graphViewer;

@PostConstruct

public void createPartControl(Composite parent) {

final Composite composite = new Composite(parent, SWT.NONE);

composite.setLayout(new FillLayout());

graphViewer = new GraphViewer(composite);

graphViewer.setContentProvider(new ArrayContentProvider());

graphViewer.setInput(new Object[] { new Object() });

graphViewer.setViewPort(new WorldRectangle(0, 0, 100, 100));

graphViewer.setShapeProvider(new IShapeProviderImplementation());

}

@Focus

public void setFocus() {

graphViewer.getControl().setFocus();

}

}

Abbildung Klasse ZugnummerRendererViewWrapped

* + 1. Kritische Betrachtung der Möglichkeiten

Für die RCS Migration kommen auschliesslich die letzten 3 aufgeführten Möglichkeiten in Frage.

Fangen wir mit der letzten – also dem 3.x e4-Bridge an:

Meiner Meinung nach ist dieser Ansatz ausschliesslich für Projekte interessant die irgendeinmal in weiter Zukunft auf E4 migrieren wollen und nach längerfristig auf E3 laufen wollen. So kann sichergestellt werden, dass die „wrapped“ Parts E4 konform sind, der späteren Umstellung steht nichts im Weg.

Wenn man allerdings so schnell wie möglich auf E4 migrieren will stellt dieser Ansatz meiner Meinung nach einen Overhead dar.

Die anderen beiden Ansätze sind sehr interessant. Ich kann hier und heute nicht beurteilen welcher Ansatz – programmatisch oder deklarativ – der bessere ist. Vielleicht wird während der Masterarbeit ein Ansatz das Rennen machen….

* + 1. Test

Getestet wurde dieser Aspekt auschliesslich über das Starten der Applikation. Es hat nach anfänglichen Schwierigkeiten alles geklappt.

1. Aspekt „Adapters / Dependency Injection“
   1. Beschreibung des Aspektes

In dieser Iteration wird die Migration von den Adapter’s und Dependency Injection behandelt. Dependency Injection gibt es in E3 nicht, deshalb wird hier verglichen welche E4 Services den „alten“ E3 Services entsprechen. Die neuen Services können mittels Dependency Injection injiziert werden.

* 1. Adapters

Adapter ermöglichen es eigentlich inkompatible Klassen zueinander zu konvertieren. Dieser Mechanismus wird in Eclipse verwendet um Informationen von Objekten über Plugin-Grenzen hinweg zu teilen. Adapter befähigen vorhandene Klassen sich an andere Schnittstellen anzupassen.

* + 1. Diskussion der Eclipse RCP 4 Lösung

In E4 steht neu die Schnittstelle org.eclipse.e4.core.services.adapter.Adapterzur Verfügung.

package org.eclipse.e4.core.services.adapter;

/\*\*

\* An adapter can adapt an object to the specified type, allowing clients to request domain-specific

\* behavior for an object.

\*/

public abstract class Adapter {

public abstract <T> T adapt(Object element, Class<T> adapterType);

}

Abbildung Klasse Adapter

Die API nimmt zwei Parameter entgegen. Es ist zum einen ein Element vom Typ Object. Dieses Element soll adaptiert werden. Der zweite Parameter ist der adapterType, dieser ist vom Typ *Class*. Das *element* wird also zu einem Object vom Typ adapterType „umgewandelt“. Die Methode gibt nun eine Instanz vom Typ adapterTypezurück oder *null* falls das Object nicht auf den adapterType adaptiert werden kann. Das folgende kleine Beispiel soll zeigen wie das funktioniert.

………

MyNumber n = new MyNumber(42);

MyString s = adapter adapt(n, MyString.class);

System.out.println("Resultat: „ + s.getString());

………

**Console**:

*Resultat: 42*

public class MyNumber{

private Integer number;

public MyNumber (Integer number) {

this. number = number;

}

public Integer getNumber() {

return number;

}

}

public class MyString{

private String aString;

public MyString (String aString) {

this. aString = aString;

}

public String getString(){

return aString;

}

}

Abbildungen Adapterbeispiel

Wir haben die obenstehenden Klassen zur Verfügung und möchten nun eine Instanz vom Typ MyNumberin eine Instanz vom Typ MyString„umwandeln“. Dazu übergeben wir dem Adapter die Instanz von MyNumber*,* sagen dass wir eine Instanz von MyStringerwarten und wenn diese Adaptierung zur Verfügung steht wird die Instanz von MyStringnicht null sein. Wo und wie die Implementierung des Adapters zur Verfügung steht werden wir weiter unten sehen. In diesem Beispiel hat der Adapter die number von MyNumberin einen String für aString von MyStringumgewandelt*.*

Die Adapter API wird als Service angeboten und kann somit über Dependency Injection injiziert werden.

