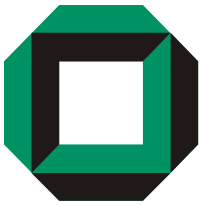
Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

Software-Entwicklung mit Eclipse
–
Proseminar im Wintersemester 2006/2007
an der Universität Karlsruhe (TH),
Institut für Programmstrukturen und Datenorganisation,
Lehrstuhl für Software-Entwurf und Qualität

Herausgeber:
Steffen Becker, Tobias Dencker, Jens Happe, Heiko Koziolek, Klaus Krogmann, Martin Krogmann, Michael Kuperberg, Ralf Reussner, Martin Sygo, Nikola Veber
Interner Bericht 2007-11



Universität Karlsruhe

Fakultät für Informatik

ISSN 1432 - 7864

Inhaltsverzeichnis

Proseminar: Software-Entwicklung mit Eclipse
Plug-In Development

Martin Sygo

Universität Karlsruhe

Zusammenfassung Diese Arbeit behandelt behandelt das Plug-In-System von Eclipse. Dieses ist tief im Kern der Eclipse-Architektur verankert. Außerdem verfügt es über eine Reihe von Mechanismen, welche die Plug-Ins untereinander koordinieren und für ein reibungsloses Zusammenwirken verschiedener Plug-Ins sorgen.

1 Die Eclipse Plug-In Architektur

Unter einem Plug-In versteht man in der Softwaretechnik ein Programm, das sich, wie der Name es vermuten lässt, in ein anderes Programm "einklinkt" und dieses erweitert, d.h. dem erweiterten Programm neuen Code und somit neue Funktionalität hinzufügt. Ein Plug-In basierter Ansatz zur Entwicklung von Software bietet Vorteile gegenüber der Entwicklung einer monolithischen Anwendung. Durch Plug-Ins kann neue Funktionalität bei Bedarf nachgeliefert werden, ohne das ganze Produkt neu ausliefern zu müssen. Außerdem wird die Anwendung flexibler, da Kunden selbst eigene Plug-Ins schreiben und somit die Anwendung an eigene Bedürfnisse anpassen können.

Im Folgenden werden wird das Plug-In-System von Eclipse beschrieben, wobei im ersten Teil der Arbeit nach einem Blick auf die Stellung von Plug-Ins in der Gesamtarchitektur von Eclipse die wesentlichen Konzepte abstrakt dargestelltwerden. Im zweiten Teil der Arbeit wird an Hand eines Beispiels darauf eingegangen, wie diese Konzepte eingesetzt werden müssen, um neue Plug-Ins zu entwickeln.

1.1 Plug-Ins in der Gesamtarchitektur von Eclipse

Bevor wir mit der Beschreibung des Plug-In-Systems von Eclipse beginnen, ist es zwingend erforderlich, darüber Klarheit zu schaffen, was Eclipse ist. Viele denken bei dem Wort Eclipse zunächst an eine IDE für die Programmiersprache Java. In der Tat hat das Eclipse-Projekt eine Java-IDE hervorgebracht, die heute vielerorts Verwendung findet. Jedoch ist Eclipse nicht nur eine IDE. Denn Ziel des Eclipse-Projektes ist nicht nur die Erstellung einer einzelnen IDE für Java, sondern auch, eine *Plattform* zu schaffen, die als Grundlage für eine Vielzahl von Werkzeugen zur Software-Entwicklung dienen kann [10]. Ein solches Entwicklungswerkzeug präsentiert sich dem Endbenutzer als Anwendungsprogramm, benutzt aber, um seine Funktionalität zur Verfügung zu stellen, die Dienste der

Plattform. Die bereits genannte Java-IDE ist das bekannteste Beispiel für ein solches Anwendungprogramm, das auf der Eclipse-Plattform aufbaut.

