



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:	0.59
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
18.07.2013	0.42	MIRO	Aspekt 3 Commands / Handler, Menus, Key Bindings
25.07.2013	0.43	MIRO	Aspekt 3 Commands / Handler, Menus, Key Bindings
26.07.2013	0.44	MIRO	Aspekt 3 Commands / Handler, Menus, Key Bindings
29.07.2013	0.45	MIRO	Aspekt 3 Commands / Handler, Menus, Key Bindings
31.07.2013	0.46	MIRO	Aspekt 3 Commands / Handler, Menus, Key Bindings
02.08.2013	0.47	MIRO	Aspekt 3 Commands / Handler, Menus, Key Bindings
02.08.2013	0.5	MIRO	Aspekt 4 Eigene Extension Points / Eigene Services
03.08.2013	0.51	MIRO	Aspekt 4 Eigene Extension Points / Eigene Services
08.08.2013	0.52	MIRO	Aufräumarbeiten in bisher behandelten Kapiteln, Aspekt 4 Eigene Extension Points / Eigene Services
09.08.2013	0.53	MIRO	Aufräumarbeiten in bisher behandelten Kapiteln, Aspekt 4 Eigene Extension Points / Eigene Services
14.08.2013	0.54	MIRO	Aufräumarbeiten in bisher behandelten Kapiteln, Aspekt 4 Eigene

			Extension Points / Eigene Services, E3 Geografische Karte
15.08.2013	0.55	MIRO	Geografische Karte, Aufräumarbeiten
16.08.2013	0.56	MIRO	Aufräumarbeiten nach Besprechung mit Marc
20.08.2013	0.57	MIRO	Aufräumarbeiten nach Besprechung mit Marc
26.08.2013	0.58	MIRO	Aufräumarbeiten nach Besprechung mit Marc
28.08.2013	0.59	MIRO	Aufräumarbeiten nach Besprechung mit Marc
29.08.2013	0.591	MIRO	Aufräumarbeiten nach Besprechung mit Marc
30.08.2013	0.592	MIRO	Aufräumarbeiten nach Besprechung mit Marc
31.08.2013	0.593	MIRO	Aufräumarbeiten nach Besprechung mit Marc
02.09.2013	0.594	MIRO	Aufräumarbeiten nach Besprechung mit Marc
03.09.2013	0.595	MIRO	Aufräumarbeiten nach Besprechung mit Marc
04.09.2013	0.596	MIRO	Aufräumarbeiten nach Besprechung mit Marc

Inhaltsverzeichnis

1.	Einleitung	7
1.1.	Zweck des Dokumentes	7
1.2.	Problemstellung	7
2.	Aspekt „Mixing E3 / E4“	8
2.1.	Beschreibung des Aspektes	8
2.1.1.	Diskussion der Eclipse RCP 4 Lösung	8
2.1.2.	Vergleich mit Eclipse RCP 3	10
2.1.3.	Vorteile E4	10
2.1.4.	Einschränkungen und Risiken	10
2.1.5.	Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3	11
2.1.6.	Migrationspfade	11
2.2.	Konkretes Beispiel RCS	13
2.2.1.	Migration	14
2.2.2.	Kritische Betrachtung der Möglichkeiten	20
2.2.3.	Test	Fehler! Textmarke nicht definiert.
2.3.	Zusammenfassung	20
2.3.1.	Gegenüberstellung E3 und E4	20
2.3.1.	Ist Migration machbar?	20
3.	Beispiel RCS für alle folgenden Aspekte	21
3.1.	Geografische Karte UI	21
3.2.	Geografische Karte Code	23
4.	Aspekt „Dependency Injection“	27
4.1.	Beschreibung des Aspektes	27
4.1.1.	Diskussion der Eclipse RCP 4 Lösung	27
4.1.2.	Vergleich mit Eclipse RCP 3	31
4.1.3.	Vorteile	31
4.1.4.	Einschränkungen und Risiken	32
4.1.5.	Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3	32
4.2.	Dependency Injection: Konkretes Beispiel RCS	32
4.2.1.	Migration	32
4.2.2.	Test	Fehler! Textmarke nicht definiert.
4.3.	Zusammenfassung	34



4.3.1.	Gegenüberstellung E3 und E4	34
4.3.2.	Ist Migration machbar?	34
5.	Aspekt „Adapter“	35
5.1.	Beschreibung des Aspektes	35
5.1.1.	Diskussion der Eclipse RCP 4 Lösung	35
5.1.1.	Vergleich mit Eclipse RCP 3	36
5.1.2.	Vorteile E4	38
5.1.3.	Einschränkungen und Risiken	38
5.1.4.	Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3	38
5.2.	Adapters: Konkretes Beispiel RCS	39
5.2.1.	Migration	39
5.2.2.	Test	Fehler! Textmarke nicht definiert.
5.3.	Zusammenfassung	40
5.3.1.	Gegenüberstellung E3 und E4	40
5.3.2.	Ist Migration machbar?	40
6.	Aspekt „Commands / Handler, Menus, Key Bindings“	41
6.1.	Beschreibung des Aspektes	41
6.1.1.	Diskussion der Eclipse RCP 4 Lösung	42
6.1.2.	Vergleich mit Eclipse RCP 3	50
6.1.3.	Vorteile E4	54
6.1.4.	Einschränkungen und Risiken	54
6.1.5.	Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3	54
6.2.	Konkretes Beispiel RCS	55
6.2.1.	Migration	55
6.2.2.	Test	58
6.3.	Zusammenfassung	59
6.3.1.	Gegenüberstellung E3 und E4	59
6.3.2.	Ist Migration machbar?	59
7.	Aspekt „Services“	60
7.1.	Beschreibung des Aspektes	60
7.1.1.	Diskussion der Eclipse RCP 4 Lösung	60
7.1.2.	Vergleich mit Eclipse RCP 3	65
7.1.3.	Vorteile E4	65
7.1.4.	Einschränkungen und Risiken	65



7.1.5.	Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3	65
7.2.	Services: Konkretes Beispiel RCS	66
7.2.1.	Migration	66
7.2.2.	Test	Fehler! Textmarke nicht definiert.
7.3.	Zusammenfassung	69
7.3.1.	Gegenüberstellung E3 und E4	69
7.3.2.	Ist Migration machbar?	69
8.	Aspekt „Eigene Extension Points“	70
8.1.	Beschreibung des Aspektes	70
9.	Reflexion	71
10.	Verzeichnisse / Quellen	72
10.1.	Literaturverzeichnis	72
10.2.	Weitere Quellen	73
10.3.	Abbildungsverzeichnis	74



1. Einleitung

1.1. Zweck des Dokumentes

In diesem Dokument werden die ausgesuchten Aspekte erörtert. Es werden jeweils zuerst der E4 Ansatz, danach der E3 Ansatz und die Migration von E3 nach E4 behandelt. Dazu werden die beiden Ansätze jeweils verglichen. Am Schluss von jedem Aspekt soll jeweils ein konkretes Beispiel aus RCS migriert werden.

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.

1.2. 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.

2. Aspekt „Mixing E3 / E4“

2.1. 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.

2.1.1. Diskussion der Eclipse RCP 4 Lösung

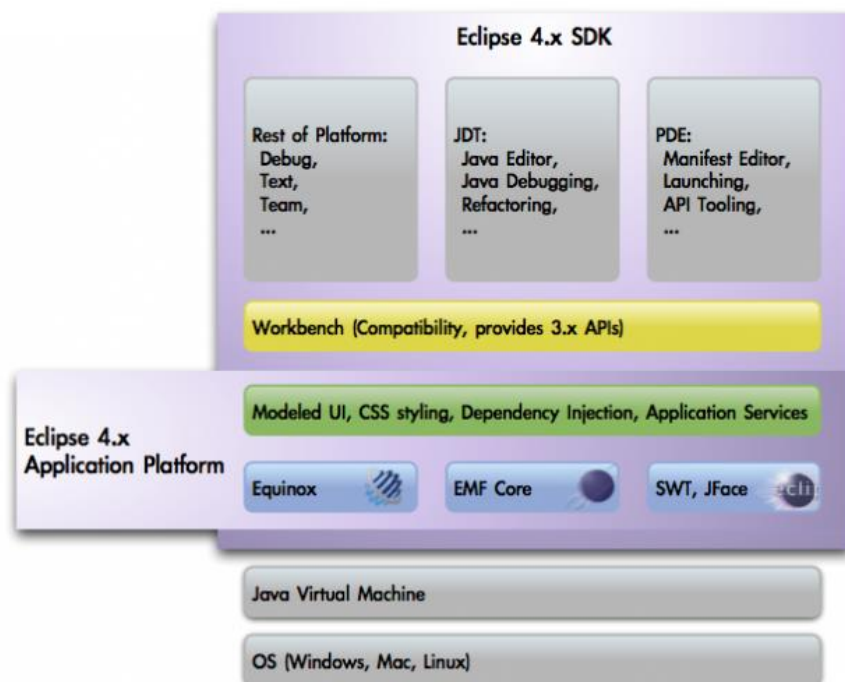


Abbildung 1 Architektur E4

E4 läuft auf einer Java Virtual Machine¹, die Java Version sollte mindestens 6 sein.

¹ http://de.wikipedia.org/wiki/Java_Virtual_Machine



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 ausgelöst 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...

2.1.2. Vergleich mit Eclipse RCP 3

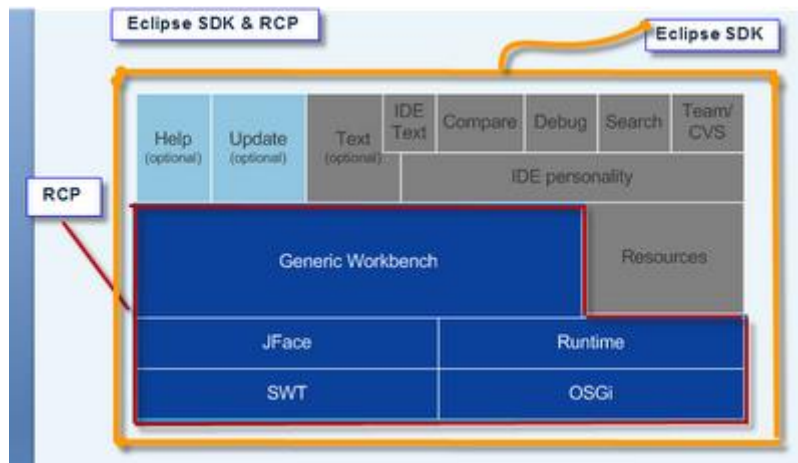


Abbildung 2 Architektur E3

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.

2.1.3. 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 Abwärtskompatibilität 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 werden erleichtert.

2.1.4. Einschränkungen und Risiken

Es gibt Berichte darüber, dass E4 in der momentan aktuellen Version 4.2 noch ziemlich instabil 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.

2.1.5. 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.

2.1.6. Migrationspfade²

<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.

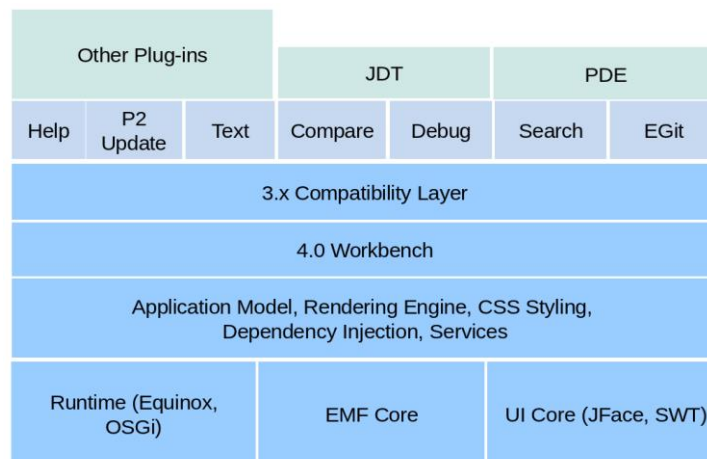


Abbildung 3 Compability Layer E3

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.

² Angelehnt an <http://eclipsesource.com/blogs/2012/06/18/migrating-from-eclipse-3-x-to-eclipse-4-e4/>



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 XML-File – konkret: LegacyIDE.e4xmi - kann aus dem Plugin *org.eclipse.platform* herauskopiert werden. Plugins die nun das Application Model erweitern wollen können dies über Prozessoren oder Fragmente tun.
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.

2.2. Konkretes Beispiel RCS

Die Option 1 und 2 kommen für RCS nicht in Frage. Option 1 wäre ausschliesslich eine E3 Applikation die dank dem Compatibility 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 4 Klasse ZugnummerRendererView E3 Stil

2.2.1. 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 5 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 *@PostConstruct* 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>
```

Abbildung 6 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>
```

Abbildung 7 *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. Das ist in diesem Fall das Plugin *ch.sbb.rcsd.client.zwl*. In unserem Beispiel fügen wir dem Application Model als child ein neues *TrimmedWindow* hinzu

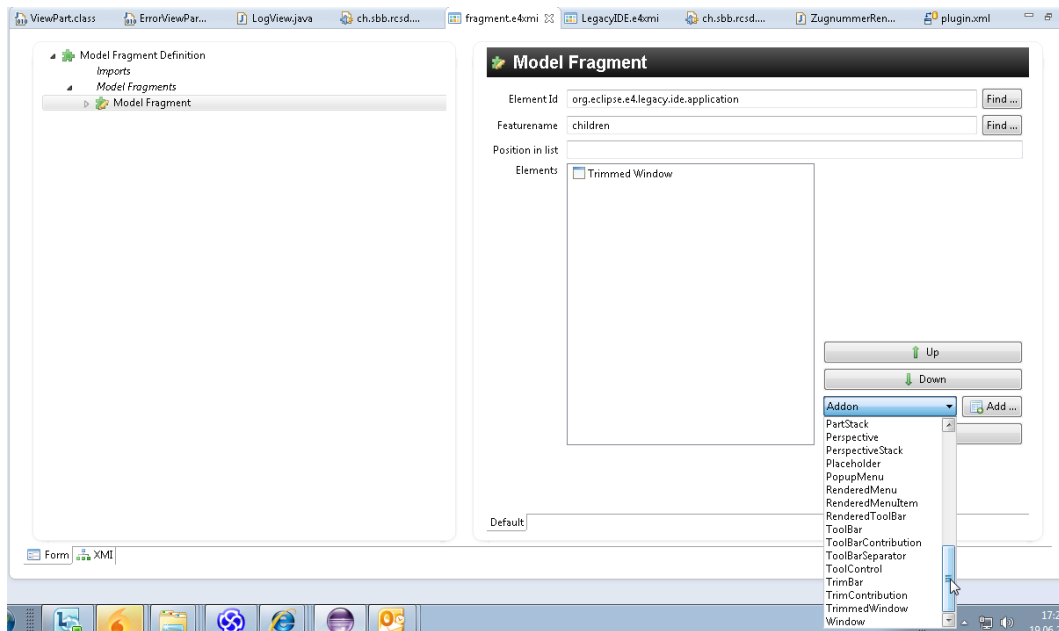


Abbildung 8 fragment.e4xmi add TrimmedWindow

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.e4legacy.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.

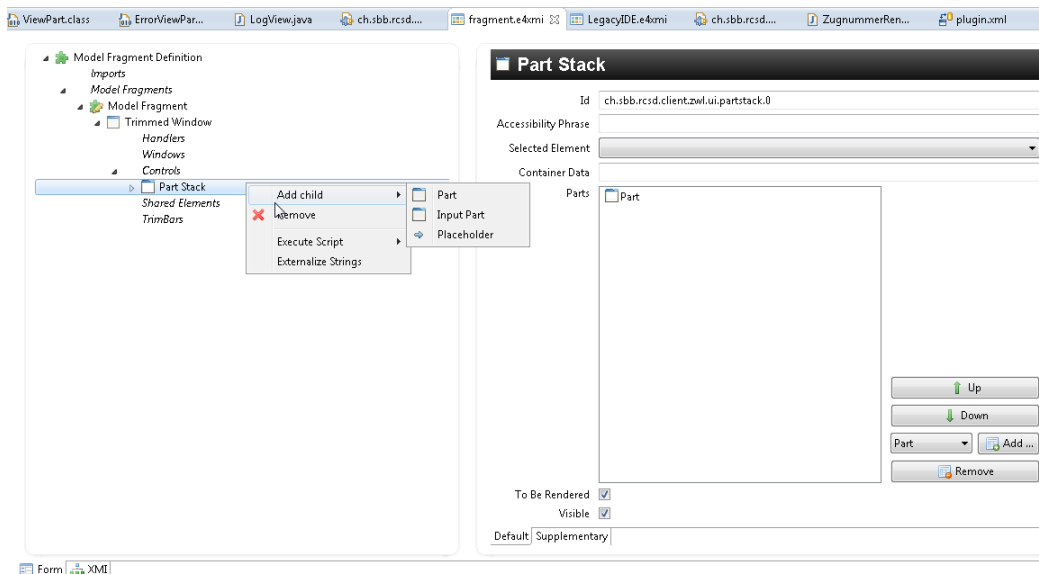


Abbildung 9 fragment.e4xmi add Part

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.

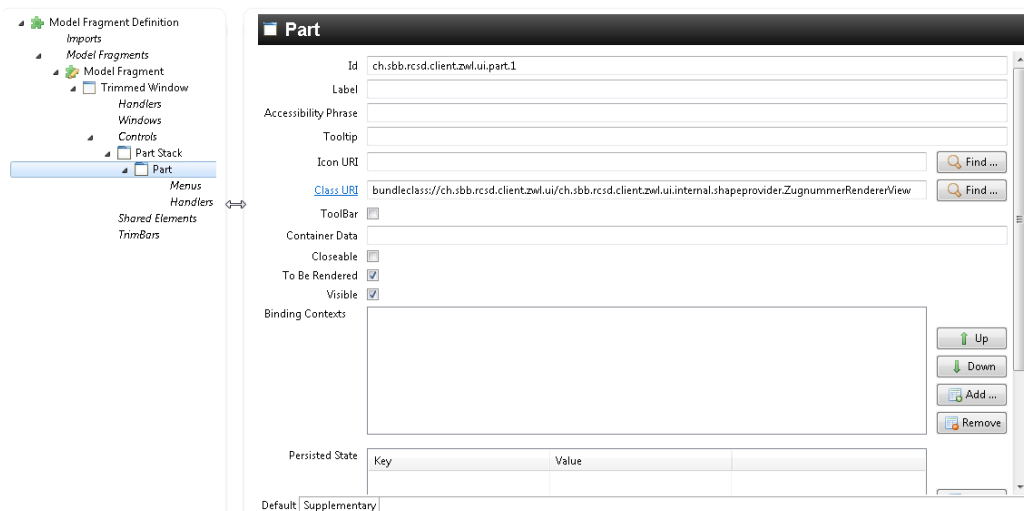


Abbildung 10 fragment.e4xmi define contributionURI

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>
```

Abbildung 11 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 12 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 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>
```

Abbildung 13 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 DViewPart 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 *DViewPart* erbt gewrappt. Dies sieht konkret folgendermassen aus:

```
package ch.sbb.rcsd.client.zwl.ui.internal.shapeprovider;  
  
import org.eclipse.e4.tools.compat.parts.DViewPart;  
  
public class ZugnummerRendererView extends DViewPart<ZugnummerRendererViewWrapped>  
{  
    public ZugnummerRendererView()  
    {  
        super(ZugnummerRendererViewWrapped.class);  
    }  
}
```

Abbildung 14 Klasse *ZugnummerRendererView* Vererbung von *DViewPart*

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 15 Klasse *ZugnummerRendererViewWrapped*

2.2.2. Kritische Betrachtung der Möglichkeiten

Für die RCS Migration kommen ausschliesslich 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....