Dies kann über die @Inject Annotation verwirklicht werden wie das folgende Beispiel zeigt:

public class MyExample {

@Inject

protected Adapter adapter;

…..

}

Abbildung Klasse mit Adapter Injection

Die aktuell angebotene Implementation von Adapter als EclipseAdapter ruft entweder- wenn das Object element IAdaptable implementiert – getAdapter(Class adapter) des element’s auf oder holt sich im Hintergrund den E3 AdapterManager und delegiert an diesen weiter. Für die Erfassung welche Objekte von welchen Objekten adaptiert werden können muss bei der Migration nichts Neues implementiert oder deklariert werden. Wie das in E3 deklariert und implementiert wird ist im nächsten Kapitel beschrieben.

* + 1. Vergleich mit Eclipse RCP 3

In E3 holt man sich den AdapterManager über die Platform, dieser ist vom Typ org.eclipse.core.runtime.IAdapterManager. Der Aufruf erfolgt – wie wir im untenstehenden Beispiel sehen – über die statische Methode getAdapterManager().Auf dem AdapterManager ruft man die Methode getAdapter() auf. Dies ist analog zum Adapter in E4, der Unterschied liegt einzig darin, dass die neue Schnittstelle Type Safe ist. Es muss also in E4 nicht mehr gecastet werden.

…..

MyNumber n = new MyNumber(42);

IAdapterManager am = Platform.*getAdapterManager*();

Object adapted = am. getAdapter(n , MyString.class);

MyString s = (MyString) adapted;

…..

Abbildung Klasse mit AdapterManager über Platform

Was jetzt noch fehlt ist die Information welche Objekte in welche Objekte umgewandelt werden können.

Dafür sind zwei Schritte nötig:

1. Implementation einer AdapterFactory
2. Registration dieser AdapterFactoy beim AdapterManager

Implementation einer AdapterFactory

Bei diesem Schritt wird die Schnittstelle org.eclipse.core.runtime.IAdapterFactory implementiert. Die Schnittstelle sieht wie folgt aus:

public interface IAdapterFactory {

public Object getAdapter(Object adaptableObject, Class adapterType);

public Class[] getAdapterList();

}

Abbildung Interface mit IAdapterFactory

Die Methode getAdapter(Object, Class) entspricht der vom IAdapterManager. Die Methode getAdapterList() sagt dem Eclipse Framework welche Klassen von dieser Factory adaptiert werden können. Eine Implementation könnte folgendermassen aussehen:

public class MyNumberAdapterFactory implements IAdapterFactory {

public Object getAdapter(Object adaptableObject, Class adapterType){

if(adapterType == MyString.class){

MyNumber mn = (MyNumber) adaptableObject;

return new MyString(String.*valueOf*(mn.getNumber()));

}

return null;

}

public Class[] getAdapterList(){

return new Class[]{MyString.class };

}

}

Abbildung Implementation von IAdapterFactory

Im obenstehenden Beispiel kann man der Implementation der getAdapterList() Methode entnehmen, dass diese Factory auf den Typ MyString adaptieren kann. In der Methode getAdapter() wird geprüft, ob die adapterType Klasse vom Typ MyString ist, wenn dies zutrifft so wird die Instanz von MyNumber auf eine Instanz von MyString adaptiert. Wenn die adapterType Klasse ein anderer Typ ist wird null zurückgegeben.

Registration dieser AdapterFactoy beim AdapterManager

Die Registration der AdapterFactory kann über zwei Wege passieren. Zum einen besteht die Möglichkeit dies deklarativ im plugin.xml zu erfassen oder man registriert die AdapterFactory programmatisch beim AdapterManager. Die beiden Möglichkeiten sehen so aus:

<extension point="org.eclipse.core.runtime.adapters">

<factory

adaptableType=" ch.mypackage MyNumber."

class="ch.mypackage.MyNumberAdapterFactory">

<adapter type=" ch.mypackage.MyString"></adapter>

</factory>

</extension>

Abbildung Deklarative Registration einer AdapterFactory

…..

IAdapterManager am = Platform.*getAdapterManager*();

MyNumberAdapterFactory af = new MyNumberAdapterFactory();

am.registerAdapters(af, MyNumber.class);

…..

Abbildung Programmatische Registration einer AdapterFactory

* + 1. Vorteile E4

Die Vorteile der E4 Lösung liegen auf der Hand:

1. Injection über DI
2. Typsicherheit beim Adaptieren
3. Keine Codeanpassungen bei AdapterFactories und deren Registrierung

Die Vorteile der Dependency Injection gegenüber dem statischen Holen der AdapterManager Instanz über die Platform liegen auf der Hand. Zum einen ist der Code so um ein Vielfaches besser testbar, es ist einfach möglich im Test-Code den Adapter zu mocken. Auf der anderen Seite muss ich als Benutzer des Adapters nicht wissen welches die richtige Implementation des Adapters ist und wie ich zu dessen Instanz komme. Das übernimmt alles die Eclipse-Runtime-Umgebung.