Einer der Dienste, welche die Plattform in eine neue Anwendung einbringen kann, ist das Plug-In-System, welches Thema dieser Arbeit ist. Daher werden wir im Folgenden nicht genauer auf die Java-IDE des Eclipse-Projektes eingehen, sondern uns vielmehr mit der Plattform beschäftigen. Abschnitt 1.1 soll zunächst einen Eindruck davon vermitteln, was die Eclipse-Plattform ist und wie die Beziehungen zwischen Plattform, Anwendungen und Plug-Ins sind

Die Plattform. Auf der Eclipse-Plattform aufbauend können komplexe Anwendungsprogramme erstellt werden, was jedoch nicht bedeutet, dass die Plattform nur eine Sammlung von Software-Bibliotheken ist. Im Gegenteil, sie ist selbst aufgebaut wie ein Anwendungsprogramm. Was sie von gewöhnlichen Anwendungsprogrammen unterscheidet ist, dass die Plattform – wie das Eclipse-Projekt selbst angibt – nicht für einen speziellen Anwendungsfall entworfen wurde: *"Die Eclipse Plattform bietet für sich genommen nicht viel Endbenutzer-Funktionalität – interessant ist, was sie ermöglicht."* [10]. So erlaubt die Plattform selbst das Öffnen und Bearbeiten von Dateien. Zum Bearbeiten bietet sie dem Endbenutzer jedoch lediglich einen einfachen Texteditor an, der in keiner Weise auf das Bearbeiten von Quelltexten einer bestimmten Programmiersprache zugeschnitten ist.

Interessant wird die Plattform dadurch, dass sie als *Integrationspunkt* [9] für Erweiterungen mit anwendungsspezifischer Funktionalität dienen kann. So kann die Eclipse-Plattform beispielsweise durch das *JDT*-Paket (*Java Development Tools*) erweitert werden – das Ergebnis ist die Java-IDE des Eclipse-Projektes. Das JDT ist eine Sammlung von Plug-Ins, die sich in die Plattform einhängen und ihr neue Features wie z.B. einen Editor mit Syntax-Highlighting für Java, Refactoring-Tools oder einen Debugger hinzufügen.

Den Aufbau der Plattform stellt Abb. 1 schematisch dar.

Die Plattform selbst ist nicht monolithisch aufgebaut, sondern besteht ihrerseits bereits aus Plug-Ins. Dies veranschaulicht Abb. 1. Die Plug-Ins der Plattform sind:

- Das **Workbench**-Plug-In enthält die grafische Benutzeroberfläche. Es verwendet die Plug-Ins **SWT** und **JFace**, welche Bibliotheken für das Fenstersystem bereitstellen.
- Das **Help**-Plug-In enthält das Eclipse-Hilfesystem.
- Das **Team**-Plug-In stellt Funktionen zur Teamarbeit bei der Entwicklung bereit. Dazu gehört unter anderem ein integrierter CVS-Client.
- Das **Workspace**-Plug-In beinhaltet die Ressourcen-Verwaltung, d.h. es stellt die Funktionen zum Öffnen und Schließen, Überwachen von Dateiänderungen usw. bereit.
- Das **Runtime**-Plug-In enthält den Laufzeitkern der Eclipse-Plattform.

Für uns ist die Eclipse-Runtime der wichtigste Teil der Plattform, denn ihr obliegt das Laden und Entladen von Plug-Ins. Die Runtime unterscheidet sich