2.3. Zusammenfassung

2.3.1. Gegenüberstellung E3 und E4

In der folgenden Tabelle werden diverse Themen zu Mixing E3/E4 einander gegenübergestellt. Die Tabelle soll einen Überblick schaffen wo sich E3 und E4 unterscheiden oder wo sie gleich sind.

Thema	E3	E4
Starten der Applikation	plugin.xml, WorkbenchAdvisor, WorkbenchWindowAdvisor, ActionBarAdvisor	plugin.xml, Application Model
Parts erstellen	plugin.xml, Erben von ViewPart	Application Model, POJO
Deklaration von Parts, Menus, etc.	plugin.xml im Application Plugin und plugin.xml in weiteren Plugins	plugin.xml und Application Model (z.B. application.e4xmi oder LegacyIDE.e4xmi) im Application Plugin und plugin.xml und fragment.e4xmi in weiteren Plugins. Möglichkeit von Prozessoren.
Werkzeug für Deklaration	plugin.xml Editor	Application Model Editor
Modeled UI	Nein	Ja
Dependency Injection	Nein	Ja
CSS	Nein	Ja
SWT/JFace	Ja	Ja, es besteht aber Möglichkeit über andere Renderer zum Beispiel JavaFX einzubinden
Runtime	Equinox, JVM	Equinox, JVM (ab Version 6)
TODO mehr? Evtl. Compability Layer?		

2.3.1. Beurteilung

Dieser Aspekt hat gezeigt, dass es durchaus möglich ist E3 und E4 zu mixen. Diese Erkenntnis ist äusserst wichtig, denn wäre dies nicht möglich wäre eine Migration einer grösseren Applikation – wie zum Beispiel RCS – schlichtweg nicht möglich. Der Bearbeitung der nächsten Aspekte steht also nichts im Weg.

3. Beispiel RCS für alle folgenden Aspekte

In diesem Kapitel wird das Beispiel eines ViewParts von RCS vorgestellt. Dieses Beispiel wird für alle folgenden Aspekte als Grundlage dienen. Ein Aspekt gilt als erfolgreich migriert wenn dieser mindestens einmal beispielhaft migriert worden ist.

3.1. Geografische Karte UI

Der folgende Part soll als Beispiel genommen werden:

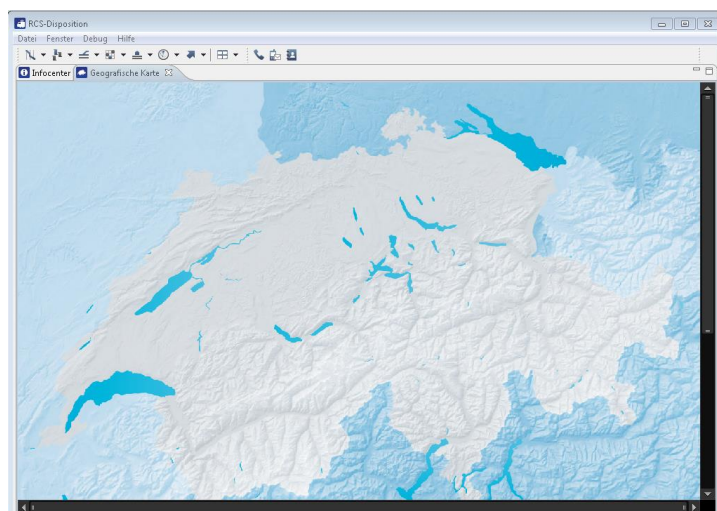


Abbildung 16 Geografische Karte RCS

Es handelt sich hierbei um eine Karte der Schweiz. In RCS ist es möglich eine Zugfahrt oder auch einen Betriebspunkt (z.B. Bahnhof) zu selektieren, das Objekt ist dann auf der Karte sichtbar. Wenn zwei Züge selektiert (rot markiert) sind, dann sieht das folgendermassen aus:

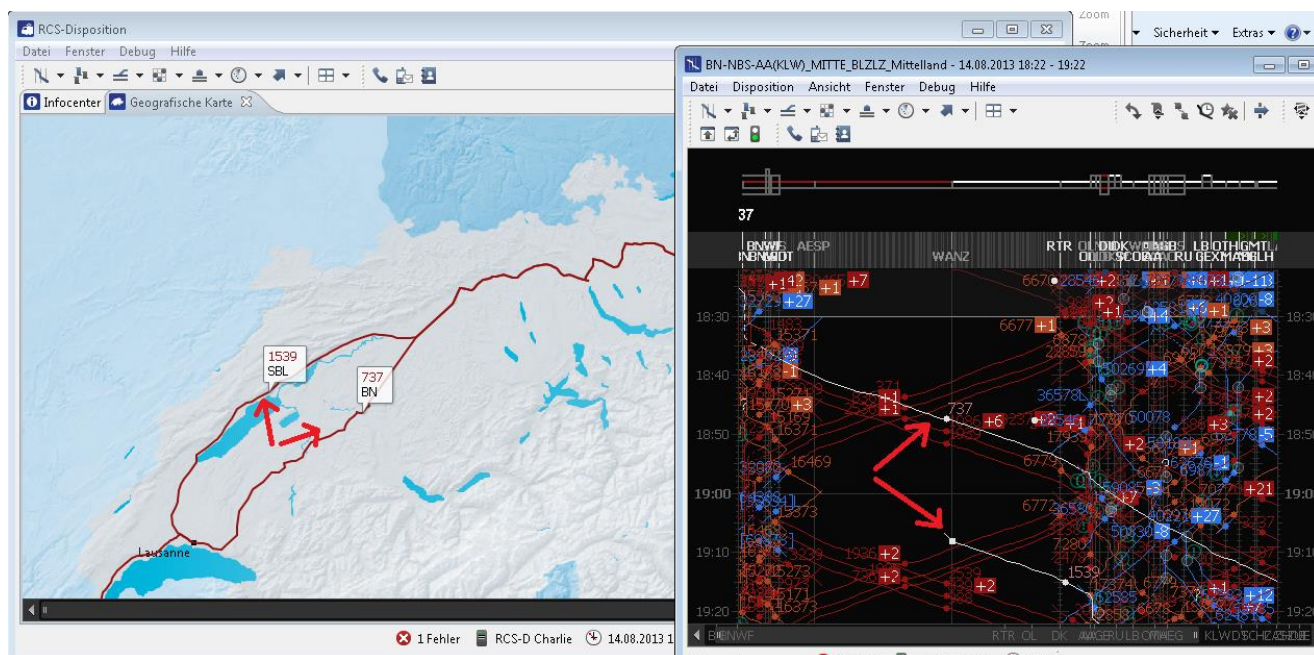


Abbildung 17 Geografische Karte RCS mit zwei selektierten Zügen

Es können - wie erwähnt - auch Bahnhöfe selektiert werden und auch diese werden dann auf der Karte angezeigt.

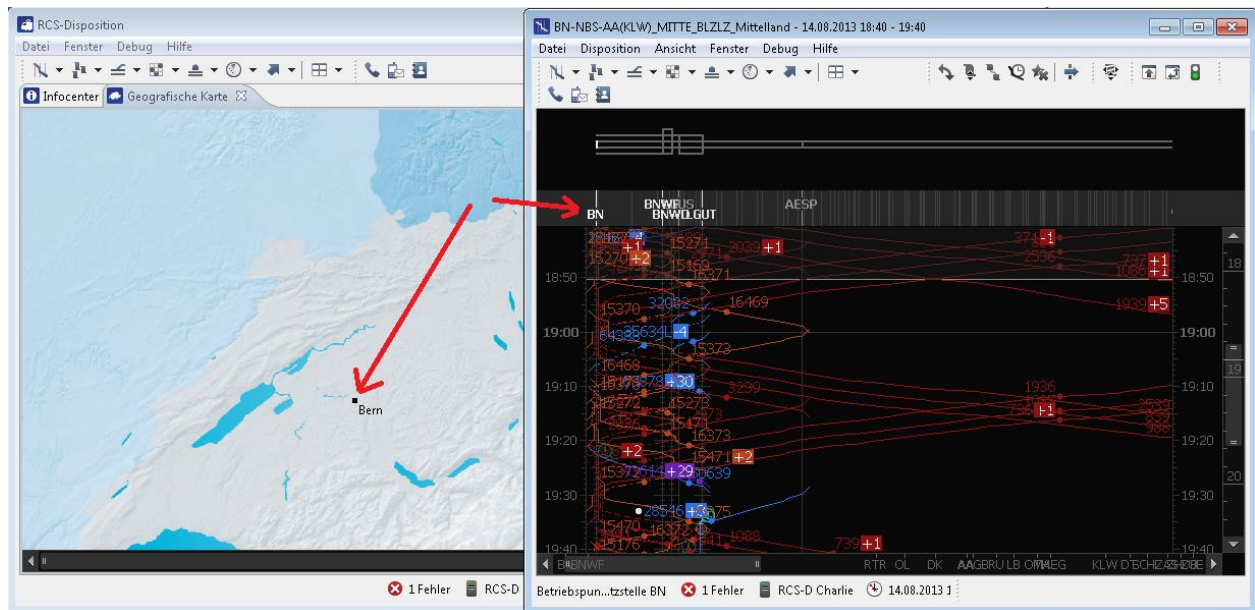


Abbildung 18 Geografische Karte RCS mit selektiertem Bahnhof

Der ViewPart wird über den Menüpunkt Fenster-Geografische Karte geöffnet. Der ViewPart wird als Part im aktiven Window geöffnet und angezeigt.

Es sollen unter Umständen auch Objekte von anderen Typen auf der Karte angezeigt werden können. Diese Möglichkeit soll bestehen bleiben. Wenn ein Objekt selektiert wird, das aktuell (noch) nicht auf der Karte angezeigt werden soll, so bleibt die Karte leer.

3.2. Geografische Karte Code

Die geografische Karte ist ein *ViewPart*. Der Code dazu ist im Folgenden aufgeführt (Die Eclipse Imports wären: *org.eclipse.jface.viewers.ISelection*, *org.eclipse.swt.SWT*, *org.eclipse.swt.layout.GridData*, *import org.eclipse.swt.layout.GridLayout*, *org.eclipse.swt.widgets.Composite*, *org.eclipse.ui.part.ViewPart*). Die Imports der RCS internen Klassen werden nicht aufgeführt. Wir gehen der Einfachheit halber davon aus, dass alle im gleichen Package sind.

```
// Keine imports aufgeführt!
public class MapView extends ViewPart {
    private GraphViewer viewer;
    private MapModel model;
    private WorkbenchSelectionTracker selectionTracker;

    @Override
    public void createPartControl(Composite parent) {
        parent.setBackground(parent.getDisplay().getSystemColor(SWT.COLOR_BLACK));
        parent.setLayout(getLayout());

        model = new MapModel();

        final ViewPortDispatcher dispatcher = new ViewPortDispatcher(parent.getDisplay());
        dispatcher.setViewPort(MapTheme.VIEWPORT);

        viewer = getGraphViewer(parent);
        OpenMagnifierHandler.install(viewer, getSite());
        getSite().setSelectionProvider(viewer);
        dispatcher.addViewPortProvider(viewer);
        dispatcher.addViewPortProvider(new UniformScaler(viewer.getControl()));

        dispatcher.addViewPortProvider(getVerticalScroller(parent), SWT.VERTICAL);
        dispatcher.addViewPortProvider(getHorizontalScroller(parent), SWT.HORIZONTAL);

        selectionTracker = new WorkbenchSelectionTracker(this) {
            @Override
            protected void onSelection(final ISelection selection) {
                onWorkbenchSelectionChanged(selection);
            }
        };
        selectionTracker.setEnabled(true);
    }

    @Override
    public void setFocus() {
        viewer.getControl().setFocus();
    }

    @Override
    public void dispose() {
        selectionTracker.dispose();
        super.dispose();
    }

    private GraphViewer getGraphViewer (Composite parent) {
        GraphViewer viewer = new GraphViewer(parent);
        viewer.setContentProvider(new MapContentProvider ());
        viewer.setShapeProvider(new MapShapeProvider ());
        viewer.setInput(model);
        viewer.getControl().setLayoutData(new GridData(GridData.FILL_BOTH));
        return viewer;
    }
}
// weitere Methoden siehe nächste Seite
```

Abbildung 19 Klasse MapView E3 (1 von 2)

```
// Weitere Methoden
private void onWorkbenchSelectionChanged(final ISelection selection) {
    boolean containsMappables = false;
    for (final Object element : SelectionEval.list(selection, Object.class)) {
        final IMappable mappable = getMappable(element);
        if (mappable != null) {
            if (!containsMappables) {
                model.clear();
                containsMappables = true;
            }
            model.addMappable(mappable);
        }
    }
    if (containsMappables) {
        model.getMappableInitJob(viewer).schedule();
    }
}

private IMappable getMappable(final Object element) {
    final IMappable mappable = SelectionEval.getAdapter(element, IMappable.class);
    if (mappable != null) {
        return mappable;
    }

    final IZuglauf zuglauf = SelectionEval.getAdapter(element, IZuglauf.class);
    if (zuglauf != null) {
        return new MappableZuglauf(zuglauf);
    }

    final IPBetriebspunktInfo betriebspunkt = SelectionEval.getAdapter(element, IPBetriebspunktInfo.class);
    if (betriebspunkt != null) {
        return new MappableBetriebspunkt(betriebspunkt);
    }
    return null;
}

private ViewPortScroller getVerticalScroller(Composite parent) {
    final ViewPortScroller vscroller = new ViewPortScroller(parent, SWT.VERTICAL);
    vscroller.setLayoutData(new GridData(GridData.FILL_VERTICAL));
    vscroller.setScrollRange(MapTheme.VIEWPORT.x, MapTheme.VIEWPORT.width);
    vscroller.setMinimumVisibleSize(MapTheme.VIEWPORT.width / 10);
    vscroller.registerMouseWheel(viewer.getControl());
    return vscroller;
}

private ViewPortScroller getHorizontalScroller(Composite parent) {
    final ViewPortScroller hscroller = new ViewPortScroller(parent, SWT.HORIZONTAL);
    hscroller.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
    hscroller.setScrollRange(MapTheme.VIEWPORT.y, MapTheme.VIEWPORT.height);
    hscroller.setMinimumVisibleSize(MapTheme.VIEWPORT.height / 10);
    hscroller.registerMouseWheel(viewer.getControl());
    return hscroller;
}

private GridLayout getLayout() {
    final GridLayout layout = new GridLayout(2, false);
    layout.horizontalSpacing = 0;
    layout.verticalSpacing = 0;
    layout.marginWidth = 0;
    layout.marginHeight = 0;
    return layout;
}
}
```

Abbildung 20 Klasse MapView E3 (2 von 2)

In der folgenden Tabelle sind die RCS Klassen die in der *MapView* instanziiert, referenziert oder einfach benötigt werden aufgeführt und kurz beschrieben. Um die Migration nachvollziehen zu können ist die Implementation der einzelnen Klassen nicht vonnöten.

Klasse bzw. Interface	Beschreibung
MapModel	Model einer Kartendarstellung
IMappable	Objekte, die sich auf einer geographischen Karte darstellen möchten, können auf dieses Interface adaptieren und eine geeignete Implementierung zur Verfügung stellen.
WorkbenchSelectionTracker	Utility um die globale Selektion in der Workbench zu tracken. Die globale Selektion ist die Selektion des aktiven Workbench-Windows und wird an die {@link #onSelection(ISelection)}-Methode übermittelt. Der Mechanismus kann über das Kommando "ch.sbb.rcsd.client.workbench.autoSelection" aktiviert werden.
GraphViewer	Dieser Viewer stellt Modellelemente auf einem Canvas graphisch dar.
SelectionEval	Hilfsmittel zu Auswerten von Workbench-Selections
MapContentProvider	Content-Provider für Karten
MapShapeProvider	Shape-Provider für die Elemente der Karte
OpenMagnifierHandler	Handler zum Öffnen der Lupe auf einem Canva
ViewPortDispatcher	Instanzen dieser Klasse koppeln beliebige {@link IViewportProvider} und Synchronisieren die dargestellten Bereiche. Damit lässt sich zum Beispiel eine Scrollbar an einen Viewer koppeln. Die Provider lassen sich entweder in beiden Achsen (X/Y) oder nur in einer Achse ankoppeln:
ViewPortScroller	Interaktiver Scroller zur Auswahl des sichtbaren Weltbereichs. Der Scroller kann entweder horizontal oder vertikal arbeiten. Er wird zum Beispiel über einen {@link ViewPortDispatcher} mit einem {@link GraphViewer} verbunden.
UniformScaler	Dieser Viewportprovider passt den sichtbaren Ausschnitt immer an das Seitenverhältnis eines Controls an. Dadurch wird gewährleistet, dass der sichtbare Ausschnitt in beiden Achsen dieselbe Skalierung aufweist.
MappableZuglauf	Implementation von IMappable für Zugläufe
MappableBetriebspunkt	Implementation von IMappable für Betriebspunkte (Bahnhöfe)

In der Methode *createPartControl* wird der *ViewPart* aufgebaut. Zuerst wird die Hintergrundfarbe gesetzt und danach das Layout des Parent-Parts. Danach werden das Model, der GraphViewer und die Scroller instanziiert. Diese werden als ViewPortProvider dem *ViewPortDispatcher* hinzugefügt. Am Schluss wird der *WorkbenchSelectionTracker* instanziiert.

Wenn eine Selektion in der ganzen Applikation ändert, so wird *onWorkbenchSelectionChanged* aufgerufen. In dieser Methode wird die aktuelle Selektion geprüft. Handelt es sich um mehrere selektierte Objekte werden alle behandelt.

Jedes selektierte Objekt wird in *getMappable* überprüft. Wenn das Objekt auf *IMappable*, *IZuglauf* oder *IPBetriebspunktinfo* adaptiert werden kann so kann es auf der Karte angezeigt werden. Im Falle von *IMappable* direkt in den anderen beiden Fällen muss jeweils ein Adapter (*MappableZuglauf* bzw. *MappableBetriebspunkt*) instanziiert werden.

Die Objekte die auf der Karte angezeigt werden können werden dem Model (als Job) hinzugefügt.