Die Typsicherheit führt dazu, dass sich schon zur Compilezeit Fehler vermeiden lassen. Mit der E3 Variante ist es zum Beispiel möglich das folgende fehlerhafte Konstrukt zu erzeugen:

…..

IAdapterManager am = Platform.*getAdapterManager*();

Object adapted = am. getAdapter(n , MyString.class);

MyNumber n = (MyNumber) adapted; // ClassCastException zur Runtime

…..

Abbildung AdapterManager ClassCastException

Je früher ein Fehler entdeckt wird (bei E4 zur Compilezeit bei E3 irgendeinmal zur Runtimezeit) desto günstiger ist es diesen zu beheben.

Die aktuell von E4 angebotene Implementation des Adapter’s ist ein Wrapper um den E3 IAdapterManager. Dort wird also im Grossen und Ganzen – ausser wenn das element ein IAdaptable ist – auf den IAdapterManager delegiert. Dieser Umstand führt dazu, dass überhaupt keine Code- oder plugin.xml-Anpassungen bei der Registration und auch bei der Implementation der AdapterFactories vollzogen werden muss.

* + 1. Einschränkungen und Risiken

Der Adapter Service kann wie oben gesehen ganz praktisch über Dependency Injection injiziert werden und der sonstige Code (AdapterFactories und deren Registrierung) muss nicht angepasst werden. Aber genau der letzte Punkt ist unter Umständen ein Problem, denn ich kann mir gut vorstellen, dass dies in Zukunft noch ändern wird und dass man dann die AdapterFactories migrieren muss.

* + 1. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3

Qualitätsmässig sind die beiden Ansätze auf derselben Ebene, da gibt es keine Unterschiede. Der E4 ist wie bereits oben erwähnt wesentlich einfacher testbar und hat somit in dieser Kategorie klar die Nase vorne.

* 1. Adapters: Konkretes Beispiel RCS
     1. Migration

TODO

* + 1. Test

TODO

* 1. Dependency Injection

Als Dependency Injection (von englisch dependency ‚Abhängigkeit‘ und injection ‚Injektion‘) wird in der objektorientierten Programmierung ein Konzept und der Vorgang dazu genannt, bei dem zur Laufzeit die Abhängigkeiten eines Objekts diesem von einem anderen Objekt als Referenzen zur Verfügung gestellt werden. (from wikipedia)

* + 1. Diskussion der Eclipse RCP 4 Lösung

Dependency Injection wurde bei Eclipse erst mit der Version 4 eingeführt. E4 bietet die Möglichkeit Konstruktoren, Felder und Methoden mit @Inject zu annotieren. In diesen Fällen versucht Eclipse jeweils die richtige Instanz zu injizieren. Wir haben in den letzten Kapiteln bereits Beispiele gesehen wie das aussehen kann, hier noch ein weiteres Beispiel:

public class MyExamplePart {

private final Composite parent;

@Inject

protected Adapter adapter;

@Inject

public MyExamplePart(Composite parent){

this.parent = parent;

}

…..

@Inject

public void setCurrentSelection(@Optional

@Named(IServiceConstants.ACTIVE\_SELECTION)

MyExampleSelection selection){

if(selection != null){

…..

}

}

}

Abbildung Klasse Beispiel Dependency Injection

Die Injektion bei der Beispielklasse MyExamplePart läuft folgendermassen ab:

Zuerst wird der Konstruktor aufgerufen, hier wird das Parent-Composite vom Framework mitgegeben. Danach werden alle Werte der mit @Inject annotierten Felder abgefüllt. Es versteht sich von selbst, dass diese nicht als final deklariert sein dürfen. Zu guter Letzt wird noch die mit @Inject annotierte Methode aufgerufen. Wenn ein Wert für ein bestimmtes Objekt ändert, hier zum Beispiel die aktive Selektion so wird der Wert erneut injiziert, also die Methode mit dem neuen Wert aufgerufen. So einfach kann auf das Event selectionChanged reagiert werden.

Die Reihenfolge bei der Injektion ist also: Konstruktor, Felder und dann Methoden.

Die Annotationen @Optional und @Named werden weiter unten behandelt.

Die injizierten Objekte kommen alle aus dem Eclipse Context also aus org.eclipse.e4.core.contexts.IEclipseContext. Die Instanzen sind im Eclipse Context abgelegt, wenn nicht ein bestimmter Name angegeben wird, so sind die Instanzen unter ihrem vollständigen Klassennamen abgelegt. So ist zum Beispiel Composite unter „org.eclipse.swt.Composite“ abgelegt.