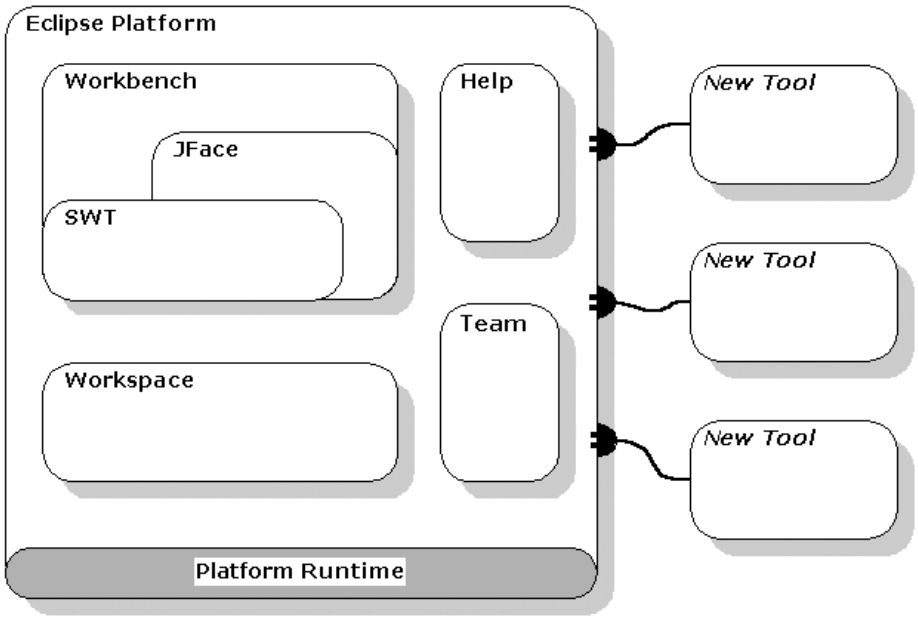


Abbildung 1. Die Eclipse-Plattform

von allen anderen Teilen der Plattform dadurch, dass sie von einem Bootstrap-Code direkt geladen wird. Alle anderen Plug-Ins werden von der Runtime geladen. (Mehr zur Aktivierung und zum Laden von Plug-Ins siehe Abschnitt 1.4). Obwohl die Runtime ausgeführt wird, bevor die Programmlogik zum Laden von Plug-Ins (die ja in der Runtime enthalten ist) verfügbar ist, so ist es doch korrekt, sie als Plug-In zu bezeichnen, denn sie hat die Struktur, die für alle Eclipse-Plug-Ins verpflichtend ist (siehe dazu Abschnitt 1.2).

Plattform-Dienste. Wir wollen noch kurz auf die Dienste eingehen, welche von der Eclipse-Plattform für Anwendungen zur Verfügung gestellt werden. Es sind dies vor allem solche Funktionen, die in vielen verschiedenen Anwendungen zum Einsatz kommen. Ein Beispiel hierfür ist zum Beispiel der Zugriff auf das Dateisystem: Die meisten Anwendungen geben dem Benutzer die Möglichkeit, Daten in Dateien zu speichern und aus Dateien zu laden. Für das Lesen und Schreiben von Dateien kann eine Anwendung, welche auf der Eclipse-Plattform aufbaut auf die Dienste der Plattform zurückgreifen. Auf einen Dienst der Plattform zurückzugreifen bietet Vorteile bei der Anwendungsentwicklung, da auf bestehende Plug-Ins und damit auf eine bereits getestete Implementierung zurückgegriffen werden kann, was die Fehleranfälligkeit verringert und die zur Neuentwicklung der Funktion nötige Zeit einspart.

Die folgende Aufzählung nennt die wichtigsten Dienste, welche die Plattform zur Verfügung stellt [5]:

- Laden von Plug-Ins
- Verwaltung der Einstellungen (*Preferences*) der Plug-Ins
- Verwaltung von Ressourcen (Dateisystem-Interaktion, Netzwerk-Zugriff etc.)
- Verwaltung von *Jobs*, d.h. Multi-Tasking-Funktionalität für Eclipse-Anwendungen

Eclipse-basierte Anwendungen. Unter einer Eclipse-basierten Anwendung wollen wir im Folgenden ein Anwendungsprogramm verstehen, das wie die Eclipse-Java-IDE auf der Plattform aufbaut. "Auf der Plattform aufbauen" bedeutet dabei, der Plattform Plug-Ins hinzuzufügen und sie auf diesem Wege zu einem Werkzeug zu erweitern, das auf ein bestimmtes Anwendungsgebiet zugeschnitten ist.

Die Plattform unterscheidet verschiedene "Typen" von Erweiterungen, die hinzugefügt werden können. Für diese lässt sich folgene Hierarchie angeben:

- *Product*
- *Features*
- *Plug-Ins*

Den Aufbau dieser Hierarchie stellt man am Besten von unten her dar, denn Plug-Ins bilden die Grundlage für aller Erweiterung, und die anderen Typen entstehen nur durch Zusammenfassen von Plug-Ins. So werden mehrere Plug-Ins gebündelt zu einem sogenannten *Feature*. Ein Feature ist die kleinste Einheit, in der Erweiterungen für Eclipse normalerweise erhältlich sind: Plug-Ins werden zu Features verpackt (wobei ein Feature durchaus aus einem einzelnen Plug-In bestehen kann) und zum Download bereitgestellt. Dabei muss ein *Feature* gezwungenermaßen eine Lizenz enthalten, um vom Update Manager von Eclipse akzeptiert zu werden [11].