In der Methode *setFocus* wird bestimmt welche *Control* den Fokus erhalten soll, in *dispose* wird der SelectionTracker disposed.

Die *MapView* wird wie bereits erwähnt über den Menüpunkt „Geografische Karte“ geöffnet. Dieser wird im *plugin.xml* folgendermassen erstellt:

```
<extension point="org.eclipse.ui.menu">
  <menuContribution locationURI="menu:ch.sbb.rcsd.client.menu.window?after=editor4">
    <command commandId="org.eclipse.ui.views.showView"
      label="%open_mapview_action">
      <parameter name="org.eclipse.ui.views.showView.viewId"
        value="ch.sbb.rcsd.client.map.ui.mapView">
      </parameter>
    </command>
  </menuContribution>
</extension>
```

Abbildung 21 Extension Point MenuContribution MapView E3

Das „Geografische Karte“-Menü wird im Menü *ch.sbb.rcsd.client.menu.window* hinter *editor4* platziert. Das Öffnen des *ViewPart*s geschieht über das vom E3 Framework zur Verfügung gestellte Command *org.eclipse.ui.views.showView*. Dem Command wird über den Parameter mit dem Namen *org.eclipse.ui.views.showView.viewId* mitgeteilt welche View angezeigt werden soll. In diesem Fall ist dies *ch.sbb.rcsd.client.map.ui.mapView*.

Dem E3 Framework wird mit dem untenstehenden Eintrag im *plugin.xml* bekannt gemacht welcher *ViewPart* die Id *ch.sbb.rcsd.client.map.ui.mapView* hat.

```
<extension point="org.eclipse.ui.views">
  <view
    class="ch.sbb.rcsd.client.map.ui.internal.MapView"
    icon="icons/view16/app_karte.png"
    id="ch.sbb.rcsd.client.map.ui.mapView"
    name="%mapview_title">
  </view>
</extension>
```

Abbildung 22 Extension Point View MapView E3

Wir haben nun gesehen wie das Beispiel in E3 implementiert wurde. Das Beispiel setzt sich aus einer Klasse vom Typ *ViewPart* und zwei Einträgen im *plugin.xml* zusammen. Es werden diverse Aspekte des Frameworks genutzt, es sind dies:

- *ViewPart* und dessen Lifecycle-Methoden
- Aktuelle Selektion (*WorkbenchSelectionTracker*)
- Adapter (*SelectionEval* Aufrufe)
- Menü (*plugin.xml*)
- Command (*plugin.xml*)

Wir können also einige Aspekte anhand dieses Beispiels migrieren.

4. Aspekt „Dependency Injection“

4.1. Beschreibung des Aspektes

In dieser Iteration wird 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.

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 (http://de.wikipedia.org/wiki/Dependency_Injection). **TODO Verweis auf guten DI Artikel**

4.1.1. 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 23 Klasse Beispiel Dependency Injection

Die Injektion bei der Beispielsklasse *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:

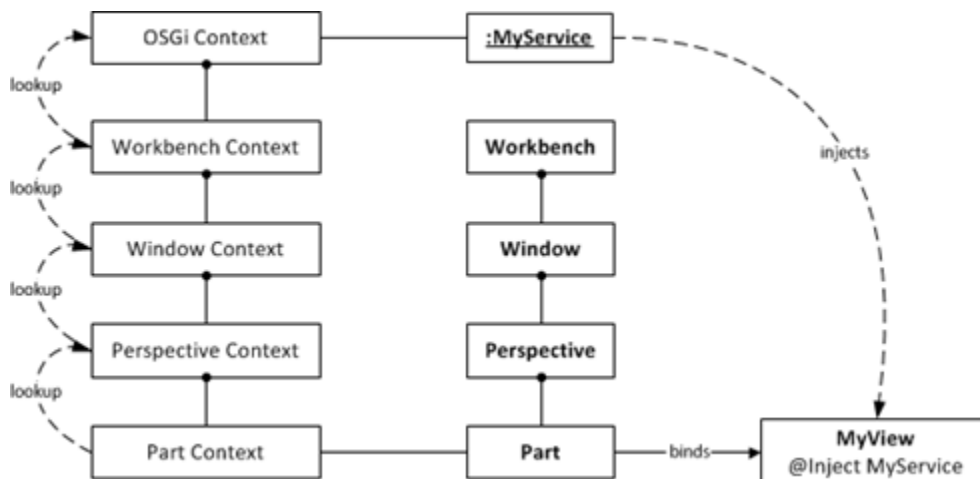


Abbildung 24 lookup im Context

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. Ausführliche 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 Instanziierung 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 aufgerufen. (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 Annotation 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	Wird 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 DynamicMenuContributionItem zu operieren.
@PostContextCreate	org.eclipse.e4.ui	org.eclipse.e4.ui	Wird aufgerufen nachdem der Context der



	workbench.lifecycle	workbench	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.

Kontextzugriff per API

Es ist möglich sich den Context injizieren zu lassen und die gesetzten Objekte aus dem Kontext über API Zugriffe rauszuholen, dies kann folgendermassen ggetätigt werden:

```
public class MyContextExample {  
  
    @PostConstruct  
    public void contextExample(IEclipseContext context){  
        System.out.println(context.get(„MyUhuString“));  
        System.out.println(context.get(MyString.class));  
    }  
}
```

Abbildung 25 Kontextzugriff per API

Dazu muss neben den anderen Dependencies die das Plugin *org.eclipse.e4.core.contexts* in die META-INF-Datei eingetragen werden.

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 26 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 mit den benötigten Objekten befüllt):

```
public class MyExample {  
  
    @Inject  
    protected MyString myString;  
  
    @Inject  
    @Named(„MyUhuString“)  
    protected MyString myUhuString;  
    .....  
}
```

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

Abbildung 27 Beispiel Manuelle Injection

4.1.2. 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.

Die Ermittlung der aktiven Selektion schaut in E3 zum Beispiel so aus:

```
// Keine imports aufgeführt!
public class ExampleView extends ViewPart {

    @Override
    public void createPartControl(Composite parent) {
        getSite().getSelectionProvider().addSelectionChangedListener(
            new ISelectionChangedListener() {
                public void selectionChanged(SelectionChangedEvent event) {
                    if (event.getSelection() instanceof IStructuredSelection) {
                        Object o = ((IStructuredSelection) event.getSelection()).getFirstElement();
                        if (o instanceof MyExampleObject) {
                            onSelectionChanged((MyExampleObject) o);
                        }
                    }
                }
            }
        );
    }

    public void onSelectionChanged(MyExampleObject o) {
        // Etwas machen.....
    }
}
```

Abbildung 28 Beispiel Manuelle Injection

Hier wird der Site des ViewParts ein SelectionListener hinzugefügt. Dies geschieht über `getSite().getSelectionProvider().addSelectionChangedListener()`. Im SelectionListener selbst muss man sich mühsam die Selection aus dem event holen (`event.getSelection()`), diese prüfen ob sie eine Instanz von `IStructuredSelection` ist, diese casten und das erste Element (`selection.getFirstElement()`) rausholen, prüfen ob dieses Element eine Instanz von unserer erwarteten Klasse (hier `MyExampleObject`) ist, die Instanz wieder casten... Es sind relativ viele Schritte notwendig um die Selektion zu erhalten.

4.1.3. Vorteile

Die Verwendung von Dependency Injection und Annotationen bringt eine bessere Wiederverwendbarkeit und Testbarkeit des Codes mit sich. In E3 kamen häufig Singletons zum Einsatz. Will man eine Klasse testen die auf Singletons zugreift so muss man diese auch in der Testumgebung verfügbar machen. Wirkliche Unit-Tests sind dadurch nicht mehr einfach möglich. Mit Dependency Injection definieren Klassen aber umgekehrt genau, was sie benötigen. Nur genau diese Objekte müssen für einen Test bereitgestellt werden. Das E4 Programmiermodell verbessert auch die Wiederverwendbarkeit von UI-Elementen. Wollte man zum Beispiel in Eclipse 3.x eine View innerhalb eines Dialogs verwenden, waren Umbauarbeiten notwendig, da die View das Interface `ViewPart` implementieren musste. In Eclipse 4 sind UI-Elemente, wie beispielsweise Views, reine POJOs und haben meist lediglich eine Abhängigkeit zu SWT. Eine typische Eclipse 4-View kann damit fast beliebig wiederverwendet werden. (Teufel)

Meistens ist in E4 dieselbe Aufgabe mit weitaus weniger Code als in E3 zu lösen. Hier ist zum Beispiel die Ermittlung der aktuellen Selektion zu nennen. Weniger Code heisst auch weniger Fehlermöglichkeiten.

4.1.4. Einschränkungen und Risiken

Die Verwaltung der Abhängigkeiten durch D-Container erschwert es dem Entwickler, sich die tatsächlichen Abhängigkeiten zu verdeutlichen, Implementierungsklassen aufzufinden oder Code-Completion in Eclipse zu nutzen. (Vogel)

4.1.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3

Zur Testbarkeit wurde bereits im Kapitel Vorteile Stellung genommen. Qualitätsmässig sind beide Ansätze gleich einzustufen, es sind keine grossen Unterschiede auszumachen.

4.2. Dependency Injection: Konkretes Beispiel RCS

Wie im Kapitel „Beispiel RCS für alle folgenden Aspekte“ beschrieben wird wo möglich immer derselbe Part migriert. Es ist dies die *MapView*.

4.2.1. Migration

Da „Dependency Injection“ das erste Kapitel nach dem Beispiel-Kapitel ist wird hier gleich auch die Migration des gesamten Parts behandelt. Damit wir nicht bereits Anpassungen an den Menus machen müssen, wählen wir die Migration über 3.x e4-Bridge dazu benötigen wir die folgenden Dependencies im *MANIFEST.MF*:

Dependency	Begründung
org.eclipse.e4.core.di	Enthält Annotationen wie @Optional, @Execute, etc.
org.eclipse.e4.core.contexts	Enthält IEclipseContext der zum Beispiel vom Compat Plugin benötigt wird.
org.eclipse.e4.tools.compat	Enthält DUIViewPart
org.eclipse.e4.ui.di	Enthält Annotationen wie @Focus

Das Plugin *javax.inject* für Annotation *@Inject* und *@Named* wird über „Imported Packages“ importiert → Eintrag: *javax.inject* **TODO Warum, Ädu fragen?**

Die Klasse *MapView* wird neu zum Wrapper unserer eigentlichen Part-Klasse *MapViewWrapped* und erbt neu von *DUIViewPart*:

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

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

public class MapView extends DUIViewPart< MapViewWrapped> {
    public MapView ()
    {
        super(MapViewWrapped.class);
    }
}
```

Abbildung 29 MapView als DUIViewPart

Der Code der E3 *MapView* wird in der neuen Klasse *MapViewWrapped* übernommen und dann nach E4 migriert. Dies schaut dann folgendermassen aus (die unveränderten Methoden gegenüber dem E3 *ViewPart* wurden der Einfachheit halber nicht wieder auscodiert):


```
// Keine imports aufgeföhrt!
public class MapViewWrapped {
    private GraphViewer viewer;
    private MapModel model;

    @Inject
    public void createPartControl(Composite parent) {
        parent.setBackground(parent.getDisplay().getSystemColor(SWT.COLOR_BLACK));
        parent.setLayout(getLayout());

        model = new MapModel();

        final ViewPortDispatcher dispatcher = new ViewPortDispatcher(parent.getDisplay());
        dispatcher.setViewPort(MapTheme.VIEWPORT);

        viewer = getGraphViewer(parent);
        dispatcher.addViewPortProvider(viewer);
        dispatcher.addViewPortProvider(new UniformScaler(viewer.getControl()));

        dispatcher.addViewPortProvider(getVerticalScroller(parent), SWT.VERTICAL);
        dispatcher.addViewPortProvider(getHorizontalScroller(parent), SWT.HORIZONTAL);
    }

    @Focus
    public void setFocus() {
        viewer.getControl().setFocus();
    }

    @PreDestroy
    public void dispose() {
        super.dispose ();
    }

    private GraphViewer getGraphViewer (Composite parent) { // bleibt gleich wie das E3 Original
    }

    @Inject
    void onWorkbenchSelectionChanged(@Optional @Named(IServiceConstants.ACTIVE_SELECTION)final ISelection selection) {
        boolean containsMappables = false;
        for (final Object element : SelectionEval.list(selection, Object.class)) {
            final IMappable mappable = getMappable(element);
            if (mappable != null) {
                if (!containsMappables) {
                    model.clear();
                    containsMappables = true;
                }
                model.addMappable(mappable);
            }
        }
        if (containsMappables) {
            model.getMappableInitJob(viewer).schedule();
        }
    }

    private IMappable getMappable(final Object element) { // bleibt gleich wie das E3 Original
    }
    private ViewPortScroller getVerticalScroller(Composite parent) { // bleibt gleich wie das E3 Original
    }
    private ViewPortScroller getHorizontalScroller(Composite parent) { // bleibt gleich wie das E3 Original
    }
    private GridLayout getLayout() { // bleibt gleich wie das E3 Original }
}
```

Abbildung 30 Klasse MapViewWrapped E4

Um den Part bzw. die ehemalige View zu migrieren sind die folgenden Schritte notwendig:

- Die oben aufgeführten Dependencies in der *META-INF*-Datei eintragen
- *extends ViewPart* durch *DIViewPart* ersetzen
- Neue (POJO-)Klasse *MapViewWrapped* erstellen und dort den Code aus der E3 *MapView* folgendermassen migrieren:
 - o Das Feld *selectionTracker* vom Typ *WorkbenchSelectionTracker* und den dazugehörigen Code entfernen
 - o die Methode *createControl* mit *@Inject* annotieren
 - o die Methode *setFocus* mit *@Focus* annotieren
 - o die Methode *dispose* mit *@PreDestroy* annotieren
 - o die Methode *onWorkbenchSelectionChanged*
 - mit *@Inject* annotieren
 - den Parameter *selection* mit *@Optional* (kann *null* sein) und *@Named(IServiceConstants.ACTIVE_SELECTION)* annotieren
 - Methode für Testzwecke von *private* auf den Default Modifier ändern
 - o den Code *getSite().setSelectionProvider(viewer)* entfernen

Mit diesen Schritten haben wir aus dem E3 ViewPart einen E4 konformen E4 Part geschaffen der durch Dependency Injection das parent-Composite injiziert erhält. Die aktuelle Selektion wird auch via (E4) Dependency Injection - bei jeder Änderung der Selektion – der Methode *onWorkbenchSelectionChanged* „injiziert“.

4.3. Zusammenfassung

4.3.1. Gegenüberstellung E3 und E4

In der folgenden Tabelle werden diverse Themen zur Dependency Injection einander gegenübergestellt. Die Tabelle soll einen Überblick schaffen wo sich E3 und E4 unterscheiden oder wo sie gleich sind.

Thema	E3	E4
Instanzen holen	Mit statischen Methoden-Aufrufe oder übers Framework (z.B. ViewPart-Hierarchie) Aktives Holen der Instanzen	Injektion der Instanzen über Dependency Injection. Passiv (Hollywood Prinzip ³)
Kopplung	Enge Kopplung ans Framework	Lose Kopplung ans Framework
Context	TODO Marc	Hierarchisch aufgebauter Context (Part Context, Perspective Context, Window Context, Workbench Context, OSGi Context) org.eclipse.e4.core.contexts.IEclipseContext

4.3.2. Ist Migration machbar?

Die Migration ist relativ einfach zu bewerkstelligen, da man in E4 praktisch alles injizieren kann. Die statischen Methodenaufrufe können grösstenteils unkompliziert durch injizierte (*@Inject*) Instanzen ersetzt werden. Um dies zu erreichen müssen die Parts jedoch vorher auf E4 (POJOs) migriert – also ins Application Model integriert - werden

³ http://en.wikipedia.org/wiki/Hollywood_principle

5. Aspekt „Adapter“

5.1. Beschreibung des Aspektes

In dieser Iteration wird die Migration von den Adapter's behandelt. 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.

5.1.1. Diskussion der Eclipse RCP 4 Lösung

In E4 steht neu die Schnittstelle `org.eclipse.e4.core.services.adapter.Adapter` aus dem Plugin `org.eclipse.e4.core.services.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 31 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.

<pre>public class MyNumber{ private Integer number; public MyNumber (Integer number) { this.number = number; } public Integer getNumber() { return number; } }</pre>	<pre>public class MyString{ private String aString; public MyString (String aString) { this.aString = aString; } public String getString(){ return aString; } }</pre>	<pre>..... MyNumber n = new MyNumber(42); MyString s = adapter.adapt(n, MyString.class); System.out.println("Resultat: „ + s.getString()); Console: Resultat: 42</pre>
--	---	--

Abbildung 32 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 injiziert werden. Dies kann über die `@Inject` Annotation verwirklicht werden wie das folgende Beispiel zeigt:

```
public class MyExample {

    @Inject
    protected Adapter adapter;

    ....