Wird ein Objekt eines bestimmten Typs angefordert wird der jeweilige Kontext durchsucht, ob ein Objekt des geforderten Typs enthalten ist. Der Context ist hierarchisch aufgebaut und wird bei einem lookup einer zu injizierenden Instanz von unten (spezifischer) nach oben (allgemeiner) durchsucht. Dies schildert die folgende Abbildung:



Abbildung lookup im Context

Im Workbench Context ist zum Beispiel die aktuelle Selektion oder die Eclipse Preferences abgelegt. Im OSGi Context befinden sich alle OSGi Services.

Was kann alles injiziert werden?

* Alle Objekte die zum Application Model gehören
* Alle Objekte die explizit dem Context hinzugefügt wurden
* Alle Preferences, also Key/Value Paare
* OSGi-Services

Annotationen für die Dependency Injection

Es gibt einige Annotationen die bei der Dependency Injection in E4 (4.3) zum Einsatz kommen (können). Die folgende Tabelle soll eine Übersicht über diese Annotationen schaffen. Es werden auch Annotationen die so nicht unbedingt mit Dependency Injection zu tun haben aufgeführt. Die Tabelle soll einen Überblick über die existierenden Annotationen schaffen und stellt nicht eine komplette Beschreibung dar. Ausführende Beschreibungen sind online zu finden.

|  |  |  |  |
| --- | --- | --- | --- |
| **Annotation** | **Package** | **Plugin** | **Beschreibung** |
| @Inject | javax.inject | javax.inject | Ist im JSR-330 beschrieben, kennzeichnet Konstruktoren, Feldern und Methoden deren Werte bzw. Parameter injiziert werden sollen. |
| @Named | javax.inject | javax.inject | Ist im JSR-330 beschrieben, kann in Kombination mit @Inject eingesetzt werden um bestimmte (mit Namensgebung bestimmt) Instanzen zu injizieren. E4 bietet eine Menge an vorhandenen Services, deren Namen sind zum Beispiel in IServiceConstants abgelegt. @Named wird vor den Parametern |
| @PostConstruct | javax.annotation | javax.annotation | JSR-250. Wird nach der erfolgreichen Instanzierung und Injizierung deren Felder vom Framework aufgerufen. Z.B. für Anmeldung von Listeners |
| @PreDestroy | javax.annotation | javax.annotation | JSR-250. Wird aufgerufen bevor eine Klasse dereferenziert wird. Z.B. für Abmeldung von Listeners |
| @Optional | org.eclipse.e4.core. di.annotations | org.eclipse.e4.core.di | Der Parameter ist optional, wenn nicht vorhanden wird vom Framework null übergeben. |
| @Execute | org.eclipse.e4.core. di.annotations | org.eclipse.e4.core. di | Bei Handler und Prozessoren wird die mit @Execute annotierte Methode aufgerufen. |
| @CanExecute | org.eclipse.e4.core. di.annotations | org.eclipse.e4.core. di | Nur wenn eine Methode, die mit @CanExecute annotiert ist, true zurückgibt wird die mit @Execute annotierte Methode aufgerufe. (return boolean) |
| @Creatable | org.eclipse.e4.core. di.annotations | org.eclipse.e4.core. di | Fügt das mit dieser Annotation markierte Objekt dem Context hinzu. Funktioniert nicht für InnerClasses. |
| @GroupUpdates | org.eclipse.e4.core. di.annotations | org.eclipse.e4.core. di | Mit dieser Annotation annotierte Methoden werden in Batches aufgerufen, also nur zu bestimmten Zeitpunkten, dies kann zum Beispiel sein wenn das System im Ruhezustand ist. |
| @Active | org.eclipse.e4.core. contexts | org.eclipse.e4.core. contexts | Injiziert auf einem Feld oder Parameter das aktuell aktive Element. Dies kann zum Beispiel @Active MPart part, also der aktive Part sein. |
| @Preference | org.eclipse.e4.core. di.extensions | org.eclipse.e4.core. di.extensions | Diese Annoation erlaubt es Eclipse Preferences zu injizieren. |
| @EventTopic | org.eclipse.e4.core. di.extensions | org.eclipse.e4.core. di.extensions | Mit dieser Annotation können Felder oder Parameter markiert werden die auf dem spezifizierten Event Topic Notifikationen erhalten wollen. |
| @UIEventTopic | org.eclipse.e4.ui.di | org.eclipse.e4.ui.di | Dasselbe wie @EventTopic, hier wird aber der Aufruf im Main Thread getätigt. |
| @Focus | org.eclipse.e4.ui.di | org.eclipse.e4.ui.di | Wir aufgerufen wenn das entsprechende Oberflächenelement den Fokus erhält. Hier sollte der Fokus an das zentrale SWT Control weitergegeben werden. |
| @Persist | org.eclipse.e4.ui.di | org.eclipse.e4.ui.di | Mit @Persists annotierte Methoden werden bei als dirty markierten Parts aufgerufen. Mit einer solchen Methode kann der State eines Editors gespeichert werden. Man sollte das Dirty-Flag des *MDirtyable* zurücksetzen. |
| @PersistState | org.eclipse.e4.ui.di | org.eclipse.e4.ui.di | Wird aufgerufen unmittelbar bevor das zugehörige Model-Element des Objekts abgebaut wird. Z.B. der Part einer View. Wird vor @PreDestroy bearbeitet. |
| @AboutToShow | org.eclipse.e4.ui.di | org.eclipse.e4.ui.di | Wird benötigt um der Liste von dynamisch angezeigten Einträgen innerhalb eines *DynamicMenuContributionItem MMenuElements* hinzuzufügen*.* |
| @AboutToHide | org.eclipse.e4.ui.di | org.eclipse.e4.ui.di | Wird benötigt um auf der Liste von dynamisch angezeigten Einträgen innerhalb eines *DynamicMenuContributionItemzu* operieren*.* |
| @PostContextCreate | org.eclipse.e4.ui. workbench.lifecycle | org.eclipse.e4.ui. workbench | Wird aufgerufen nachdem der Context der Applikation aufgerufen wurde. |
| @ProcessAdditions | org.eclipse.e4.ui. workbench.lifecycle | org.eclipse.e4.ui. workbench | Mit dieser Annotation kann das Application Model bearbeitet werden, bevor es an den Renderer übergeben wird. |
| @ProcessRemovals | org.eclipse.e4.ui. workbench.lifecycle | org.eclipse.e4.ui. workbench | Mit dieser Annotation kann das Application Model bearbeitet werden, bevor es an den Renderer übergeben wird. |
| @PreSave | org.eclipse.e4.ui. workbench.lifecycle | org.eclipse.e4.ui. workbench | Methoden die so annotiert sind werden aufgerufen bevor das Application Model gespeichert wird. |