Auf oberster Ebene steht das *Product*. Ein *Product* ist eine Erweiterung, die bestimmt, als welche Anwendung sich die erweiterte Plattform dem Benutzer präsentiert. Es bundelt alle Funktionalität, die Anwendung ausmacht. Aus Sicht der Plattform ist es ein "Komplett-Paket", das alles enthält, was für die Anwendung benötigt wird, oder, wie es die Plattform-SDK-Dokumentation sagt: *"Products enthalten allen Code und alle Plug-Ins, die zu ihrer Ausführung benötigt werden. Dazu gehört eine Java-Laufzeitumgebung (JRE) und der Code der Eclipse-Plattform."* [7].

1.2 Aufbau von Plug-Ins

Plug-Ins werden unter Umständen erst entwickelt, nachdem ein Softwareprodukt längst fertiggestellt und schon vielerorts im Einsatz ist. Damit aber ein Plug-In nachträglich installiert und geladen werden kann, muss ihm zwangsläufig eine gewisse Struktur auferlegt werden. Andernfalls könnte der Code in der Plattform,

der das Plug-In zur Ausführung bringen soll, nur schwerlich wissen, wie er das Plug-In laden und vor allem an welche Routine aus dem Code des Plug-Ins er die Kontrolle übergeben sollte. Dieser Abschnitt beschreibt, wie diese Mechanismen in der Eclipse-Plattform verwirklicht sind.

Verzeichnis-Layout bei Plug-Ins. Um ganz oberflächlich zu beginnen, bemerken wir zunächst, dass alle Plug-Ins in einer Eclipse-Installation in einen Unterordner `plugins` installiert werden. Dort werden sie entweder in einen eigenen Unterordner entpackt oder als `jar`-Archiv gespeichert. Diese Ordner bzw. Archive enthalten als `class`-Dateien den Code, der die Plattform erweitern soll, sowie alle übrigen Ressourcen, die ein Plug-In benötigt, beispielsweise HTML-Dokumente für ihre Hilfeseiten oder Bilddateien für Icons oder Splash-Screens (p. 37)[11]. Weiterhin enthält jedes Plug-In in seinem Wurzelverzeichnis eine Datei mit dem Namen `plugin.xml`. Dies ist eine *Manifest*-Datei, die Informationen darüber enthält, wie die Plattform das Plug-In zur Ausführung bringen kann [1]. Man kann also den Aufbau eines Plug-Ins unterteilen in Code, Ressourcen und Metainformationen.

Es müssen nicht immer alle drei genannten Elemente vorhanden sein, mit Ausnahme der Manifest-Datei `plugin.xml`, ohne die das Plug-In von der Plattform ignoriert werden würde. Welche davon im konkreten Fall vorhanden sind, hängt vom geplanten Verwendungszweck für das Plug-In ab. Dieser kann unter anderem einer der folgenden sein:

- Die meisten Plug-Ins dienen dazu, neue Programmlogik zur Plattform hinzuzufügen. Solche Plug-ins können entweder nur aus Class-Dateien mit Java-Bytecode bestehen (wenn sie etwa abstrakte Programmlogik für eine Anwendung implementieren), oder sie können neben den Class-Dateien noch andere Ressourcen enthalten. Z.B. kann ein Plug-In, das einen neuen Menübefehl in das User-Interface einfügen will, ein Icon für diesen Menübefehl mitliefern.
- Manche Plug-Ins enthalten überhaupt keinen Code, sondern nur Ressourcen-Dateien. Dazu gehören z.B. Plug-Ins, die nur Dokumentation zum Hilfesystem hinzufügen und dementsprechend nur HTML-Dokumente enthalten, die das Hilfesystem anzeigen kann. (Das Manifest enthält in diesem Fall Informationen darüber, wo die neuen Hilfeseiten im Inhaltsverzeichnis der Hilfe angezeigt werden sollen.)
- Werden für eine Anwendung externe Programm-Bibliotheken benötigt, so werden diese oftmals in ein *Bibliotheks-Plug-In* verpackt (pp.109f)[11]. Solche Plug-Ins enthalten zwar Code, unterscheiden sich aber von den in dieser Aufzählung zuerst aufgeführten Plug-Ins dadurch, dass sie lediglich eine Bibliothek importieren und nicht direkt die Plattform erweitern, da durch das Laden eines solchen Bibliotheks-Plug-Ins nocht keine neuen Funktionen in die Plattform eingefügt werden (d.h. der Benutzer hat damit noch keine Möglichkeit, die Ausführung von Methoden aus dieser Bibliothek zu veranlassen).