}
```

Abbildung 33 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.

5.1.1. 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 34 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 AdapterFactory 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 35 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 36 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 AdapterFactory 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 37 Deklarative Registration einer AdapterFactory

```
.....
IAdapterManager am = Platform.getAdapterManager();
MyNumberAdapterFactory af = new MyNumberAdapterFactory();
am.registerAdapters(af, MyNumber.class);
.....
```

Abbildung 38 Programmatische Registration einer AdapterFactory

5.1.2. 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 39 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.

5.1.3. 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.

5.1.4. 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.

5.2. Adapters: Konkretes Beispiel RCS

5.2.1. Migration

Die folgende Abbildung zeigt die migrierte *MapViewWrapped* mit Adapter (die unveränderten Methoden gegenüber dem Part aus dem DI Kapitel wurden der Einfachheit halber nicht wieder auscodiert):

```
// Keine imports aufgeführt!
public class MapViewWrapped {
    private GraphViewer viewer;
    private MapModel model;

    @Inject
    private Adapter adapter;

    @Inject
    public void createPartControl(Composite parent) { // bleibt gleich wie das E4 Original }

    @Focus
    public void setFocus() { // bleibt gleich wie das E4 Original }

    @PreDestroy
    public void dispose() { // bleibt gleich wie das E4 Original }

    private GraphViewer getGraphViewer (Composite parent) { // bleibt gleich wie das E3 Original }

    @Inject
    void onWorkbenchSelectionChanged(@Optional @Named(IServiceConstants.ACTIVE_SELECTION)final ISelection selection) {
        // bleibt gleich wie das E3 Original
    }

    private IMappable getMappable(final Object element) {
        final IMappable mappable = adapter.adapt(element, IMappable.class);
        if (mappable != null) {
            return mappable;
        }

        final IZuglauf zuglauf = adapter.adapt(element, IZuglauf.class);
        if (zuglauf != null) {
            return new MappableZuglauf(zuglauf);
        }

        final IPBetriebspunktInfo betriebspunkt = adapter.adapt(element, IPBetriebspunktInfo.class);
        if (betriebspunkt != null) {
            return new MappableBetriebspunkt(betriebspunkt);
        }
        return null;
    }

    private ViewPortScroller getVerticalScroller(Composite parent) { // bleibt gleich wie das E3 Original }
    private ViewPortScroller getHorizontalScroller(Composite parent) { // bleibt gleich wie das E3 Original }
    private GridLayout getLayout() { // bleibt gleich wie das E3 Original }
}
```

Abbildung 40 MapView mit E4 Adapter

Um die Klasse auf den E4 Adapter zu migrieren werden folgende Schritte benötigt:

- Ein Field vom Typ *org.eclipse.e4.core.services.adapter.Adapter* hinzufügen
- Den statischen Aufruf von *SelectionEval.getAdapter()* ersetzen durch den Aufruf *adapter.adapt()*.

Das war's, diese Migration war äusserst einfach zu bewerkstelligen.



5.3. Zusammenfassung

5.3.1. Gegenüberstellung E3 und E4

In der folgenden Tabelle werden diverse Themen zu Adapter einander gegenübergestellt. Die Tabelle soll einen Überblick schaffen wo sich E3 und E4 unterscheiden oder wo sie gleich sind.

Thema	E3	E4
Adapter(manager) Instanz holen	Statischer Methoden-Aufrufe (Platform.getAdapterManager()) um eine Instanz vom IAdapterManager zu erhalten	Injektion der Adapter-Instanz über DI
Deklaration der AdapterFactory's	Im plugin.xml	Im plugin.xml
Implementation der AdapterFactory's	Klasse die org.eclipse.core.runtime.IAdapterFactory implementiert	Klasse die org.eclipse.core.runtime.IAdapterFactory implementiert
Typsicherheit	Cast notwendig	Kein Cast notwendig

5.3.2. Ist Migration machbar?

Die Migration ist – vorausgesetzt die vorherige Migration, also ViewPart nach POJO etc., ist gemacht – ist sehr einfach durchzuführen. Die Aufrufe von *Platform.getAdapterManager()* können problemlos durch Injizieren von *Adapter*-Instanzen ersetzt werden.

Es stellt sich nur die Frage, ob die Deklaration und sonstige Handhabung der Adapter-Factories in Zukunft so bleiben wird. Denn gegenüber E3 hat sich nichts verändert, die Factories müssen nach wie vor im plugin.xml deklariert werden.

6. Aspekt „Commands / Handler, Menus, Key Bindings“

6.1. 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.

Mit den Commands in E3 kam ein neuer Ansatz zum Einsatz. Die Idee war einen Abstraktions Layer zwischen der Präsentation (UI) und dem Verhalten einzuführen. Ein Command ist keine Präsentations- und auch keine Verhaltens-Implementation, es ist eine abstrakte Repräsentation von einem semantischen Verhalten. Die folgende Abbildung zeigt wie ein Command zwischen Präsentation und Verhalten eingebettet ist:

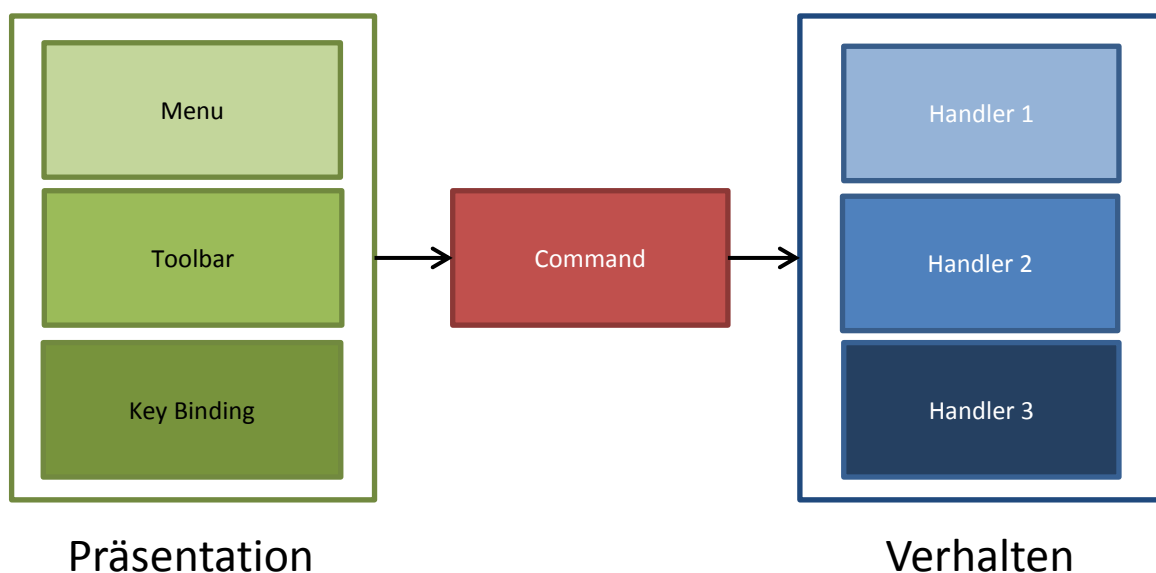


Abbildung 41 Trennung Präsentation und Verhalten mit Commands

Dieses Konzept wird in E3 und auch E4 eingesetzt.

Ein gutes Beispiel für dieses Konzept ist das „Kopieren in die Ablage“ (copy to clipboard). Auf der Präsentationsseite haben wir zum Beispiel ein Menu Item für das Edit Window Menu, einen Toolbar Button, diverse Kontextmenüs und das Ganze soll mittels Tastenkombination Ctrl + C ausgelöst werden können.

Auf der anderen Seite brauchen wir – je nach Kontext - verschiedene Handler-Implementationen für das Kopieren-Command. Ein File Explorer kopiert die Datei-Referenz ins Clipboard, ein Text-Editor kopiert den selektierten Text ins Clipboard und ein Image Viewer kopiert die Bitmap Daten ins Clipboard. (Hoffmann)

6.1.1. Diskussion der Eclipse RCP 4 Lösung

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

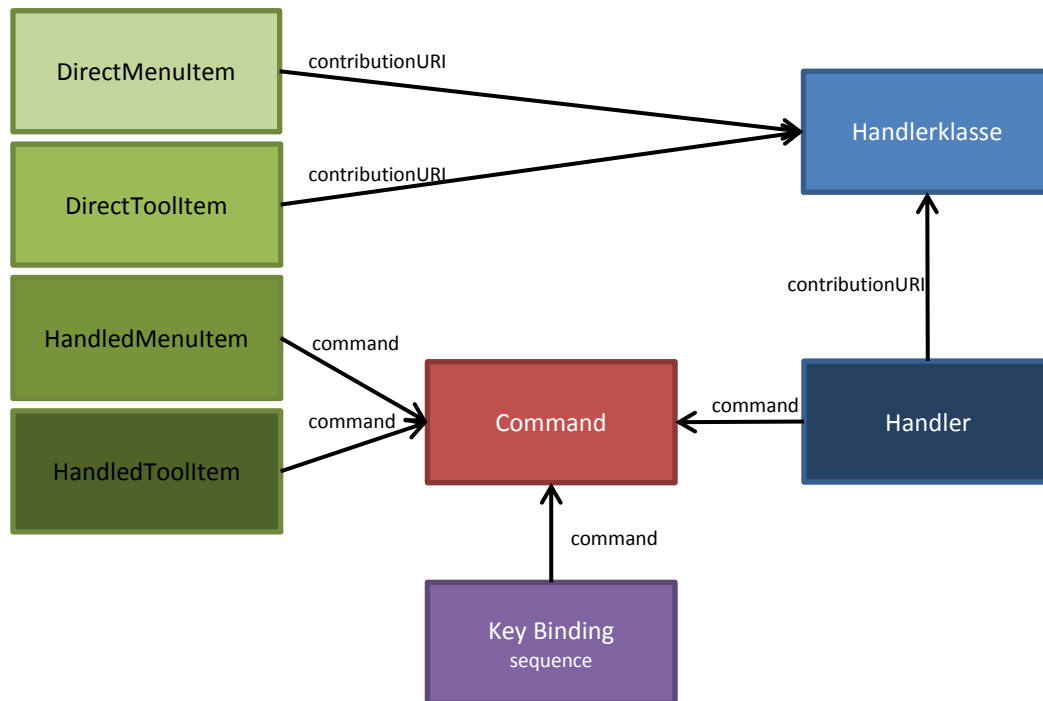


Abbildung 42 Handler, Command, Key Binding und Menu/Toolbar

Commands

Commands werden direkt unterhalb der Application erfasst. Die folgende Tabelle zeigt einen Ausschnitt aus den bereits defaultmässig bestehenden Command Ids auf.

Command	Id
Save	org.eclipse.ui.file.save
Save All	org.eclipse.ui.file.saveAll
Undo	org.eclipse.ui.edit.undo
Redo	org.eclipse.ui.edit.redo
Cut	org.eclipse.ui.edit.cut
Copy	org.eclipse.ui.edit.copy
Paste	org.eclipse.ui.edit.paste
Delete	org.eclipse.ui.edit.delete
Import	org.eclipse.ui.file.import
Export	org.eclipse.ui.file.export
Select All	org.eclipse.ui.edit.selectAll
About	org.eclipse.ui.help.aboutAction
Preferences	org.eclipse.ui.window.preferences
Exit	org.eclipse.ui.file.exit

Weitere Ids können im Interface *org.eclipse.ui.IWorkbenchCommandConstants* gefunden werden.



Command Categories

Eine Command Category kann auch unmittelbar innerhalb der Application im Application Model erfasst werden.

Handler

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. Wenn keine Methode auf der Handlerklasse mit `@CanExecute` annotiert ist, so ist der Menupunkt aktiviert. Es darf maximal eine Methode der Handlerklasse mit `@Execute` und `@CanExecute` annotiert sein.

In der folgenden Abbildung ist exemplarisch ein *TeamInfoHandler* implementiert.

```
public class TeamInfoHandler {  
  
    @CanExecute  
    public boolean canExecute() {  
        return true;  
    }  
  
    @Execute  
    public void execute(@Named(IServiceConstants.ACTIVE_SHELL) Shell shell) throws Exception {  
        TeamInfoDialog dialog = new TeamInfoDialog (shell);  
        dialog.open();  
    }  
}
```

Abbildung 43 E4 TeamInfoHandler Beispiel

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

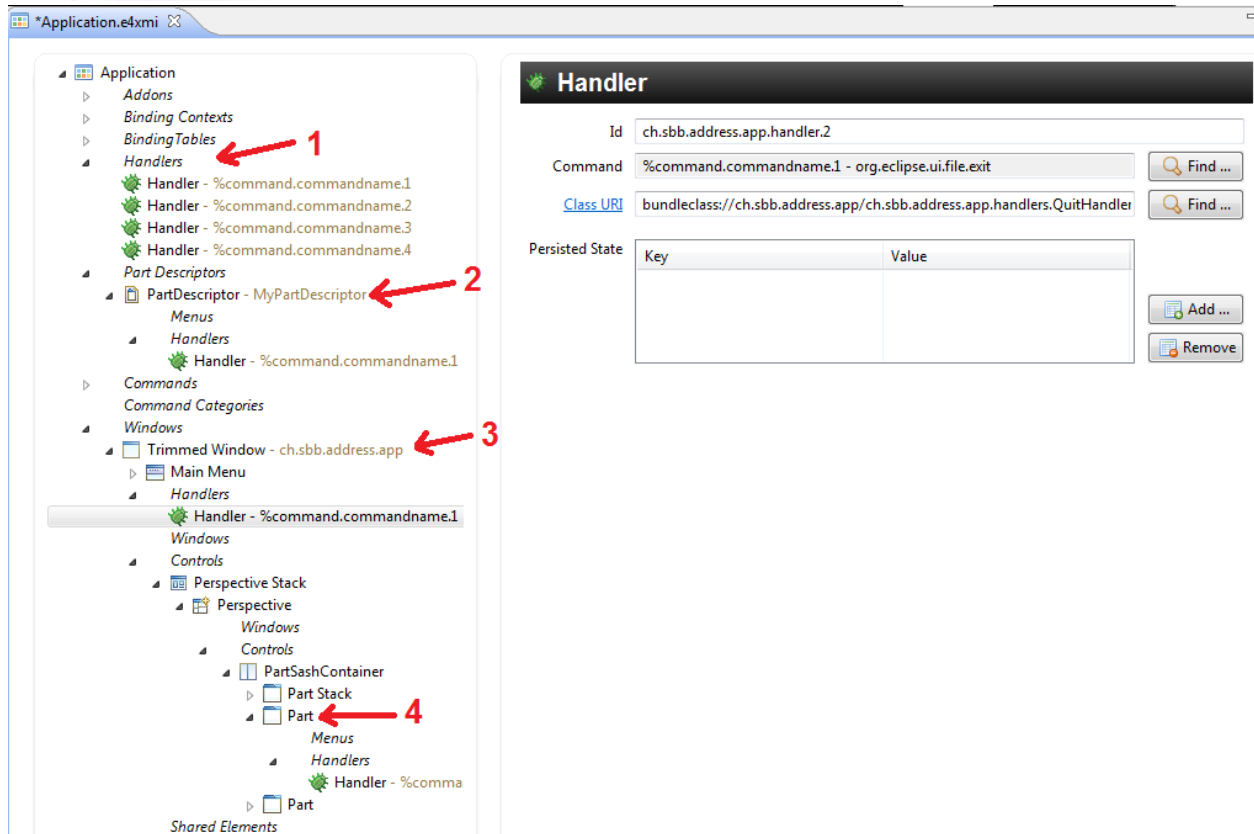


Abbildung 44 Handler im Application Model

Die Möglichkeiten für Handler-Definitionen sind also:

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

Dies ist auch in der Typen Hierarchie ersichtlich:

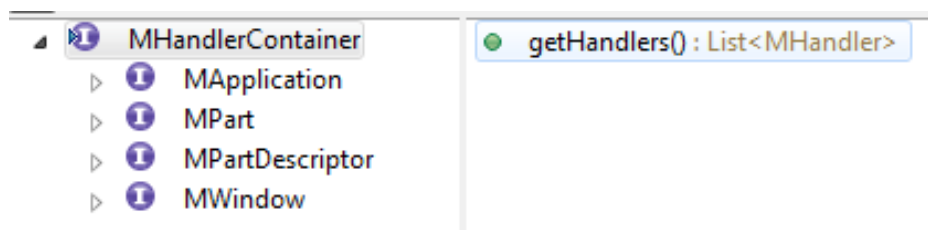


Abbildung 45 MHandlerContainer Hierarchie

Wir sehen, dass *MApplication*, *MPart*, *MPartDescriptor* und auch *MWindow* das Interface *MHandlerContainer* erweitern.

Handlerklassen

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



Menu- und Tool-Items

Es gibt grundsätzlich zwei verschiedene Arten von Menus bzw. Tool-Items:

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.rcsd.client.infopages/ch.sbb.rcsd.client.infopages.internal.team.TeamInfoHandler.

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.

Menu

Menus können an drei Stellen im Application Model definiert werden:

- Als Hauptmenus unterhalb eines Fensters (Window) → Main Menu
- Als Menus von Parts → View Menu, wird zur Laufzeit als Drop-Down-Menu rechts oberhalb des Part angezeigt.
- Als Popup Menus von Parts → Popup Menu

In der untenstehenden Abbildung ist aufgeführt wie man dem Fenster ein Menu hinzufügen kann

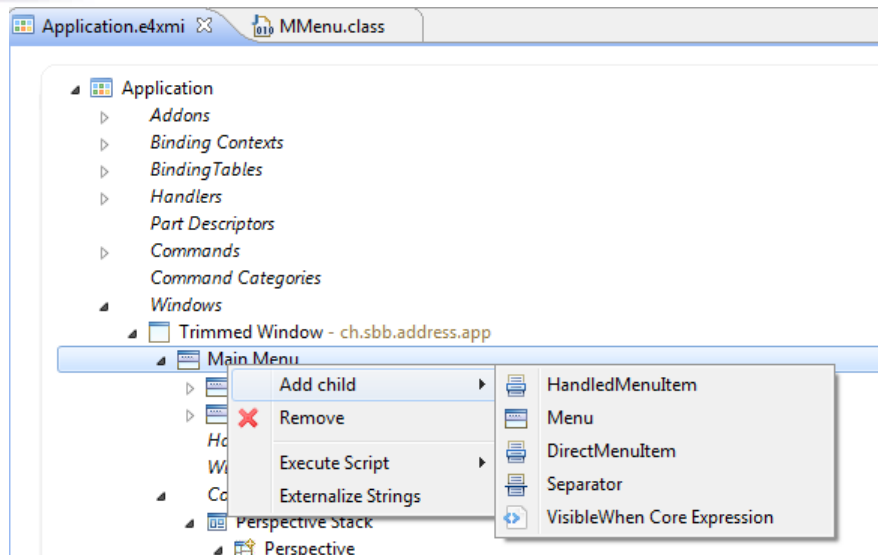


Abbildung 46 Add Menu to Window

Mit der Komponente Menu können mehrere Menus gruppiert werden. Handled- und DirectMenuItem haben wir oben schon kennengelernt. Mit dem Separator können Menüpunkte separiert werden.

Mit einer „VisibleWhen Core Expression“ kann definiert werden wann ein Menu sichtbar sein soll und wann nicht. Die „VisibleWhen Core Expression“ wird über ihre Id referenziert. Die Definition einer solchen Expression erfolgt auf dem aus E3 bekannten Weg im plugin.xml File. Es wird ein Extension Point `org.eclipse.core.expressions.definitions` definiert. Dies wird hier nicht näher beschrieben.

Hier noch die Möglichkeit einem Part Menus hinzuzufügen:

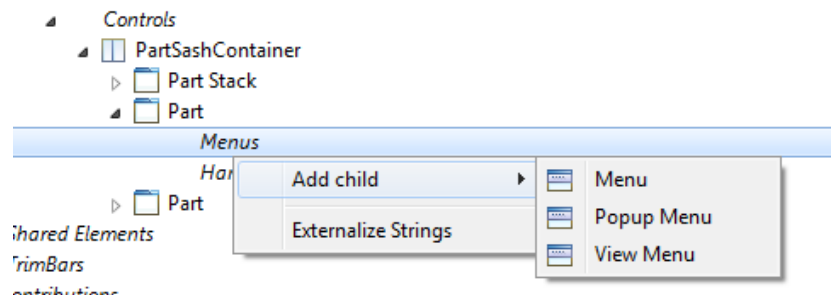


Abbildung 47 Add Menu to Part

Toolbar

Die wichtigsten und am meisten genutzten Funktionen können der Toolbar hinzugefügt werden. Toolbars können im Anwendungsfenster selbst oder in Parts untergebracht werden. Das Fenster muss dann aber zwingend vom Typ `MTrimmedWindow` sein. Um einem Fenster eine Toolbar hinzuzufügen, fügt man unter TrimBars ein „Window Trim“-Element hinzu. Hier kann man jetzt entscheiden ob die Toolbar links, oben, rechts oder unten im Fenster (in einem sogenannten Trim-Bereich) platziert werden soll:

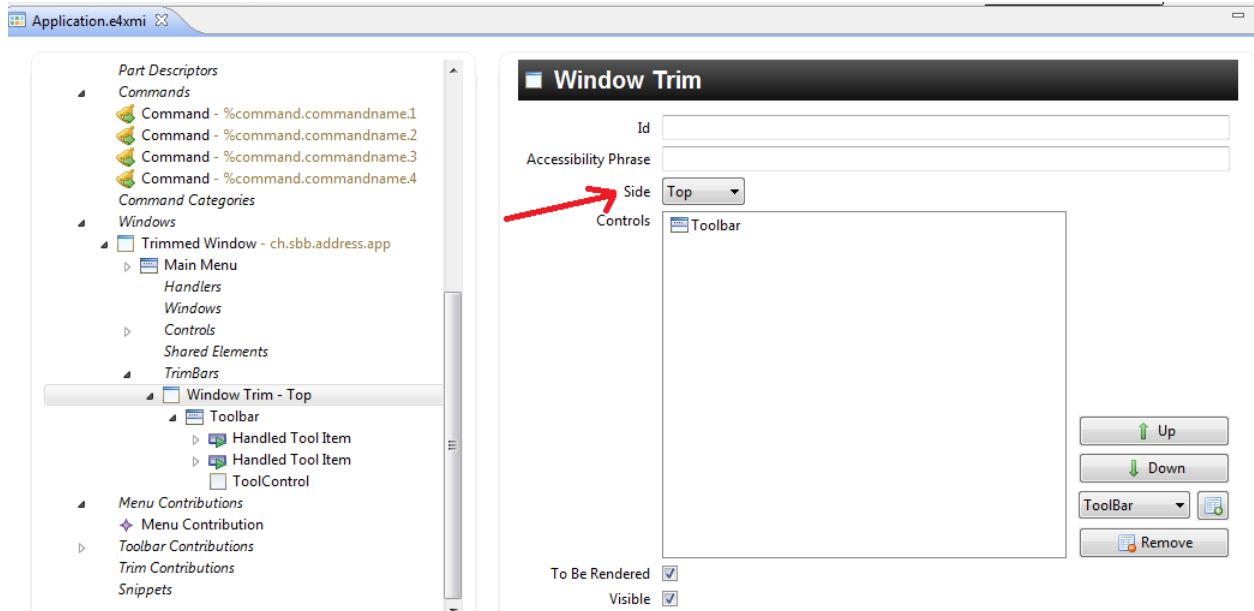


Abbildung 48 Toolbar auf Window

Dem Window Trim Element kann nun eine Toolbar (oder auch eine Toolcontrol) hinzugefügt werden.

Um einem Part eine Toolbar hinzuzufügen muss das Flag „ToolBar“ aktiviert werden:

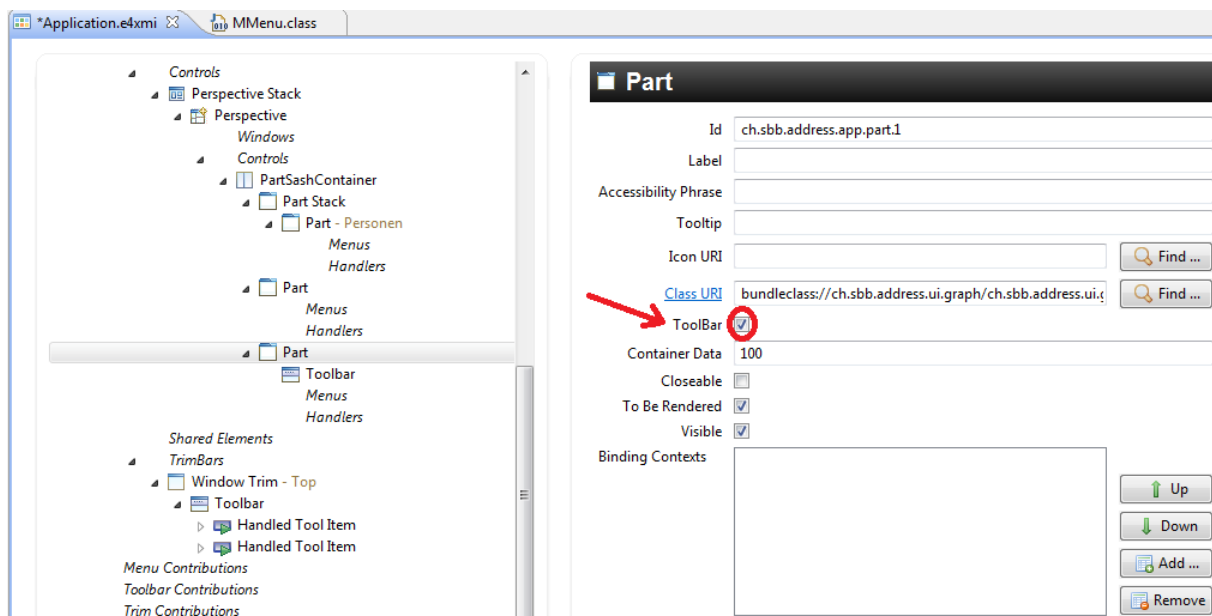


Abbildung 49 Toolbar auf Part

Der Toolbar können die bereits oben erwähnten `Handled-` und `DirectToolItem`'s hinzugefügt werden, weiter ist es möglich einen Separator hinzuzufügen. Dazu kommen noch die `ToolControl`'s, diese ermöglichen es der Toolbar programmatisch Controls hinzuzufügen. Dazu wird eine `ToolControl` Klasse erstellt:

```
public class ExampleToolItem {

    @PostConstruct
    public void createControls(Composite parent) {
        final Composite comp = new Composite(parent, SWT.NONE);
        comp.setLayout(new GridLayout());

        Button button = new Button(comp, SWT.PUSH);
        button.setText("Demo");

        GridDataFactory.fillDefaults().hint(130, 20).applyTo(button);
    }
}
```

Abbildung 50 Beispiel `ToolControl` Klasse

und im Application Model Editor referenziert:

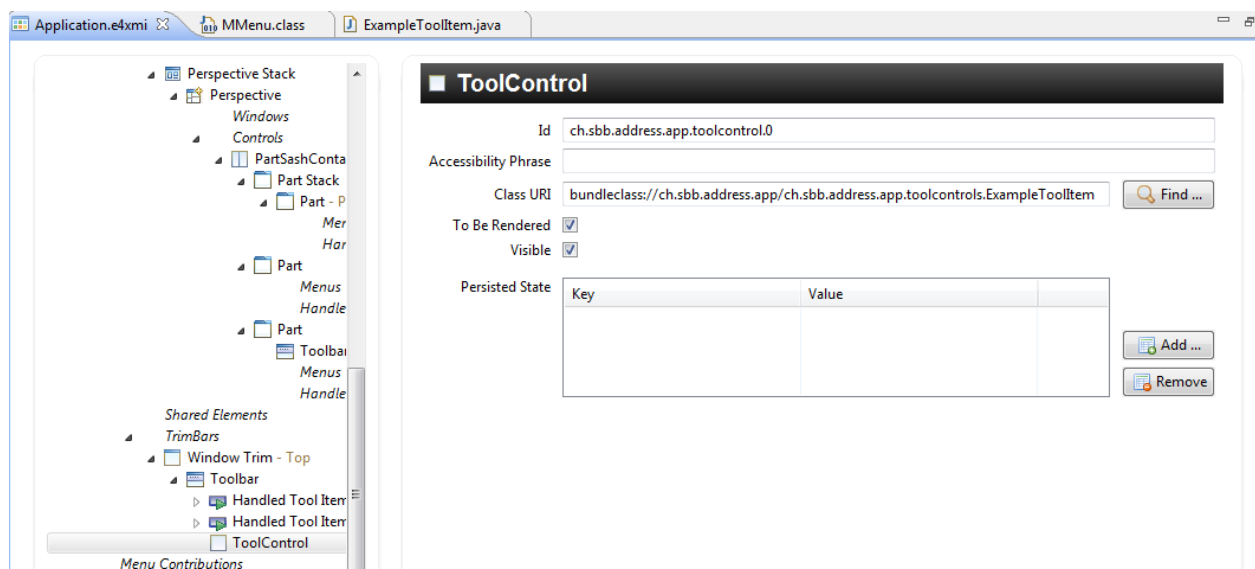


Abbildung 51 Beispiel `ToolControl` im Application Model

Mit dieser Möglichkeit stehen einem ziemlich viele Wege offen für kreative Toolbars.

Menu Contributions

Wenn ein bestimmtes Menu als PopUp- wie auch als Haupt-Menu eingesetzt werden soll, so kann man es - anstatt es zweimal zu definieren – einmal als Menu Contribution definieren.

Toolbar Contributions

Das ist das Pendant zu Menu Contributions für Toolbars.

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.

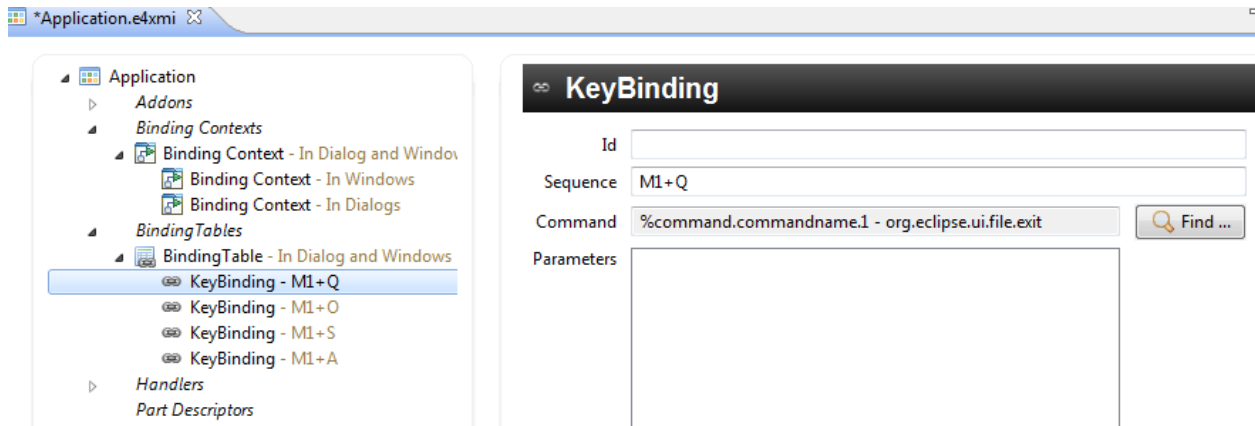


Abbildung 52 Key Binding im Application Model

Die folgende Tabelle zeigt welche Control Keys ausgewählt werden können um auf verschiedenen Betriebssystemen zu funktionieren.

Control Key	Windows und Linux	Mac
M1	Ctrl	Command
M2	Shift	Shift
M3	Alt	Alt
M4	Undefined	Ctrl

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.

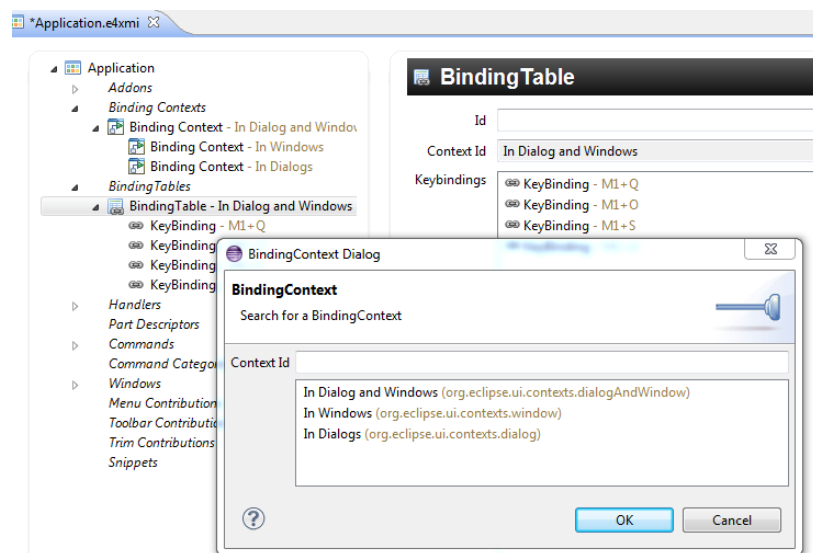


Abbildung 53 Binding Table im Application Model

Binding Contexts

Binding Contexts werden auch unterhalb der Application im Model erfasst. Wie im Kapitel Binding Table erwähnt kann man für jeden Binding Table Eintrag bestimmen in welchem Kontext die Bindings gültig sein sollen. Binding Context's können hierarchisch aufgebaut werden. So ist im folgenden Beispiel der „Dialog und Window“-Kontext aus dem Dialog-Kontext und dem Window-Kontext aufgebaut:

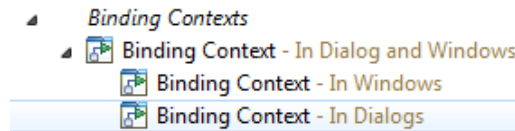


Abbildung 54 E4 Binding Context

Modularisierung

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

Programmatisch

Wie für alle Elemente des Application Models gilt auch hier: Die Elemente können auch programmatisch erzeugt und dem Application Model hinzugefügt werden. Bei grossen Projekten ist es aber übersichtlicher wenn die Elemente deklarativ definiert werden.

6.1.2. Vergleich mit Eclipse RCP 3

E3 bietet zwei Möglichkeiten an um Aktionen hervorzurufen, es sind dies:

- Actions
- Commands

Actions

Actions sollten bereits seit längerer Zeit nicht mehr gebraucht werden, sie entsprechen nicht den Best Practices. Aus diesem Grund und weil RCS kein Actions im Einsatz hat werden sie in diesem Dokument nicht weiter behandelt.

Commands

Commands und Actions sind zwei verschiedene APIs die das gleiche Ziel haben: Der Workbench Funktionalität zur Verfügung stellen. Das Action Framework ist dabei das ältere und über die Zeit gewachsen und es wurde ausgebaut um Funktionalität die anfangs nicht angedacht war. Die grösste Schwäche der Action API ist, dass das UI und die Durchführung einer Action so eng aneinander gekoppelt sind. Um diese und andere Schwächen auszumerzen erschien mit dem Eclipse 3.1 Release das Command API. Mit dem Release 3.3 von Eclipse wurde das API stabilisiert und es wurde zur echten Alternative für das Action Framework.

Command deklarieren

Ein Command wird folgendermassen im *plugin.xml* definiert:

```
<extension point="org.eclipse.ui.commands">
  <category
    id="myplugin.commands.category"
    name="Beispiel Command Category"
    description="Beispiel Command Category"/>
  <command
    id="myplugin.commands.beispiel"
    categoryId="myplugin.command.category"
    name="Beispiel Command"
    description="Beispiel Command"
  />
</extension>
```

Abbildung 55 Deklaration Command

Handlers

Das Verhalten von Commands wird über Handlers spezifiziert. Dies geschieht über den Extension Point *org.eclipse.ui.handlers*. Mit diesem Extension Point können eine oder mehrere Klassen die das Interface *IHandler* implementieren assoziiert werden. Gewisse Methoden werden bereits von der abstrakten Klasse *AbstractHandler* implementiert, im folgenden Beispiel – welches dem vom E4 Handler entspricht – machen wir von dieser abstrakten Klasse Gebrauch:

```
public class TeamInfoHandler extends AbstractHandler {

    @Override
    public boolean isEnabled () {
        return true;
    }

    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        final Shell parent = HandlerUtil.getActiveShell(event);
        TeamInfoDialog dialog = new TeamInfoDialog (parent);
        dialog.open();

        return null;
    }
}
```

Abbildung 56 E3 TeamInfoHandler Klasse

Jetzt muss der Handler noch mit einem Command assoziiert werden. Das passiert im plugin.xml und zwar folgendermassen:

```
<extension point="org.eclipse.ui.handlers">
    <handler
        class="myplugin.handlers.TeamInfoHandler"
        commandId="myplugin.commands.beispiel">
    </handler>
</extension>
```

Abbildung 57 E3 Handler und Command verbinden

Handlers programmatisch erfassen

Dies geschieht über die Services *IHandlerService*.