E4 Services

E4 bietet diverse Services an die über Dependency Injection injiziert werden können. Die E4 Entwickler haben sich zum Ziel gesetzt sich auf so wenige Services (auch bekannt unter „The 20 things“) wie möglich zu beschränken. Es wurde einiges an Ballast abgeworfen, die E4 Services zeichnen sich durch schlanke Signaturen und damit einer hohen Wiederverwendbarkeit aus. Leider noch nicht alle Services aus 3.. In der folgenden Tabelle werden diese Services aufgelistet und kurz beschrieben.

|  |  |  |
| --- | --- | --- |
| **Interface** | **Zugehörigkeit** | **Beschreibung** |
| org.eclipse.e4.ui.workbench.modeling.ESelectionService | Top Level Window Context | Erlaubt das Verwalten der aktuellen Selection einer Anwendung. |
| org.eclipse.e4.ui.workbench.modeling.ISaveHandler | Top Level Window Context | Unterstützung zum Speichern von Dirty Parts |
| org.eclipse.e4.ui.workbench.modeling.EPartService | Part Context | Bietet nützliche Methoden für Operationen auf Parts und Perspektiven an. |
| org.eclipse.e4.core.commands.ECommandService | Application Context | Service für Commands |
| org.eclipse.e4.core.commands.EHandlerService | Application Context | Service für Handlers |
| org.eclipse.e4.core.services.adapter.Adapter | Application Context | Adapterservicer, siehe oben |
| org.eclipse.e4.core.services.events.IEventBroker | Application Context | Zentrale Schnittstelle für das Event-Handling |
| org.eclipse.e4.core.services.log.Logger | Application Context | Logger |
| org.eclipse.e4.core.services.translation.TranslationService | Application Context | Für die Internationalisierung der Anwendung, also Übersetzung von Texten. |
| org.eclipse.core.runtime.Platform | Application Context | Registry der installierten Plugins, Adapter Manager, Log, und Authorisierungsinfo Management |
| org.eclipse.core.databinding.observable.Realm | Application Context | TODO |
| org.eclipse.core.runtime.dynamichelpers.IExtensionTracker | Application Context | Hält die Verbindung von Extensions und deren abgeleiteten Objekten |
| org.eclipse.core.runtime.IExtensionRegistry | Application Context | Registry für alle Extension Points und Extensions |
| org.eclipse.e4.ui.css.swt.theme.IThemeEngine | Application Context | Registry für Themes, Möglichkeit zum Ändern des Themes |
| org.eclipse.e4.ui.services.IStylingEngine | Application Context | Styling |
| org.eclipse.e4.ui.workbench.IPresentationEngine | Application Context | Zuständig für die Übersetzung des generischen Workbench Model in Widgets |
| org.eclipse.e4.ui.workbench.IResourceUtilities | Application Context | Laden von ImageDesc |
| org.eclipse.e4.ui.workbench.modeling.EModelService | Application Context | Bietet nützliche Methoden für Operationen auf dem Application Model an. |
| org.eclipse.equinox.app.IApplicationContext | Application Context | Der Kontext der zum Starten einer Applikation benutzt wird |
| org.eclipse.jface.preference.PreferenceManager | Application Context | Verwaltet eine Hierarchie von Preferences und assoziierten Preference Pages |
| org.eclipse.ui.ISharedImages | Application Context | Registry von gemeinsamen Images |
| org.eclipse.ui.progress.IProgressService | Application Context | Support für den Workbench progress Support |