Metainformationen. Neben dieser äußeren Struktur, die rein auf dem Layout des Verzeichnisses, in dem das Plug-In installiert ist, beruht, besitzt ein Eclipse-Plug-In aber auch noch eine innere Struktur. Letztere ist wesentlich bedeutsamer als der Aufbau des Plug-In-Verzeichnisses, da sie bestimmt, wie das Plug-In in die Plattform eingefügt wird und mit dieser zusammenarbeitet. Die Rede ist hierbei von der bereits erwähnten Manifest-Datei des Plug-Ins, der `plugin.xml` und den in ihr enthaltenen Metainformationen.

Diese Metainformationen sind nicht nur deshalb vonnöten, damit ein Plug-In überhaupt in die Plattform geladen werden kann. Denn in Eclipse kann man mit Plug-Ins nicht nur eine Anwendung erweitern: Die Eclipse-Architektur sieht vor, dass Plug-Ins, die eine Anwendung erweitern, selbst wieder erweitert werden können. Das macht die Vorgänge natürlich komplizierter, denn Plug-Ins können sowohl die Rolle des *Erweiterers* annehmen [1], als auch selbst Erweiterungen aufnehmen. Die Information darüber, in welche anderen Plug-Ins sich ein gegebenes Plug-In einhängen will, und inwiefern es selbst erweitert werden kann, ist in der `plugin.xml` festgehalten. Um diese Beziehungen organisieren zu können, haben die Eclipse-Entwickler das Konzept der **Extensions** und **Extension Points** eingeführt.

1.3 Extensions und Extension-Points

Plug-Ins erweitern die Plattform oder Anwendungen, die auf der Plattform aufbauen, und Plug-Ins werden erweitert. Das Konzept, mit dem die Eclipse-Entwickler verhindern, dass durch Rivalitäten zwischen Plug-Ins Fehlverhalten entsteht, heißt **Extension Points**. Gamma und Beck beschreiben Extension Points als "Plätze, an denen Funktionalität eingefügt wird" (p. 25)[11]'. Erweiterungen, auf Englisch **Extensions**, hingegen liefern "eingefügte Funktionalität" (p. 25)[11].

Während ein Extension Point einen Punkt in einem Plug-In markiert, an dem es erweitert werden kann, also ein strukturelles Merkmal eines Plug-Ins, das *erweitert wird*, bezeichnet der Begriff Erweiterung oder engl. Extension ein strukturelles Merkmal des *erweiternden* Plug-Ins. Unter einem "strukturellen Merkmal" ist in diesem Sinne ein Eintrag in die Manifest-Datei des Plug-Ins zu verstehen. Extensions und Extension Points müssen zueinander passen: Ein Plug-In kann nur eine Extension für einen bereits existierenden Extension Point bereitstellen (p. 435)[2].