```
...
final IHandlerService handlerService = (IHandlerService) getSite().getService(IHandlerService.class);
handlerService.activateHandler(„“, new TeamInfoHandler ());
...
```

Abbildung 58 E3 Handler programmatisch aktivieren

Menu Contributions für Commands

Um Menus für im Kontext von Commands zu erstellen wird der Extension Point *org.eclipse.ui.menu* benötigt. Dies kann folgendermassen aussehen:

```
<extension point="org.eclipse.ui.menu">
  <menuContribution
    locationURI="menu:org.eclipse.ui.main.menu?after=additions">
    <menu
      id="myplugin.menus.beispiel"
      label="Beispiel"
      mnemonic="B">
      <menu
        commandId="myplugin.commands.beispiel"
        label="Beispiel Command"
        tooltip="Beispiel Command "/>
      </menu>
    </menu>
  </menuContribution>
</extension>
```

Abbildung 59 Deklaration Menu Contribution

Die locationURI beschreibt wo das Menu hin soll, diese URI ist folgendermassen aufgebaut:

URI Teil	Beispiel
scheme	menu
identifizier	org.eclipse.ui.main.menu
arguments	after=additions

Für scheme sind die folgenden Werte möglich:

Name	Bedeutung
menu	Main Application Menu oder View Pull-down Menu
popup	Pop-Up (Context) Menu in einer View oder einem Editor
toolbar	Main Application Toolbar oder Toolbar

Für den identifizier sind die folgenden Werte möglich:

Name	Bedeutung
org.eclipse.ui.main.menu	Hauptmenu
org.eclipse.ui.main.toolbar	Haupt-Toolbar
org.eclipse.ui.views.ProblemView	Problems View
org.eclipse.ui.views.ContentOutline	Outline View
org.eclipse.ui.popup.any	Irgendwo im Context Menu

Der letzte Aspekt in der URI sind die „Arguments“. Mit den Argumenten kann man angeben an welcher spezifischen Stelle innerhalb des gegebenen Menu's, PopUp's oder der Toolbar eine Contribution platziert werden soll. Das kann sein *before* oder *after* gefolgt von = und der Identifikation eines bestimmten Items innerhalb eines Menu's, PopUp's oder der Toolbar. Der Identifier *additions* kann benutzt werden um die Contribution an der Default Location zu platzieren.

Enabling von Handlers, Commands und UI Contributions

Dieser Punkt ist sehr schön in <http://www.vogella.com/articles/EclipseCommandsAdvanced/article.html> beschrieben.

Bindings

Über den Extension Point *org.eclipse.ui.bindings* können Key Bindings erfasst werden. Das folgende Beispiel zeigt wie das aussehen kann:

```
<extension point="org.eclipse.ui.bindings">
  <key
    sequence="M2+F5"
    commandId="myplugin.commands.beispiel"
    schemeld="org.eclipse.ui.defaultAcceleratorConfiguration"
    contextId="org.eclipse.ui.contexts.dialog"/>
</extension>
```

Abbildung 60 E3 Key Binding und Command verbinden

Das Beispiel zeigt eine Binding Definition für das Command mit der Id *myplugin.commands.beispiel* das mit der Tastenkombination M2 und F5 ausgelöst werden kann. Bei E3 gilt für die „M“-Keys das gleich Mapping wie für E4 (siehe oben). Das Binding ist für das Schema *org.eclipse.ui.defaultAcceleratorConfiguration* gültig. Das ist das Default Workbench Schema und macht dieses Binding in der ganzen Applikation verfügbar und gültig. Es können eigene Schemas kreiert werden, dies wird aber hier nicht weiter beschrieben. Wie das gemacht werden kann ist zum Beispiel unter <http://www.vogella.com/articles/EclipseCommandsKeybindings/article.html> beschrieben. Mit der *ContextId* definiert man in welchem Context ein Key Binding gültig ist. Siehe dazu den Extension Point *org.eclipse.ui.contexts* an. Wenn die *ContextId* nicht spezifiziert ist so gilt der Defaultwert *org.eclipse.ui.contexts.window*.

Contexts

Context's werden über den Extension Point *org.eclipse.ui.contexts* erfasst. Wie in E4 können sie auch hierarchisch aufgebaut werden, dies ist im folgenden Beispiel ersichtlich:

```
<extension point="org.eclipse.ui.contexts">
  <context
    id="org.eclipse.ui.contexts.dialog"
    name="Dialog Context"
    parentId="org.eclipse.ui.contexts.dialogAndWindow"/>
</extension>
```

Abbildung 61 E3 Contexts

Ein Context wird üblicherweise in der *createPartControl* eines *ViewPart's* aktiviert:

```
.....
IContextService cs =(IContextService) getSite().getService(IContextService.class);
IContextActivation ca = cs.activateContext("org.eclipse.ui.contexts.dialogAndWindow");
.....
```

Abbildung 62 E3 Context Activation

Ein Context kann auch programmatisch erstellt werden:

```
.....
Context context = contextService.getContext("org.eclipse.ui.contexts.dialogAndWindow");
if(context.isDefined()) {
    context.define("Dialog Context", "No description", "org.eclipse.ui.contexts.window");
}
.....
```

Abbildung 63 E3 Programmatische Context Erstellung



6.1.3. Vorteile E4

Mit E4 ist es meiner Meinung nach wesentlich einfacher und intuitiver Menus, Commands, Handlers und Key Bindings zu erfassen. Auch die Wartung dieser Komponenten ist wesentlich einfacher, da sie im Application Model relativ schnell zu finden sind. Der Extension Point Ansatz von E3 ist um einiges schwieriger zu handhaben. Ich zum Beispiel muss mir immer wieder die Frage stellen, wie dieser oder jener Extension Point schon wieder heisst und was muss ich da alles erfassen.

Die Handlerklassen von E4 sind nicht so eng ans Framework gekoppelt wie die von E3. E3 Handlerklassen müssen die *AbstractHandler*-Klasse erweitern. Mit diesem Ansatz geht Flexibilität verloren und das das testen wird erschwert.

In E4 gibt es ausschliesslich ein API welches schlank und gut verständlich aufgebaut ist. Mit E3 kommen Commands und die Altlast Actions daher, dieser Umstand kann zu Verwirrungen führen.

6.1.4. Einschränkungen und Risiken

Zu diesem Punkt gibt es nichts zu sagen.

6.1.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3

Meiner Meinung nach ist die Unterbringung der Menus, Commands, Handlers und Key bindings im Application Model sehr gut gelungen. Mit der intuitiven und einfachen Deklaration der Elemente kann die Qualität und Wartbarkeit einer Applikation erhöht werden.

Die Testbarkeit mit JUnit ist bei den E4 Handlerklassen um einiges einfacher, viele benötigte Objekte werden jeweils in der mit *@Execute* annotierten Methode übergeben. Diese können zum Testen einfach gemockt werden. Die Execute-Methode der E3 Handlerklasse ist fest vorgegeben und zwar mit dem Parameter vom Typ *ExecutionEvent*. Hier muss man umständlich den Event mocken und dann zum Beispiel die aktuelle Selection im Event verpacken. Das ist fürs Testen eher mühsam und die Erfahrung zeigt, dass solche Tests dann eher Mal nicht geschrieben werden.

6.2. Konkretes Beispiel RCS

6.2.1. Migration

In diesem Beispiel wollen wir den Menüpunkt der die MapView anzeigt inklusive der Commands und Handlers migrieren. Dazu müssen wir zuerst alle E3 Einträge entfernen. Der erste Eintrag den wir entfernen ist der Menu-Eintrag im *plugin.xml*:

```
<extension point="org.eclipse.ui.menus">
  <!--menuContribution locationURI="menu:ch.sbb.rcsd.client.menu.window?after=editor4">
    <command commandId="org.eclipse.ui.views.showView"
      label="%open_mapview_action">
      <parameter name="org.eclipse.ui.views.showView.viewId"
        value="ch.sbb.rcsd.client.map.ui.mapView">
      </parameter>
    </command>
  </menuContribution-->
</extension>
```

Abbildung 64 Entfernen E3 MapView Menu Eintrag

Im selben Schritt können wir auch gleich den View Eintrag (MapView) entfernen

```
<extension point="org.eclipse.ui.views">
  <!--view
    class="ch.sbb.rcsd.client.map.ui.internal.MapView"
    icon="icons/view16/app_karte.png"
    id="ch.sbb.rcsd.client.map.ui.mapView"
    name="%mapview_title">
  </view-->
</extension>
```

Abbildung 65 Entfernen E3 MapView Eintrag

Die Klasse *MapView* – die ja von *DIViewPart* erbt - kann jetzt gelöscht werden. Desweiteren kann auch die Dependency auf *org.eclipse.e4.tools.compat* aus der MANIFEST.MF-Datei entfernt werden. Die Klasse *MapViewWrapped* benennen wir neu *MapView*.

Neu erstellen wir im Map Plugin im selben Verzeichnis wo auch das *plugin.xml* liegt eine fragment Datei, wir nenne sie *fragment.e4xmi*.

Über den grafischen Editor fügen wir dem Application Model nun unsere gewünschten Einträge hinzu.

Als erstes erfassen wir ein Command, dazu muss man unter Model Fragments zuerst ein neues Model Fragment erstellen. Dieses Fragment soll innerhalb der Application sein und zwar in den Commands:

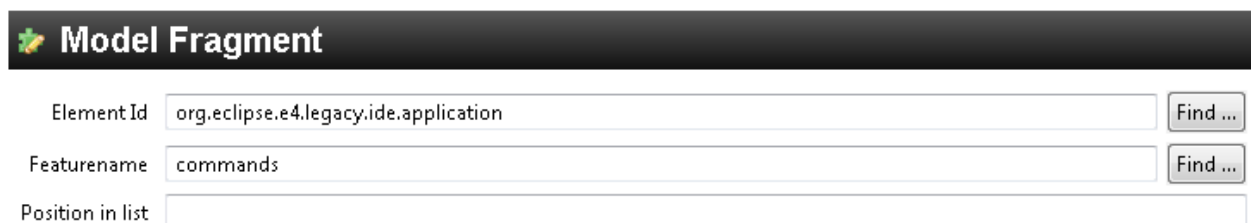


Abbildung 66 E4 MapView Command Model Fragment Eintrag

Nun ist es möglich diesem Model Fragment ein Command hinzuzufügen:

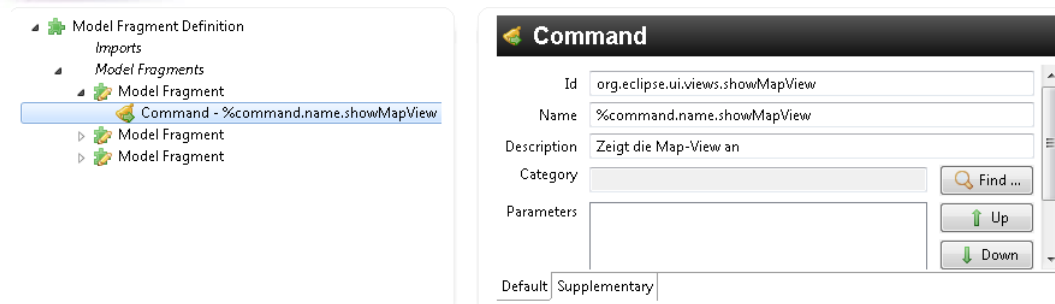


Abbildung 67 E4 MapView Command Eintrag

Als nächstes benötigen wir einen MenuContribution Eintrag, auch hier ist es notwendig zuerst einen Model Fragment Eintrag zu erstellen. Dies ist wieder innerhalb der Application, diese Mal aber das Feature menuContributions:

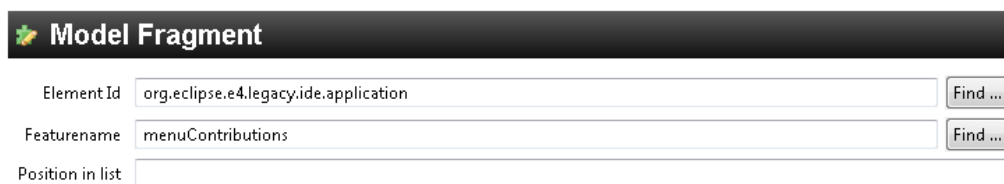


Abbildung 68 E4 MapView Menu Contributions Model Fragment Eintrag

Diesem Fragment fügen wir nun eine MenuContribution hinzu, Die Position und Parent-Id entsprechen dem menuContribution.locationURI Eintrag im *plugin.xml*:

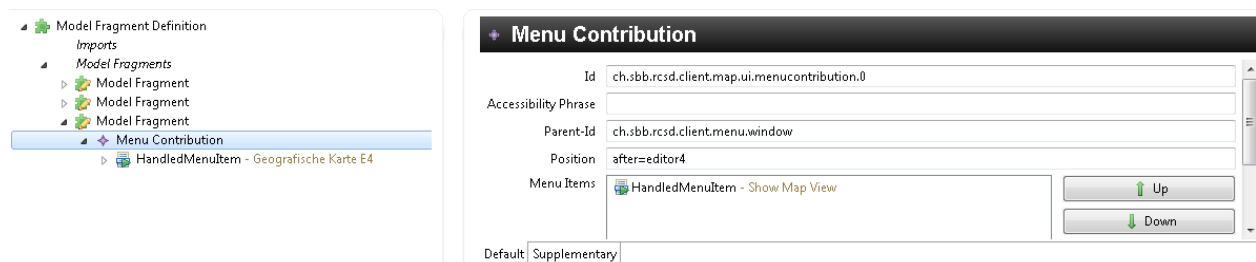


Abbildung 69 E4 MapView Menu Contributions Eintrag

Der MenuContribution fügen wir ein HandledMenuItem hinzu welches auf das vorher erstellte Command verweist:

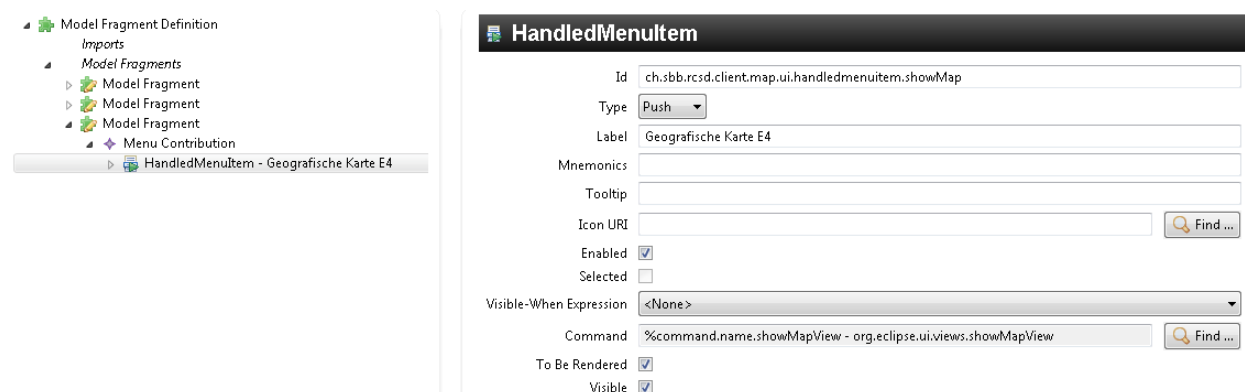


Abbildung 70 E4 MapView Menu Item Eintrag

Als letzten Eintrag im Application-Model erfassen wir noch einen Handler. Auch dazu wird zuerst ein Model Fragment für die Application erstellt, das Feature hier ist handlers:

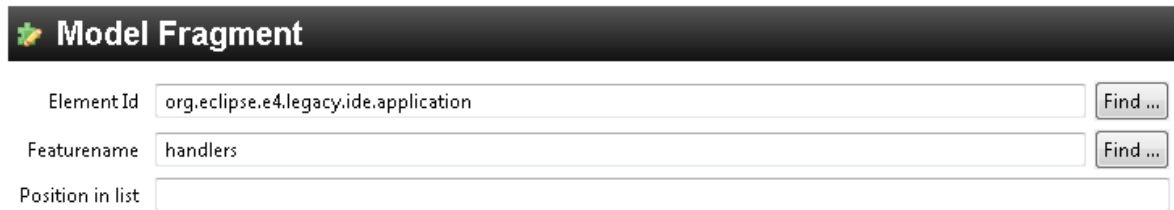


Abbildung 71 E4 MapView Handler Model Fragment Eintrag

Danach können wir den Handler erfassen. Der zeigt auf das vorher erstellte Command und auf die neu erstellte Klasse *OpenMapViewHandler* (siehe unten):

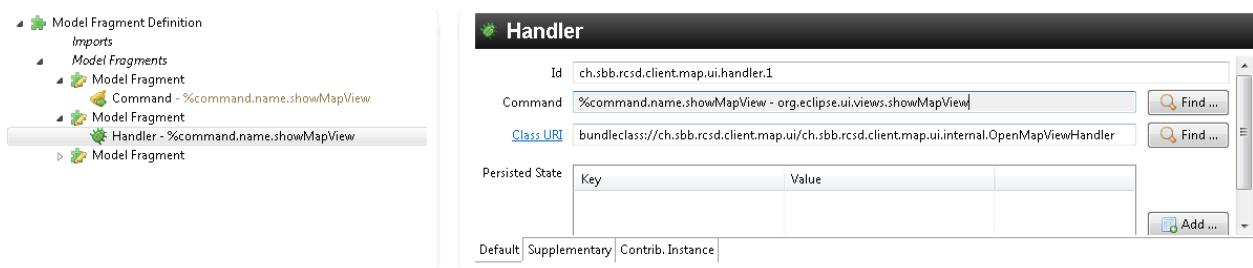


Abbildung 72 E4 MapView Handler Eintrag

TODO Hier Key Binding

Um die *OpenMapViewHandler*-Klasse überhaupt referenzieren zu können, muss diese vorgängig erstellt werden:

```
public class OpenMapViewHandler{

    @Execute
    public void execute(MWindow activeWindow) {
        final MPartStack partStack = findFirstPartStack(activeWindow);
        if (partStack != null) {
            MPart part = MBasicFactory.INSTANCE.createPart();
            part.setContributionURI("bundleclass://ch.sbb.rcsd.client.map.ui/ch.sbb.rcsd.client.map.ui.internal.MapView");
            part.setIconURI("platform:/plugin/ch.sbb.rcsd.client.map.ui/icons/view16/app_karte.png");
            part.setCloseable(true);
            part.setLabel(MapMessages.mapview_title);
            partStack.getChildren().add(part);
        }
    }