Runtime Data

activePart (IServiceConstants#ACTIVE\_PART)

org.eclipse.e4.core.locale (TranslationService#LOCALE)

org.eclipse.e4.ui.model.application.MApplication

selection (ESelectionService#SELECTION)

Application Parameters

applicationCSS (E4Workbench#CSS\_URI\_ARG)

applicationCSSResources (E4Workbench#CSS\_RESOURCE\_URI\_ARG)

applicationXMI (E4Workbench#XMI\_URI\_ARG)

clearPersistedState (E4Workbench#CLEAR\_PERSISTED\_STATE)

deltaRestore (E4Workbench#DELTA\_RESTORE)

cssTheme (E4Application#THEME\_ID)

initialWorkbenchModelURI (E4Workbench#INITIAL\_WORKBENCH\_MODEL\_URI)

instanceLocation (E4Workbench#INSTANCE\_LOCATION)

persistState (E4Workbench#PERSIST\_STATE)

<http://wiki.eclipse.org/Eclipse4/RCP/EAS/List_of_All_Provided_Services>

Manuelle Möglichkeiten

Erweitern Context

Dem Context kann manuell ein Objekt hinzugefügt werden. Dies passiert folgendermassen:

…..

IEclipseContext context = EclipseContextFactory.*getServiceContext*(Activator.getContext());

context.set (MyString.class, new MyString(„Uhu“));

// Oder

context.set („MyUhuString“, new MyString(„Uhu“));

Abbildung Beispiel Manuelle Erweiterung IEclipseContext

Manuelle Dependency Injection

Einer Klasse können die Dependencies programmatisch, also manuell, gesetzt werden. Und zwar auf diese Weise (Annahme: Der Context wurde wie oben initialisiert):

public class MyExample {

@Inject

protected MyString myString;

@Inject

@Named(„MyUhuString“)

protected MyString myUhuString;

…..

}

…..

MyExample example = ContextInjectionFactory.*make* (MyExample.class, context);

…..

Abbildung Beispiel Manuelle Injection

Eigenen Service definieren

Eclipse 4 oder genauer gesagt OSGi bietet die Möglichkeit eigene Services zu erstellen und zu registrieren. Diese können zur Laufzeit über Dependency Injection injiziert werden. Services bestehen immer aus einer Service Definition und beliebig vielen Implementationen. Die Service Definition ist ein Java-Interface welches die API des Services beschreibt. Durch die Konfiguration der Anwendung kann von den verfügbaren Implementationen eine bestimmte ausgewählt werden. . Somit können Implementationen ausgetauscht werden ohne den Code anpassen zu müssen. Somit können zum Beispiel auch die von Eclipse bereits mitgelieferten Services ersetzt werden.

Service implementieren

Ein eigener Service kann zum Beispiel folgendermassen aussehen:

public interface GreetingService {

void greet();

}

public class GreetingServiceConsole implements GreetingService {

@Override

public void greet() {

System.out.println("Hello to the world");

}

}

Abbildung Beispiel eigener Service

Im Interface spezifiert man die Schnittstelle des Services. Eine mögliche Implementation des GreetingService ist die GreetingServiceConsole-Klasse die ganz simpel etwas auf der Konsole ausgibt.

Service / Komponente registrieren

Um den Service einsetzen zu können muss er nun registriert werden. Dies geschieht in Eclipse über den Menüpunkt „New Component Definition“. Hier kann nun der Ablageort (OSGI-INF, entspricht der Konvention), den Filenamen, den Servicenamen und die Implementation des Services aus. Beim Betätigen des „Finish“ Buttons ist Eclipse so nett und fügt unserem Manifest (MANIFEST.MF) den Ort der Service-Definition (Service-Component: OSGI-INF/greet.xml) hinzu.



