**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: | 0.2 |
| 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 |

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. Definition der Abnahmekriterien 12

3.2.2. Migration 13

3.2.3. Kritische Betrachtung der Möglichkeiten 19

3.2.4. 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

4.2.2. Vorteile 21

4.2.3. Vergleich mit Eclipse RCP 3 21

4.2.4. Einschränkungen und Risiken 21

4.2.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 21

4.3. Adapters: Konkretes Beispiel RCS 21

4.3.1. Definition Abnahmekriterien 21

4.3.2. Migration 21

4.3.3. Test 21

4.4. Dependency Injection: Diskussion der Eclipse RCP 4 Lösung 21

4.4.1. Vorteile 21

4.4.2. Vergleich mit Eclipse RCP 3 21

4.4.3. Einschränkungen und Risiken 21

4.4.4. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3 21

4.5. Dependency Injection: Konkretes Beispiel RCS 21

4.5.1. Definition Abnahmekriterien 21

4.5.2. Migration 21

4.5.3. Test 21

4.6. Beschreibung des Aspektes „Commands / Handler, Menus, Key Bindings“ 23

4.7. Diskussion der Eclipse RCP 4 Lösung 23

4.7.1. Vorteile 23

4.7.2. Vergleich mit Eclipse RCP 3 23

4.7.3. Einschränkungen und Risiken 23

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

4.8. Konkretes Beispiel RCS 23

4.8.1. Definition Abnahmekriterien 23

4.8.2. Migration 23

4.8.3. Test 23

4.9. Beschreibung des Aspektes „Eigene Extension Points / Eigene Services“ 23

4.10. Diskussion der Eclipse RCP 4 Lösung 23

4.10.1. Vorteile 23

4.10.2. Vergleich mit Eclipse RCP 3 23

4.10.3. Einschränkungen und Risiken 23

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

4.11. Konkretes Beispiel RCS 23

4.11.1. Definition Abnahmekriterien 23

4.11.2. Migration 23

4.11.3. Test 23

4.12. Beschreibung des Aspektes „Application Model vs. Advisors“ 23

4.13. Diskussion der Eclipse RCP 4 Lösung 23

4.13.1. Vorteile 23

4.13.2. Vergleich mit Eclipse RCP 3 23

4.13.3. Einschränkungen und Risiken 23

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

4.14. Konkretes Beispiel RCS 23

4.14.1. Definition Abnahmekriterien 24

4.14.2. Migration 24

4.14.3. Test 24

4.15. 24

5. Reflexion 24

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.

Vorbereitung

Im Vorfeld wurde Know

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

In dieser Iteration soll geprüft werden welche Migrationsmöglichkeiten überhaupt 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.platform* herauskopiert 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. Definition der Abnahmekriterien

Damit dieser Aspekt als erfolgreich abgeschlossen gilt wurden vom Betreuer die folgenden Abnahmekriterien definiert:

1. Möglichkeiten aufgeführt und kritisch bewertet und beschrieben
2. Prototyp und Demo
   * 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 injected 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 *createPartControl* und *setFocus* 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 Plugin *org.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.platform* gefunden.

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 dazudefinieren. 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 unserers *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.model* der 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 *PartSashContainer* wird 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 unserers *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% 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 injected 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.Adapter* zur 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 *adapterType* zurück 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 String getFirstName() {

return firstName;

}

}

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 *MyNumber* in eine Instanz vom Typ *MyString* „umwandeln“. Dazu übergeben wir dem Adapter die Instanz von *MyNumber,* sagen dass wir eine Instanz von *MyString* erwarten und wenn diese Adaptierung zur Verfügung steht wird die Instanz von *MyString* nicht 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 *MyNumber* in einen String für *aString* von *MyString* umgewandelt*.*

Die Adapter API wird als Service angeboten und kann somit über Dependency Injection injected 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

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

Probleme

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

Wenn field mit @Inject annotiert wird:

org.eclipse.e4.core.di.InjectionException: Unable to process "ZwlViewPart.site": no actual value was found for the argument "IWorkbenchPartSite".

Wenn in @PostConstruct angegeben:

Keine Fehlermeldung, aber Methode wird nicht aufgerufen

* 1. Beschreibung des Aspektes „Commands / Handler, Menus, Key Bindings“
  2. 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
  3. Konkretes Beispiel RCS
     1. Definition Abnahmekriterien
     2. Migration
     3. Test
  4. Beschreibung des Aspektes „Eigene Extension Points / Eigene Services“
  5. 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
  6. Konkretes Beispiel RCS
     1. Definition Abnahmekriterien
     2. Migration
     3. Test
  7. Beschreibung des Aspektes „Application Model vs. Advisors“
  8. 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
  9. Konkretes Beispiel RCS
     1. Definition Abnahmekriterien
     2. Migration
     3. Test

1. Reflexion