    private MPartStack findFirstPartStack(MElementContainer<?> elementContainer) {
        List<?> children = elementContainer.getChildren();
        for (Object o : children) {
            if (o instanceof MPartStack) {
                return (MPartStack) o;
            }
            if (o instanceof MElementContainer<?>) {
                MPartStack partStack = findFirstPartStack((MElementContainer) o);
                if (partStack != null) {
                    return partStack;
                }
            }
        }
        return null;
    }
}
```

Abbildung 73 E4 MapView Handlerklasse

Dieser Handler sucht sich den ersten PartStack innerhalb der Hierarchie des aktuell aktiven Fensters und fügt diesem einen Part hinzu der auf unsere *MapView*-Klasse referenziert (contributionURI). Desweiteren wird dem Part ein Icon und ein Label gesetzt. Dazu soll der Part closeable sein. **TODO spezielle Abhängigkeit notwendig?**

Damit die Applikation weiss, dass wir das Application Model erweitert haben müssen wir nun im plugin.xml noch die Fragment-Datei bekannt geben, dies geschieht über den Extension Point *ch.sbb.rcsd.client.map.ui.fragment*:

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

Abbildung 74 E4 Fragment Eintrag im plugin.xml

6.2.2. Test

Die Migration wurde mit dem Aktivieren des Menüpunktes getestet. Beim Betätigen des Menüpunktes erschien wie gewünscht die MapView zum Vorschein.

6.3. Zusammenfassung

6.3.1. Gegenüberstellung E3 und E4

In der folgenden Tabelle werden diverse Themen zu Commands / Handler, Menus, Key Bindings einander gegenübergestellt. Die Tabelle soll einen Überblick schaffen wo sich E3 und E4 unterscheiden oder wo sie gleich sind.

Thema	E3	E4
Deklaration der Items (Command, Menu, etc.)	Im plugin.xml	Im Application Model
Command	Extension Point org.eclipse.ui.commands im plugin.xml	Im Application Model direkt unter Application
Handler	Extension Point org.eclipse.ui.handlers im plugin.xml	Im Application Model direkt unter Application, innerhalb von Part Descriptors, innerhalb eines Windows oder innerhalb eines Parts
Implementation Handlerklasse	Handlerklasse muss <i>org.eclipse.core.commands.IHandler</i> implementieren, Vielfach wird <i>org.eclipse.core.commands.AbstractHandler</i> erweitert.	POJO, Auszuführende Methode mit <i>@Execute</i> annotiert, optional eine Methode mit <i>@CanExecute</i> annotieren
Referenzierung Handlerklasse	Über Extension Point org.eclipse.ui.handlers im plugin.xml	Über Bundleclass-Notation aus Handler, DirectMenuItem oder DirectToolItem im Application Model
Menu	Extension Point org.eclipse.ui.menus im plugin.xml, mittels locationURI wird bestimmt um was für ein Menu es sich handelt (Menu, Popup, Toolbar) und wo es hin soll (Main-Menu, Main-Toolbar, etc.)	Im Application Model innerhalb eines Windows, als Menu innerhalb eines Parts, oder als PopUp Menu innerhalb eines Parts
Toolbar	Siehe Menu, in der locationURI wird einfach toolbar angegeben	Im Application Model innerhalb von TrimmedWindow's (TrimBars) oder innerhalb von Parts (Option Toolbar muss ausgewählt werden)
Binding	Extension Point org.eclipse.ui.bindings im plugin.xml	Im Application Model innerhalb Application (BindingTables)
Binding Context	Extension Point org.eclipse.ui.contexts im plugin.xml	Im Application Model innerhalb Application (Binding Context)
Action	Nicht mehr brauchen!!	Gibt nichts dergleichen
TODO Mehr?		

6.3.2. Ist Migration machbar?

Die Migration ist nach anfänglichen Schwierigkeiten gelungen. Mir war zuerst nicht klar was genau ich für Items im Fragment erstellen muss und das hat enorm viel Zeit gekostet. Wenn man diese Informationen aber einmal gesammelt hat, so ist Migration relativ leicht zu bewältigen.

Es stellt sich aber für grosse Anwendungen wie RCS die Frage, ob der Aufwand der manuellen Migration vertretbar ist. Sehr wahrscheinlich ist das nicht der Fall. Man müsste hier wohl Tools zum Einsatz bringen die uns sämtliche Menu, Command etc. Einträge im plugin.xml nach Einträgen ins fragment.e4xmi übersetzen. Ein Thema das man hierzu anschauen sollte ist XSL Transformation.



7. Aspekt „Services“

7.1. Beschreibung des Aspektes

In diesem Kapitel geht es darum aufzuzeigen welche Standard-Services in E4 existieren und wie diese eingesetzt werden. eigene Services definiert und implementiert werden können, wie sie registriert und dann auch benutzt werden können.

7.1.1. Diskussion der Eclipse RCP 4 Lösung

Standard-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. **TODO 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	Ein Realm definiert einen Kontext von welchem der Zugriff auf Objekte, die auf IObservable basieren, erfolgen muss. In der Praxis entspricht ein Realm einem Thread. Um den Zugriff über verschiedene Realms zu ermöglichen, können Subklassen von Binding verwendet werden. (javawiki)
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

Eine Liste der zur Verfügung stehenden Services findet man unter http://wiki.eclipse.org/Eclipse4/RCP/EAS/List_of_All_Provided_Services.

TODO Standard Services überschreiben

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:

<pre>public interface GreetingService { void greet(); }</pre>	<pre>public class GreetingServiceConsole implements GreetingService { @Override public void greet() { System.out.println("Hello to the world"); } }</pre>
---	--

Abbildung 75 Beispiel eigener Service

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

Service / Komponente deklarieren und 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.

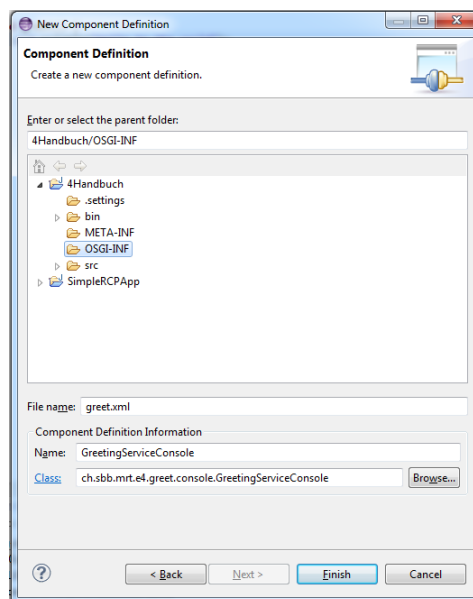


Abbildung 76 New Component Definition

Die Übersicht über die Komponente sieht wie folgt aus:

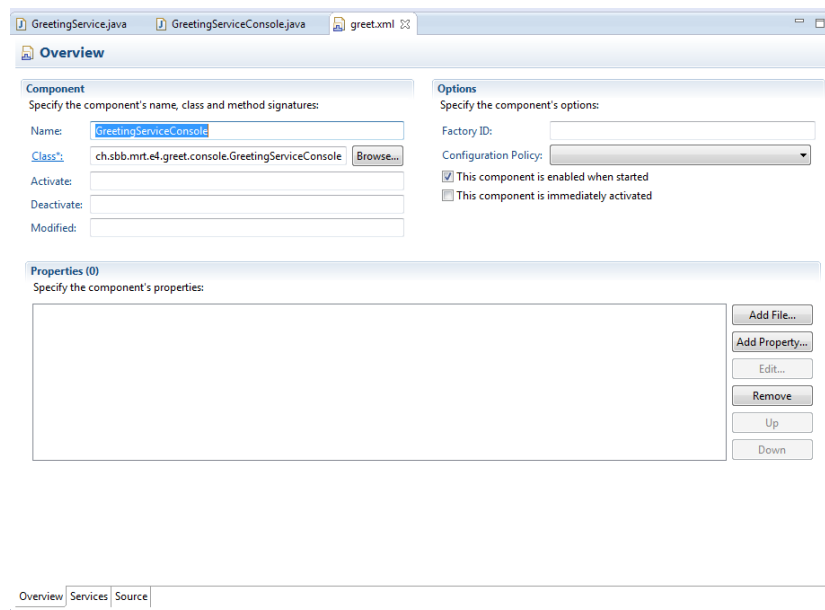


Abbildung 77 Component Definition

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 werden⁴.

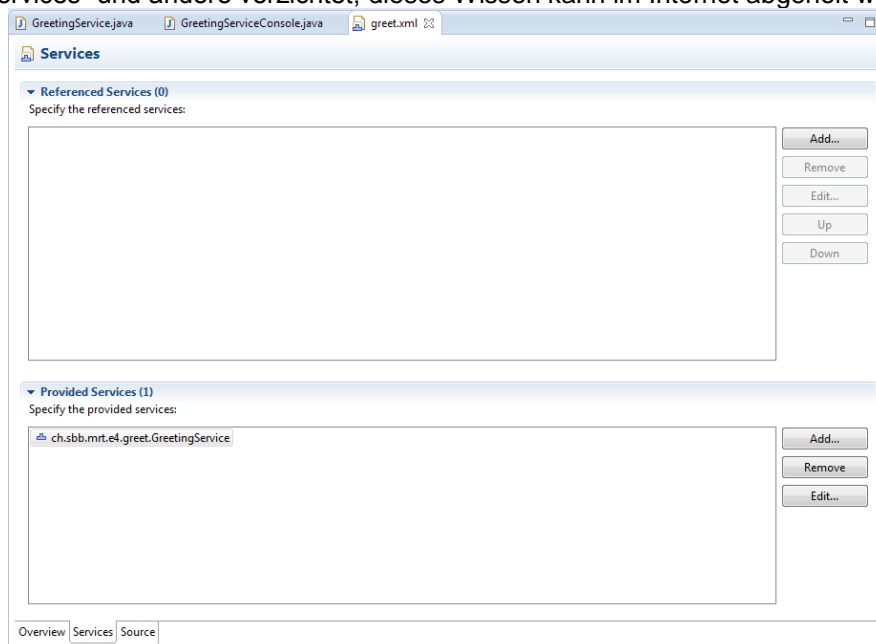


Abbildung 78 Provided Services

⁴ <http://www.vogella.com/articles/OSGiServices/article.html#desconsumer>,
<http://www.eclipsezone.com/eclipse/forums/t97690.rhtml>



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 79 manuelles Registrieren eines Services

Services injizieren

Die Injizierung wird über die bekannte Annotation `@Inject` erreicht. Dies gilt für die Standard-Services wie auch die eigens erstellten OSGi-Services. Im folgenden Beispiel wird der eigene Service von oben injiziert:

```
public interface GreetingService {  
  
    @Inject  
    private GreetingService greeter;  
  
    public void doSomething(){  
        greeter.greet();  
    }  
}
```

Abbildung 80 Service Injection



7.1.2. Vergleich mit Eclipse RCP 3

In E3 werden Standard-Services über das Interface *IServiceLocator* geholt.

```
serviceLocator.getService(Class serviceClass);
```

Das Interface *IWorkbenchSite* erweitert zum Beispiel das *IServiceLocator* 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. Diese Abhängigkeiten machen den Code auch schwerer testbar und auch wiederverwendbar. Des Weiteren ist es unmöglich die Standarddienste zu verändern oder durch eigene Implementationen auszutauschen. Viele nützliche Funktionen der auf die API der Workbench verteilt. **TODO Services auflisten!**

Für eigene Services werden oft eigene Frameworks gebaut. In RCS haben wir zum Beispiel eine Service-Registry Marke Eigenbau im Einsatz. Hier werden die Instanzen über statische Methodenaufrufe über eine Klasse namens *Services* geholt.

7.1.3. Vorteile E4

Die Anzahl der E4 Services wurde bewusst klein gehalten, es ist als Anwender relativ einfach sich die wenigen Services zu merken. Dazu können die Implementationen der Services durch eigene Implementationen ersetzt werden, das kann durchaus nützlich sein. Dies ist ein klarer Vorteil gegenüber E3 wo dies nicht möglich ist.

7.1.4. Einschränkungen und Risiken

TODO Welche Services wurden nicht von E3 nach E4 migriert?

7.1.5. Qualität und Testbarkeit im Vergleich zu Eclipse RCP 3

Die Testbarkeit von Klassen die E4 Services referenzieren ist eindeutig besser als die Testbarkeit der Klassen die E3 Services einsetzen, da das Holen der Instanzen nicht ans Framework gekoppelt ist. Dieser Umstand ist aber eher dem Thema Dependency Injection zuzuordnen. Qualitätsmässig sind beide Ansätze gleich einzustufen, es sind keine grossen Unterschiede auszumachen.

7.2. Services: Konkretes Beispiel RCS

In diesem Kapitel soll ein RCS Service zu einem E4 bzw. OSGi Service migriert werden. Da im *MapView* Beispiel keine Services eingesetzt werden, erweitern wir den *OpenMapViewHandler* mit einer *@Execute* annotierten Methode. In dieser Methode soll geprüft werden, ob der Benutzer die notwendigen Rechte besitzt um die View zu öffnen. Dies geschieht über den *IAuthenticationService*. Ich verzichte hier auf eine Auflistung der Schnittstelle und auch deren Implementation da diese Information für die Migration irrelevant ist. Es ist ausschliesslich wichtig zu wissen, dass die Methode *hasPermission(String)* angeboten wird wo man abfragen kann, ob der angemeldete Benutzer die gewünschte Berechtigung besitzt.

Die *@Execute* Methode der *MapView* wird folgendermassen implementiert:

```
.....
@CanExecute
public boolean canExecute(IAuthenticationService as){
    return as.hasPermission (IPermissions.MAP_VIEW);
}
.....
```

Abbildung 81 CanExecute auf OpenMapViewHandler mit IAuthenticationService

Hinweis: Der *IAuthenticationService* könnte man auch als Field der *MapView* führen und sich dieses Feld injizieren lassen. Ich habe mich hier für die obenstehende Variante entschieden, da der Service sonst nirgends in der *OpenMapViewHandler*-Klasse benötigt wird.

Nachteil dieser Variante: Wenn der Service nicht richtig instanziiert werden konnte (aus welchem Grund auch immer) wird die Methode schlichtweg nicht aufgerufen, es gibt auch keine Fehlermeldung vom Framework. Eine Fehlermeldung bzw. Exception würde es bei der Variante mit dem Field geben.

7.2.1. Migration

Um den sich den Service wie oben injizieren zu lassen muss er OSGi-fiziert werden. Wir machen dies auf die deklarative Art.

Dependencies

Dazu fügen wir im *MANIFEST.MF* des Plugins (*ch.sbb.rcsd.client.auth*) wo der Service platziert ist die folgenden Dependencies hinzu:

```
org.eclipse.equinox.util
org.eclipse.equinox.ds
org.eclipse.osgi.services
```

Aktivierung Plugin

Dazu ist es notwendig dem Plugin beizubringen, dass es aktiviert werden soll, sobald eine der Klassen geladen wird, dies geschieht folgendermassen:

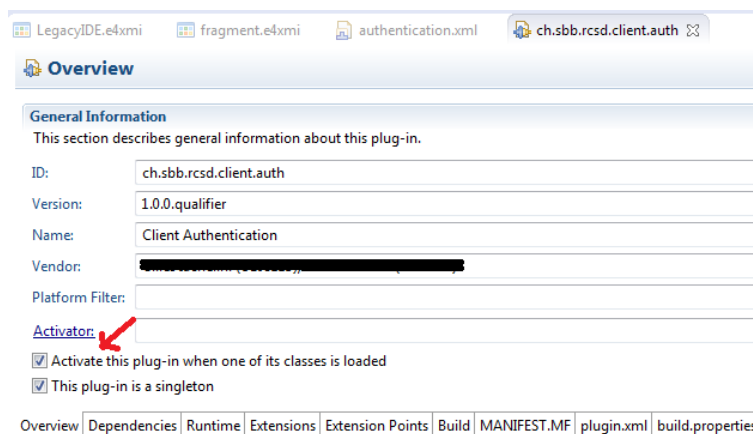


Abbildung 82 Authentisierungs-Service Aktivierung Plugin wenn Klasse geladen

Start-Level

Bei der Produkt-Definition ist es wichtig den Plugins *org.eclipse.core.runtime* und *org.eclipse.equinox.ds* einen Start-Level von unter 4 zu geben:

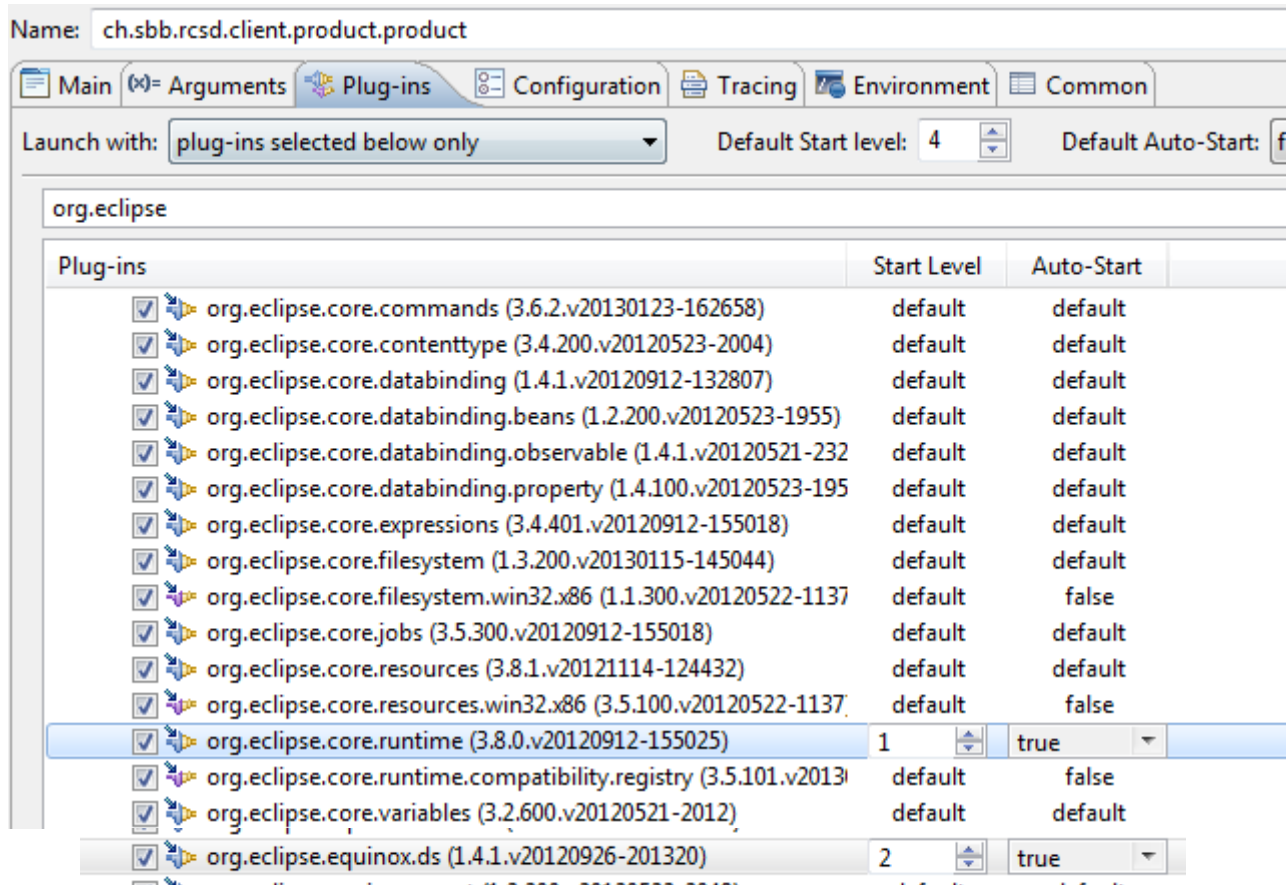


Abbildung 83 Authentisierungs-Service Start-Level von Plugins

Das Plugin *org.eclipse.core.runtime* definiert die OSGi Runtime und das Plugin *org.eclipse.equinox.ds* ist verantwortlich für das Lesen der Komponenten-Metadaten und für die Registrierung der auf Definitionsdateien von Komponenten basierenden Services. (<http://www.vogella.com/>)

Es ist also notwendig, dass die zwei Plugins gestartet werden bevor der Service verfügbar wird.

Service deklarieren

Wie bereits im theoretischen Teil dieses Aspektes wählen wir nun den Menüpunkt New... Component Definition. Im Dialog wählen wir den Datei-Namen (hier *authentication.xml*), den Ort der Ablage (Im Projekt bzw. Plugin im Ordner OSGI-INF), den Namen der Komponente (*authenticationService*) und wir wählen die implementierende Klasse (*ch.sbb.rcsd.client.auth.internal.AuthenticationService*) aus.

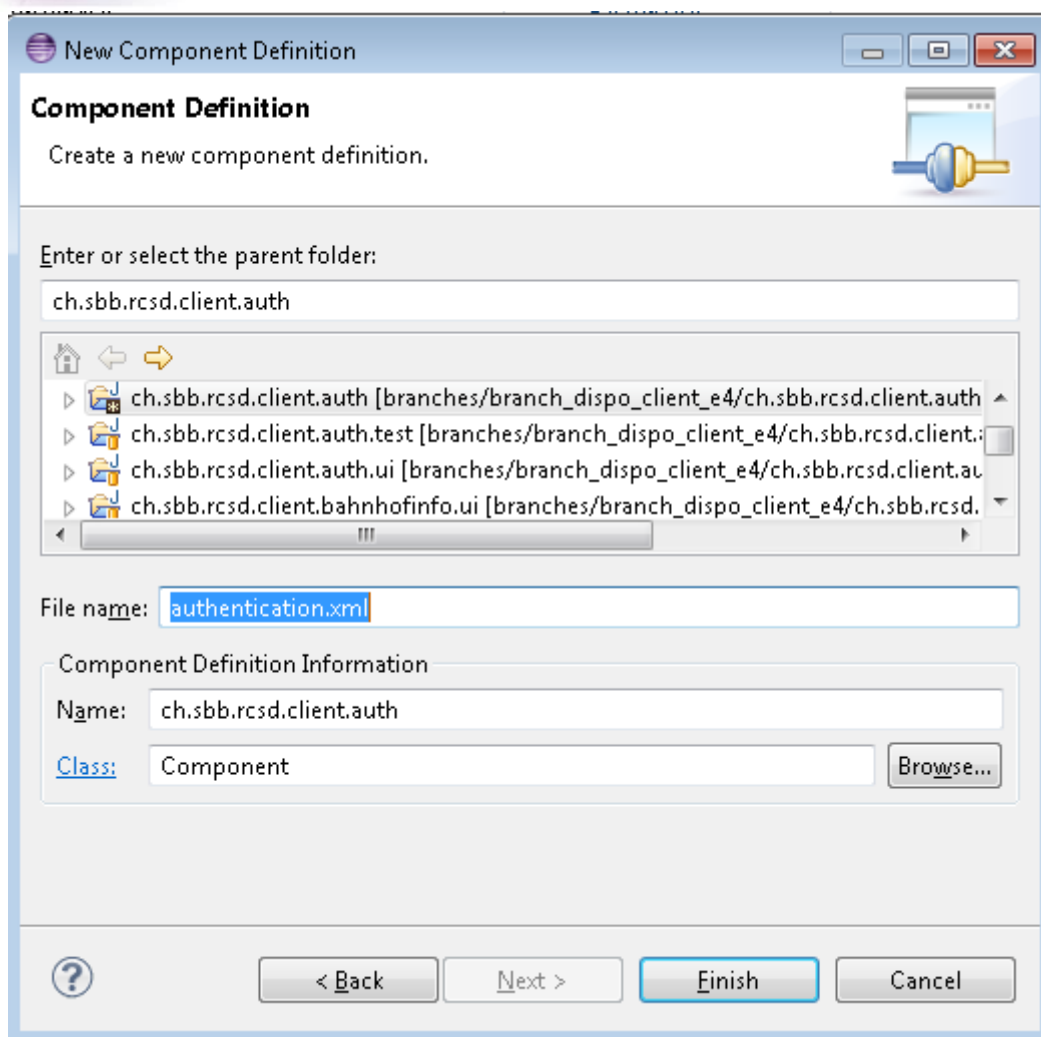


Abbildung 84 Authentisierungs-Service New Component

In der Übersicht sieht das dann so aus:

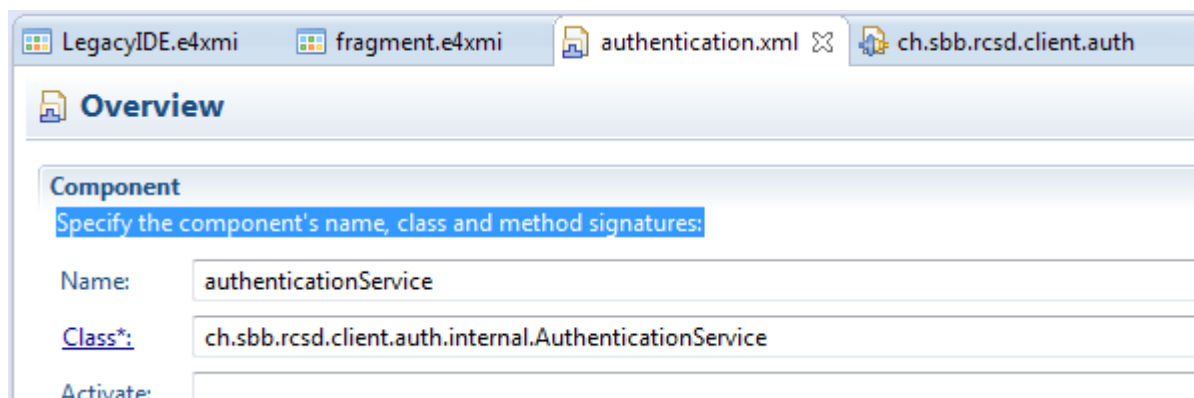


Abbildung 85 Authentisierungs-Service Component Overview

Im Tab „Services“ wählen wir das Interface `ch.sbb.rcsd.client.auth.IAuthenticationService` als Provided Service aus:

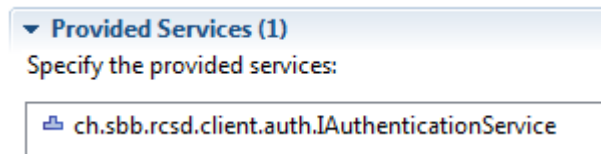


Abbildung 86 Authentisierungs-Service Provided Services

Die Sourcen des Files `authentication.xml` sehen danach folgendermassen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    immediate="true" name="authenticationService">
  <implementation class="ch.sbb.rcsd.client.auth.internal.AuthenticationService"/>
  <service>
    <provide interface="ch.sbb.rcsd.client.auth.IAuthenticationService"/>
  </service>
</scr:component>
```