Abbildung New Component Definition

Die Übersicht über die Komponente sieht wie folgt aus:



Abbildung Component Definition

Dem Services können über diese Maske Properties gesetzt werden, es kann bestimmt werden, ob der Service sofort aktiviert werden soll und vieles mehr. Siehe dazu TODO

Jetzt muss noch spezifiziert werden welcher Service bzw. welche Services von unserer Komponente zur Verfügung gestellt werden. Dies geschieht im „Services“-Reiter in der Component View über den den Add.. Button bei den „Provided Services“. Es wird hier auf Erklärungen von weiteren Möglichkeiten wie „Referenced Services“ und andere verzichtet, dieses Wissen kann im Internet abgeholt werden.



Abbildung Provided Services

Nach diesen Schritten, also Komponente / Service implementieren und registrieren, kann der Service nun von anderen Klassen benutzt werden.

Ein Service kann auch manuell im BundleContext registriert werden. Dies kann folgendermassen durchgeführt werden:

public void start(BundleContext bundleContext){

bundleContext.registerService(GreetingService.class, new GreetingServiceConsole(), null);

}

Abbildung manuelles Registrieren eines Services

Service injizieren

Die Injizierung wird über die bekannte Annotation @Inject erreicht.

public interface GreetingService {

@Inject

private GreetingService greeter;

public void doSomething()

{

greeter.greet();

}

}

Abbildung Service Injection

* + 1. Vergleich mit Eclipse RCP 3

In E3 werden Services oder sonstige Objekte wie die Instanz der Workbench über statische Methodenaufrufe geholt. Beispiele dafür sind

Platform.getWorkbench();

Platform.getExtensionRegistry();

ResourcePlugin.getWorkspace();

Diese Abhängigkeiten und statische Methodenaufrufe machen den Code schwer testbar und auch schwer wiederverwendbar. Es ist unmöglich die Standarddienste zu verändern oder durch eigene Implementationen auszutauschen. Des Weiteren sind viele nützliche Funktionen der auf die API der Workbench verteilt.

Service Instanzen können über das Interface IServiceLocater geholt werden.

serviceLocator.getService(Class serviceClass);

Das Interface IWorkbenchSite erweitert zum Beispiel das IServiceLocater Interface. In einem ViewPart kommt man über den folgenden Beispiel-Aufruf zu einer Service Instanz:

getSite().getService(MyService.class);

Mit dieser Variante ist man sehr eng ans Framework gekoppelt. Man ist gezwungen ViewPart zu erweitern und verliert dadurch Flexibilität.

Viele Services..

* + 1. Vorteile

Siehe Kapitel 5.6 Teufel..

* + 1. Einschränkungen und Risiken
    2. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3
  1. Dependency Injection: Konkretes Beispiel RCS
     1. Migration
     2. Test

1. Aspekt „Commands / Handler, Menus, Key Bindings“
   1. Beschreibung des Aspektes

Mit diesem Aspekt sollen Commands, Handlers, Menus und auch das Key-Binding - also Shortcuts – behandelt werden. Wie sehen diese in E4 aus, wie in E3 und wie können sie von E3 nach E4 migriert werden.

* + 1. Diskussion der Eclipse RCP 4 Lösung

Um einen ersten Überblick über die Thematik zu schaffen hilft die folgende Abbildung:

DirectMenuItem

DirectToolItem

Handlerklasse

contributionURI

contributionURI

HandledMenuItem

HandledToolItem

Command

command

command

Handler

command

contributionURI

Key Binding

sequence

command

Abbildung Handler, Command, Key Binding und Menu/Toolbar

Jede Menu-Aktion mündet in der Ausführung eines Handlers bzw. einer Handlerklasse. Eine Handlerklasse wird vom Entwickler implementiert, die restlichen Teile können deklarativ im Application Model definiert werden.

Die Methode die bei einer Aktion auf einem Handler ausgeführt werden soll, wird mit der Annotation @Execute markiert. Optional kann eine Methode mit @CanExecute annotiert werden, diese muss ein boolean Wert zurückgeben, nur wenn die Methode true retourniert wird das Framework – beim Rendern des entsprechenden Menus oder Toolbar - den Menupunkt aktivieren.

In der folgenden Abbildung ist exemplarisch ein SaveHandler implementiert.

public class SaveHandler {

@CanExecute

public boolean canExecute(@Named(IServiceConstants.ACTIVE\_PART) MDirtyable dirtyable) {

if (dirtyable == null) {

return false;

}

return dirtyable.isDirty();

}

@Execute

public void execute(IEclipseContext context,

@Named(IServiceConstants.ACTIVE\_SHELL) Shell shell,

@Named(IServiceConstants.ACTIVE\_PART) final MContribution contribution) throws Exception {

final IEclipseContext pmContext = context.createChild();

ProgressMonitorDialog dialog = new ProgressMonitorDialog(shell);

dialog.open();

dialog.run(true, true, new IRunnableWithProgress() {

public void run(IProgressMonitor monitor) throws InvocationTargetException, InterruptedException {

pmContext.set(IProgressMonitor.class.getName(), monitor);

if (contribution != null) {

Object clientObject = contribution.getObject();