Deklaration von Extension-Points. Damit Extensions geschrieben werden können, müssen zunächst einmal Extension Points existieren. Diese deklariert ein Plug-In in seinem Manifest. Dies geschieht, indem in die `plugin.xml` ein XML-Tag `<extension-point>` aufgenommen wird. Dieses teilt der Plattform die Existenz des neuen Extension Points mit (p. 85)[11]. Dieses Tag enthält weitere Elemente und Attribute, durch die der neue Extension-Point genauer beschrieben wird. Hier ist zunächst das Attribut `id` zu nennen, das den Extension-Point identifiziert. Durch Voranstellen des Names desjenigen Plug-Ins, das

den Extension-Point deklariert, wird ein eindeutiger Bezeichner für den Extension-Point gebildet. Unter den weiteren Attributen, die ein `<extension-point>`-Tag enhalten kann, ist vor allem noch die Angabe einer XML-Schema-Datei mit Vorgaben für die Deklaration der Extensions zu diesem Extension-Point von Bedeutung. Dadurch kann für die `plugin.xml` eines Plug-Ins, das diesen Extension-Point erweitern will, für den Abschnitt, der die Extension deklariert, ein Format festgelegt werden. Die Verwendung dieser abschnittsweisen Formatdefinition wird in den nächsten beiden Abschnitten besprochen.

Deklaration von Extensions. Wenn ein Plug-In einen Extension-Point erweitert, so muss es dies in seiner `plugin.xml` bekanntgeben. Dies geschieht über das `<extension>`-Tag. Ein `<extension>`-Element muss in jedem Falle ein Attribut `id` angeben, das die ID des erweiterten Extension-Points enthält [1]. Weiter gilt, dass Extension Points in der Regel mehrfach erweitert werden können. Gamma und Beck erheben es sogar zum Grundsatz für jeden Plug-In-Entwickler (p. 83)[11], dass Plug-Ins nach Möglichkeit so entwickelt werden sollten, dass ihre Extension-Points mehrere Extensions aufnehmen und korrekt verarbeiten können.

Schnittstelle zwischen Extension-Point und Extensions. Ein `<extension>`-Element in der Manifest-Datei eines Plug-Ins kann oder muss neben der ID des entsprechenden Extension-Points noch weitere Informationen beinhalten, je nachdem, welches Format die XML-Schema-Datei aus der Deklaration des Extension-Points für das `<extension>`-Element vorschreibt. Eine wichtige Rolle spielen diese zusätzlichen, für den Extension-Point spezifischen Informationen bei der Festlegung einer Schnittstelle zwischen Extension und Extension-Point. Dazu werden in der Regel *Callback*-Objekte [1] verwendet, die das erweiterte Plug-In aufruft. Dadurch geht der Programmfluss auf das erweiternde Plug-In über und erlaubt diesem somit, seine neue Funktionalität einzubringen. Um dafür zu sorgen, dass das erweiterte Plug-In weiß, wie es mit einem solchen Callback-Objekt umzugehen hat, kann das erweiterte Plug-In ein (Java-)`interface` definieren und mit dem entsprechenden XML-Schema dafür sorgen, dass erweiternde Plug-Ins ein Objekt bereitstellen, das ebendieses `interface` implementiert. D.h. die Definition des Extension Points wird so getroffen, dass jedes Plug-In, das eine Extension in diesen Extension-Point einfügen will, in seiner Manifest-Datei ein passend gestaltetes `<extension>`-Tag aufführen muss. Konkret wird dabei gefordert, dass das erweiternde Plug-In in seiner `plugin.xml` ein `class`-Attribut enthalten muss, dessen Wert auf den vollqualifizierten Namen einer Java-Klasse gesetzt wird, die das vom Extension-Point vorgeschriebene `interface` implementiert (pp. 83f)[11].

Beispiel: Erweiterung des Workbench-Menüs. An dieser Stelle ist ein Beispiel (s. pp. 42ff, [11]) angebracht, um diesen Mechanismus zu verdeutlichen. Wenn irgendein Plug-In mit dem Benutzer interagieren will, so benötigt es eine Repräsentation im Benutzerinterface, also in der Workbench. Dies wird in