Abbildung 87 Source AuthenticationService Component

Das wars, der Service ist OSGi-fiziert. Die Injizierung in den

7.3. Zusammenfassung

7.3.1. Gegenüberstellung E3 und E4

In der folgenden Tabelle werden diverse Themen zu „Services“ einander gegenübergestellt. Die Tabelle soll einen Überblick schaffen wo sich E3 und E4 unterscheiden oder wo sie gleich sind.

Thema	E3	E4
Service-Art	Etwas selbst gestricktes, in RCS zum Beispiel eigene Service-Registry	OSGi-Services
Holen der Service Instanzen	Über IServiceLocator welcher von der Workbench oder WorkbenchWindow etc. implementiert wird. In RCS über statischen Methodenaufruf auf Services	Über Dependency Injection, OSGi Service-Instanzen werden im OSGi-Kontext abgelegt.
Eigene Implementationen für Standard-Services	Nein, nicht möglich	Ja, möglich
Anzahl Standard-Services	Viele	Weniger, Stichwort: The 20 things

7.3.2. Ist Migration machbar?

Die Migration der Standard-Services ist gegeben. Ein Anwender in E4 nützt die neuen Services. Die Schwierigkeit der Migration von eigenen Services kann - je nachdem wie diese implementiert sind – ziemlich unterschiedlich sein. Bei RCS ist die Migration relativ leicht machbar, da immer mit Interfaces gearbeitet wurde.

Auch hier stellt sich wieder die Frage ob die RCS-Services nach E4 mit den geeigneten Tools automatisch migrieren lassen. Dies müsste man zumindest prüfen, dies würde aber den Zeitrahmen dieser Arbeit sprengen.



8. Aspekt „Eigene Extension Points“

8.1. Beschreibung des Aspektes

Dieser Aspekt konnte aus Mangel an Zeit nicht mehr bearbeitet werden.

In diesem Kapitel geht es darum aufzuzeigen wie eigene Extension Points aus E3 in E4 abgelöst werden können. Da dieses Thema oder dieser Aspekt weit aus konkreter ist als die bisherigen weicht die Kapitelstruktur auch ziemlich vom bereits gewohnten ab. Es sollen ein paar konkrete Extension Points aus RCS genauer durchleuchtet werden und für jeden einen Weg gefunden werden wie dieser migriert werden kann.

ExtensionPoints (eigene):

- ch.sbb.rcsd.client.ui.windowicons
- ch.sbb.rcsd.client.ui.rtvviewer.bildtypes
- ch.sbb.rcsd.client.sound.ui.events
- ch.sbb.rcsd.client.ui.theme.cursors
- ch.sbb.rcsd.client.ui.theme.images
- ch.sbb.rcsd.client.services.providers
- ch.sbb.rcsd.client.application.statusbar
- ch.sbb.rcsd.client.sound.ui.sounds
- Events

Alle EXSDs

- bildtypes
- bookmarks
- caches
- ch.sbb.rcsd.client.application.statusbar
- ch.sbb.rcsd.client.zwl.extension
- channels (2x)
- cursors
- decorators
- definitions
- dispopbview
- events
- images
- inspection
- loginDialogContribution
- navigator
- providers
- receivers
- renderer
- renderingflags
- report
- sounds
- status
- windowicons



9. Reflexion



Berner Fachhochschule

Technik und Informatik

Software-Schule Schweiz



10. Verzeichnisse / Quellen

10.1. Literaturverzeichnis

- Hoffmann, M. (kein Datum). <https://github.com/marchof/article-eclipsecommands>. Von <https://github.com/marchof/article-eclipsecommands>: <https://github.com/marchof/article-eclipsecommands> abgerufen
- http://de.wikipedia.org/wiki/Dependency_Injection. (kein Datum). *Wikipedia.org*. Von *Wikipedia.org*: http://de.wikipedia.org/wiki/Dependency_Injection abgerufen
- <http://www.vogella.com/>. (kein Datum). *www.vogella.com*. Von http://www.vogella.com/articles/OSGiServices/article.html#declarativeservices_startlevel abgerufen
- javawiki. (kein Datum). <http://javawiki.sowas.com/>. Von <http://javawiki.sowas.com/>: <http://javawiki.sowas.com/doku.php?id=eclipse-rcp:realm> abgerufen
- Teufel, M. (kein Datum). Eclipse 4. In M. Teufel, *Eclipse 4* (S. Kapitel 5.6).
- Vogel, L. (kein Datum). <http://www.heise.de/developer/artikel/Services-und-Dependency-Injection-in-der-Eclipse-4-0-Application-Platform-1048518.html?artikelseite=2>. Von <http://www.heise.de/developer/artikel/Services-und-Dependency-Injection-in-der-Eclipse-4-0-Application-Platform-1048518.html?artikelseite=2>: <http://www.heise.de/developer/artikel/Services-und-Dependency-Injection-in-der-Eclipse-4-0-Application-Platform-1048518.html?artikelseite=2> abgerufen



10.2. Weitere Quellen

Allgemein

<http://blog.doubleslash.de/mehr-flexibilitaet-durch-eclipse-4/>
<http://eclipse-source.com/blogs/2012/06/18/migrating-from-eclipse-3-x-to-eclipse-4-e4/>
<http://martinzimmermann1979.wordpress.com/2011/07/03/migration-eines-eclipse-rcp-projektes-auf-4-x/>
<http://www.vogella.com/articles/Eclipse4MigrationGuide/article.html>
<http://tomsondev.bestsolution.at/2012/04/13/why-is-eclipse-4-better-than-eclipse-3/>
<http://www.heise.de/developer/artikel/Eclipse-4-die-naechste-Generation-der-freien-IDE-962171.html>
http://www.eclipse.org/eclipse/development/readme_eclipse_4.2.html#KnownIssues
<http://www.heise.de/developer/artikel/Services-und-Dependency-Injection-in-der-Eclipse-4-0-Application-Platform-1048518.html?artikelseite=2>

Adapter

<http://www.vogella.com/articles/Eclipse4Services/article.html>
http://wiki.eclipse.org/E4/EAS/Adapting_Objects
<http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html>
<http://javawiki.sowas.com/doku.php?id=eclipse:adapter>
http://wiki.eclipse.org/FAQ_How_do_I_use_IAdaptable_and_IAdapterFactory%3F
<http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html>
<http://books.google.ch/books?id=ZAWxJ9dfoE8C&pg=PA295&lpg=PA295&dq=Adapter+Eclipse&source=bl&ots=3A-O-BQnKf&sig=CcBQ5vQ5aBvNYnwFC8xfwY7XUyk&hl=de&sa=X&ei=0-PJUezZPMO7PbiagOgH&ved=0CFAQ6AEwBA#v=onepage&q=Adapter%20Eclipse&f=false>

DI

http://de.wikipedia.org/wiki/Dependency_Injection
http://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection
<http://wiki.eclipse.org/E4/Contexts>
<http://de.slideshare.net/LarsVogel/eclipse-e4-tutorial-eclipsecon-2010-3587897>
<http://www.heise.de/developer/artikel/Services-und-Dependency-Injection-in-der-Eclipse-4-0-Application-Platform-1048518.html>

Services

<http://www.vogella.com/articles/OSGiServices/article.html>
http://wiki.eclipse.org/Eclipse4/RCP/EAS/List_of_All_Provided_Services

Command Handler etc.

<http://hermanlintvelt.blogspot.ch/2009/06/eclipse-rcp-commands-part-3-visiblewhen.html>
<http://www.vogella.com/articles/EclipseRCP/article.html#commands>
http://wiki.eclipse.org/FAQ_What_is_the_difference_between_a_command_and_an_action%3F
<http://www.vogella.com/articles/EclipseCommands/>
http://wiki.eclipse.org/Platform_Command_Framework
http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_cmd_menus.htm
http://wiki.eclipse.org/Menu_Contributions
<http://www.vogella.com/articles/EclipseCommandsKeybindings/article.html>
http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg.eclipse.ui_bindings.html
<https://github.com/marchof/article-eclipsecommands>

Download Links

<http://download.eclipse.org/e4/downloads/>



10.3. Abbildungsverzeichnis

Abbildung 1 Architektur E4.....	8
Abbildung 2 Architektur E3.....	10
Abbildung 3 Compability Layer E3.....	11
Abbildung 4 Klasse ZugnummerRendererView E3 Stil	13
Abbildung 5 Klasse ZugnummerRendererView E4 Stil	14
Abbildung 6 plugin.xml entferne aus org.eclipse.ui.views	14
Abbildung 7 plugin.xml mit dem applicationXMI Eintrag.....	15
Abbildung 8 fragment.e4xmi add TrimmedWindow	16
Abbildung 9 fragment.e4xmi add Part.....	16
Abbildung 10 fragment.e4xmi define contributionURI	17
Abbildung 11 plugin.xml mit dem fragment.e4xmi Eintrag.....	17
Abbildung 12 Klasse ZugnummerRendererView E4 Stil	18
Abbildung 13 plugin.xml mit dem processor Eintrag.....	18
Abbildung 14 Klasse ZugnummerRendererView Vererbung von DUIViewPart	19
Abbildung 15 Klasse ZugnummerRendererViewWrapped	19
Abbildung 16 Geografische Karte RCS	21
Abbildung 17 Geografische Karte RCS mit zwei selektierten Zügen.....	21
Abbildung 18 Geografische Karte RCS mit selektiertem Bahnhof.....	22
Abbildung 19 Klasse MapView E3 (1 von 2)	23
Abbildung 20 Klasse MapView E3 (2 von 2)	24
Abbildung 21 Extension Point MenuContribution MapView E3	26
Abbildung 22 Extension Point View MapView E3	26
Abbildung 23 Klasse Beispiel Dependency Injection	27
Abbildung 24 lookup im Context	28
Abbildung 25 Kontextzugriff per API	30
Abbildung 26 Beispiel Manuelle Erweiterung IEclipseContext	30
Abbildung 27 Beispiel Manuelle Injection	30
Abbildung 28 Beispiel Manuelle Injection	31
Abbildung 29 MapView als DUIViewPart.....	32
Abbildung 30 Klasse MapViewWrapped E4	33
Abbildung 31 Klasse Adapter.....	35
Abbildung 32 Adapterbeispiel	35
Abbildung 33 Klasse mit Adapter Injection	36
Abbildung 34 Klasse mit AdapterManager über Platform	36
Abbildung 35 Interface mit IAdapterFactory.....	36
Abbildung 36 Implementation von IAdapterFactory	37
Abbildung 37 Deklarative Registration einer AdapterFactory	37
Abbildung 38 Programmatische Registration einer AdapterFactory	37
Abbildung 39 AdapterManager ClassCastException	38
Abbildung 40 MapView mit E4 Adapter	39
Abbildung 41 Trennung Präsentation und Verhalten mit Commands.....	41
Abbildung 42 Handler, Command, Key Binding und Menu/Toolbar	42
Abbildung 43 E4 TeamInfoHandler Beispiel	43
Abbildung 44 Handler im Application Model	44
Abbildung 45 MHandlerContainer Hierarchie	44
Abbildung 46 Add Menu to Window	46
Abbildung 47 Add Menu to Part	46
Abbildung 48 Toolbar auf Window	47
Abbildung 49 Toolbar auf Part	47
Abbildung 50 Beispiel ToolControl Klasse	48
Abbildung 51 Beispiel ToolControl im Application Model	48
Abbildung 52 Key Binding im Application Model.....	49
Abbildung 53 Binding Table im Application Model.....	49
Abbildung 54 E4 Binding Context	50
Abbildung 55 Deklaration Command	50
Abbildung 56 E3 TeamInfoHandler Klasse	51



Abbildung 57 E3 Handler und Command verbinden.....	51
Abbildung 58 E3 Handler programmatisch aktivieren.....	51
Abbildung 59 Deklaration Menu Contribution	52
Abbildung 60 E3 Key Binding und Command verbinden	53
Abbildung 61 E3 Contexts.....	53
Abbildung 62 E3 Context Activation.....	53
Abbildung 63 E3 Programmatische Context Erstellung	53
Abbildung 64 Entfernen E3 MapView Menu Eintrag.....	55
Abbildung 65 Entfernen E3 MapView Eintrag.....	55
Abbildung 66 E4 MapView Command Model Fragment Eintrag	55
Abbildung 67 E4 MapView Command Eintrag.....	56
Abbildung 68 E4 MapView Menu Contributions Model Fragment Eintrag.....	56
Abbildung 69 E4 MapView Menu Contributions Eintrag	56
Abbildung 70 E4 MapView Menu Item Eintrag	56
Abbildung 71 E4 MapView Handler Model Fragment Eintrag	57
Abbildung 72 E4 MapView Handler Eintrag.....	57
Abbildung 73 E4 MapView Handlerklasse	58
Abbildung 74 E4 Fragment Eintrag im plugin.xml.....	58
Abbildung 75 Beispiel eigener Service	62
Abbildung 76 New Component Definition	62
Abbildung 77 Component Definition	63
Abbildung 78 Provided Services	63
Abbildung 79 manuelles Registrieren eines Services.....	64
Abbildung 80 Service Injection.....	64
Abbildung 81 CanExecute auf OpenMapViewHandler mit IAuthenticationService	66
Abbildung 82 Authentisierungs-Service Aktivierung Plugin wenn Klasse geladen.....	66
Abbildung 83 Authentisierungs-Service Start-Level von Plugins.....	67
Abbildung 84 Authentisierungs-Service New Component	68
Abbildung 85 Authentisierungs-Service Component Overview	68
Abbildung 86 Authentisierungs-Service Provided Services.....	69
Abbildung 87 Source AuthenticationService Component	69
Abbildung 88 Debug Parameter AuthenticationService.....	Fehler! Textmarke nicht definiert.
Abbildung 89 Test AuthenticationService Menüpunkt enabled und disabled.....	Fehler! Textmarke nicht definiert.