ContextInjectionFactory.invoke(clientObject, Persist.class, pmContext, null);

}

}

});

pmContext.dispose();

}

}

Abbildung SaveHandler Beispiel

Es gibt grundsätzlich zwei verschiedene Arten von Menus bzw. ToolItems:

Handlerklassen werden direkt referenziert, dazu gehören

* + DirectMenuItem
  + DirectToolItem

Die Referenz wird über die contributionURI gesetzt. Diese URI wird in der Bundleclass-Notation angegeben. Dies schaut dann zum Beispiel so aus: bundleclass://ch.sbb.address.app/ch.sbb.address.app.handlers.SaveHandler. Handler die so referenziert werden müssen nicht im Application Model deklariert werden.

Handlerklassen werden indirekt über Handler und Commands referenziert, hier sind das

* + HandledMenuItem
  + HandledToolItem

Diese Items referenzieren jeweils ein Command über dessen id. Jetzt kann dem Application Model ein Handler hinzugefügt werden der über die id (Command id) das Command und über die Bundleclass-Notation die zugehörige Handlerklasse referenziert. Nun kann auch ein Key Binding definiert werden und damit ein Command referenziert werden. Damit wird definiert mit welchem Tastaturkommando (oder Shortcut) ein Command ausgelöst werden soll.

Die zweite Art also die über Commands hat ganz klare Vorteile:

* + Nur hier können Tastaturkommandos (Shortcuts) definiert werden
  + Minimiert Redundanzen 🡪 Wartungsfreundlicher. Wenn zum Beispiel ein Handler ersetzt werden muss, so muss man nur die Handlerklasse auf dem Handler der zum Command gehört anpassen. Bei den Direct\*Items müsste man jedes Item anpassen, welches das alte Command referenziert hat.

Jetzt bleibt nur noch die Frage offen wo man die Items, Handlerklassen, Handlers, Commands und Key Bindings überall erfassen kann.

Items

* Main Menu
* Je Part

Separator

Menu

Direct Menu Item

Handled Menu Item

Toolbar

Menu Bar Contributions

Toolbar Contributions

Handlerklassen

Handlerklassen wurden weiter oben bereits behandelt. Da es sich um normale Java-Klassen handelt werden sie in Plugins in den gewünschten Packages abgelegt.

Handlers

Handlers können an vier verschiedenen Orten im Application Model erfasst werden, dies zeigt die folgende Abbildung:



Abbildung Handler im Application Model

Die Möglichkeiten für Handler-Definitionen sind also:

1. Direkt in der Application
2. Innerhalb eines PartDescriptors
3. Innerhalb eines Windows
4. Innerhalb eines Parts

Commands

TODO

Command Categories

TODO

Key Bindings

Key Bindings werden der Application hinzugefügt. Sie werden in „Binding Table“s gruppiert. Ein Key Binding ist definiert durch die Id, die Tastaturkommandos und die Referenz auf ein Command.



Abbildung Key Binding im Application Model

Binding Table

Mit „Binding Table“s werden Key Bindings gruppiert und definiert in welchen „Binding Context“s die Bindings gültig sein sollen. Ein Binding Table Eintrag wird definiert durch die Id und die Referenz auf den Binding Context.



Abbildung Binding Table im Application Model

Binding Contexts

Für alle gilt, dass sie auch in anderen Plugins also Erweiterungen erfasst werden können, dies geschieht jeweils über die fragment.xmi Datei.

* + 1. Vergleich mit Eclipse RCP 3
    2. Vorteile E4
    3. Einschränkungen und Risiken
    4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3
  1. Diskussion der Eclipse RCP 4 Lösung
     1. Vorteile
     2. Vergleich mit Eclipse RCP 3
     3. Einschränkungen und Risiken
     4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3
  2. Konkretes Beispiel RCS
     1. Definition Abnahmekriterien
     2. Migration
     3. Test
  3. Beschreibung des Aspektes „Eigene Extension Points / Eigene Services“
  4. Diskussion der Eclipse RCP 4 Lösung
     1. Vorteile
     2. Vergleich mit Eclipse RCP 3
     3. Einschränkungen und Risiken
     4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3
  5. Konkretes Beispiel RCS
     1. Definition Abnahmekriterien
     2. Migration
     3. Test
  6. Beschreibung des Aspektes „Application Model vs. Advisors“
  7. Diskussion der Eclipse RCP 4 Lösung
     1. Vorteile
     2. Vergleich mit Eclipse RCP 3
     3. Einschränkungen und Risiken
     4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3
  8. Konkretes Beispiel RCS
     1. Definition Abnahmekriterien
     2. Migration
     3. Test

Reflexion