der Regel ein Menüeintrag oder eine Schaltfläche in der Werkzeugleiste sein. Die Workbench erlaubt es Erweiterern, neue Einträge zu Menüs und Werkzeugleisten hinzuzufügen. Genauer gesagt stellt das Plug-In `org.eclipse.ui` einen Extension Point bereit, in den Extensions eingehängt werden können, die neue Menübefehle implementieren. Menübefehle werden von der Workbench zu sog. *Action Sets* zusammengefasst. Entsprechend heißt der Extension-Point, mit dem ein Plug-In neue Menübefehle in die Workbench einfügen kann, `org.eclipse.ui.actionSets`. Die Deklaration dieses Extension-Points verlangt nun (durch die angegebene XML-Schema-Datei), dass ein Plug-In, das diesen Extension-Point erweitern will, in die Definition einer Extension ein `<actionSet>`-Tag aufnimmt. Dieses muss ein oder mehrere `action`-Tags enthalten, welche die neuen Menübefehle beschreiben. Neben Informationen darüber, an welcher Stelle im Menü der neue Befehl angezeigt werden soll, müssen die `<action>`-Tags auch ein `<class>`-Attribut enthalten. Der Wert, auf den dieses Attribut gesetzt ist, muss der Name einer Java-Klasse sein, die das Interface `org.eclipse.ui.IWorkbenchWindowActionDelegate` implementiert. Die Workbench erzeugt eine Instanz dieser Klasse als Callback-Objekt für den Fall, dass der Benutzer das neu definierte Menü- oder Werkzeugleistenelement anklickt. Somit ist sichergestellt, dass ein Extension-Point bei seinen Extensions eine bekannte Schnittstelle vorfindet.

Abhängigkeiten. Hier wird aber die Notwendigkeit für einen weiteren Mechanismus zur Koordination zwischen Extension-Points und ihren Extensions klar: Ein Plug-In, das einen Extension-Point erweitert, wird unter Umständen dazu gezwungen, ein `interface` zu implementieren, dessen Definition sich in einem anderen Plug-In befindet. Dieses andere Plug-In kann dasjenige sein, das den Extension-Point deklariert hat, oder ein drittes Plug-In. Es existieren also *Abhängigkeiten* [1] zwischen Plug-Ins. Die Plattform muss einen Mechanismus bereitstellen, um diese Abhängigkeiten auflösen zu können. Wiederum findet das Manifest des Plug-Ins Verwendung. In die Datei `plugin.xml` wird ein Element `<requires>` eingefügt. Dieses kann als Unterelemente beliebig viele `<import>`-Elemente enthalten, von denen jedes mit einem Attribut `plugin`, dessen Wert die ID eines zu importierenden Plug-Ins ist, eine Abhängigkeit angibt (p. 67)[11]. Die Plattform trägt dann dafür Sorge, dass alle so importierten Plug-Ins vor dem importierenden Plug-In geladen werden. Im obigen Beispiel müsste also das erweiternde Plug-In, das einen neuen Menübefehl in die Workbench einfügen will, das Plug-In `org.eclipse.ui` als Abhängigkeit angeben, damit die Java- Klassendatei, die das Interface
`IWorkbenchWindowActionDelegate` aus dem Paket `org.eclipse.ui` enthält, bereits zur Verfügung steht, wenn das erweiternde Plug-In geladen wird. Auch bei den oben bereits erwähnten *Bibliotheks-Plug-Ins* ist es unmittelbar einleuchtend, dass Plug-Ins, welche auf die Bibliothek zugreifen, das *Bibliotheks-Plug-In* als Abhängigkeit angeben müssen.

Implementierung von Extension-Points. Wenn ein Plug-In einen neuen Extension-Point erstellen will, so muss es nicht nur die Deklaration desselben in

der Manifest-Datei enthalten. Es muss auch eine *Implementierung* des Extension-Points vorhanden sein [2]. Der erste Schritt dazu ist die Definition des Interface für die Callback-Objekte. An irgendeiner Stelle im Code wird das Plug-In, das den Extension-Point bereitstellt, von den Extensions Gebrauch machen wollen, ansonsten wäre es unnötig gewesen, den Extension-Point zu deklarieren. Um beim obigen Beispiel zu bleiben, wird das `org.eclipse.ui`-Plug-In als Reaktion auf einen Mausklick auf eines der Menü-Elemente eine Aktion ausführen wollen. Diese Aktion ist aber dem UI-Plug-In nicht bekannt, weshalb es an dieser Stelle auf seine Extensions zugreifen wird.

Das übliche Vorgehen für einen Extension-Point sieht so aus, dass sich das Plug-In, das den Extension-Point enthält, eine Liste mit allen für den Extension-Point angemeldeten Extensions besorgt. Diese Liste erhält das Plug-In von der Plattform (p. 88)[11]. Ebenfalls erhält das Plug-In von der Plattform eine Liste aller XML-Elemente, die im `<extension>`-Tag in der `plugin.xml` des erweiternden Plug-Ins enthalten sind. Aus dieser Liste liest die Implementierung des Extension-Points die `<class>`-Tags aus, um zu erfahren, welche Klassen zum Erzeugen von Callback-Objekten zur Verfügung stehen. Mit diesen Informationen kann der Extension-Point (wiederum mit Hilfe der Plattform) die Callback-Objekte erzeugen und die benötigten Methoden aufrufen, um die Extension ihre Dienste verrichten zu lassen. Dabei sollte die Implementierung des Extension-Points explizit überprüfen, ob die in den `<class>`-Tags angegebenen Klassen auch tatsächlich die geforderten Interfaces implementieren, bevor es die Callback-Methoden aufruft. Dies ist notwendig, um die Plattform vor fehlerhaften Erweiterungen zu schützen (pp. 89/91)[11].

Explizite Erweiterung. Durch das Konzept der Extension-Points wird der Vorgang des Erweiterns einer Anwendung offensichtlich komplizierter, als er es wäre, wenn man einfach nur die Klassen und Methoden der Anwendung für alle Erweiterer offen legen würde. Stattdessen wird gesonderter Aufwand getrieben, um nur an bestimmten Stellen und vom zu erweiternden Plug-In kontrolliert Erweiterungen zuzulassen. Es findet also nur *explizite Erweiterung* (p. 82)[11] statt.

Diese bringt gegenüber einer bloßen Veröffentlichung der API gewisse Vorteile bezüglich der Absicherung gegenüber fehlerhaftem Verhalten der Software auf der einen Seite als auch gegenüber späteren Erweiterungen. Durch einen Extension-Point legt ein Plug-In explizit einen Teil seiner API zur Erweiterung offen. Plug-Ins, die den Extension-Point erweitern wollen, können somit nur über eine wohldefinierte Schnittstelle Zugriff auf die bereits bestehende Funktionalität im erweiterten Plug-In erhalten. Dies gibt dem erweiterten Plug-In die Möglichkeit bevor es die Kontrolle an die Erweiterung abgibt, zu überprüfen, ob die Erweiterung auch tatsächlich die an sie gestellten Anforderungen bezüglich der Schnittstelle erfüllt.

1.4 Aktivierung von Plug-Ins

In engem Zusammenhang mit dem im letzten Abschnitt besprochenen Aufruf von Callback-Methoden durch die Implementierung eines Extension-Point steht die Aktivierung des Plug-In, welches die Extension enthält. Denn für den sicheren Aufruf einer solchen Callback-Methode muss nicht nur sichergestellt sein, dass dieselbe existiert, sondern auch, dass sie arbeiten kann. Soll z.B. eine Callback-Methode Daten aus einer Datei auf der Festplatte auslesen, so muss sichergestellt sein, dass die Datei auch geöffnet ist und der Callback-Methode der Zugriff auf ein Datei-Handle möglich ist. Es kann also sein, dass vor der Ausführung der ersten Methode aus einem Plug-In Initialisierungsarbeiten notwendig sind. Ein Plug-In zu verwenden, bedeutet nicht nur, Code aus den vom Plug-In gelieferten Klassendateien aufzurufen, sondern auch dem Plug-In die Möglichkeit zu solchen Initialisierungen zu geben. Dass diese stattfinden wird durch das Konzept der Plug-In-Aktivierung sichergestellt, um das es in diesem Abschnitt geht.

Die `Plugin`-Klasse. In der Plattform existiert eine Klasse `Plugin`, welche ein Plug-In in der Runtime repräsentiert. Diese Klasse enthält Methoden `start()` und `stop()`, die zur Aktivierung und zur Deaktivierung des Plug-Ins aufgerufen werden. Diese Klasse `Plugin` nimmt die Aufgaben, die während des *Plug-In-Lebenszyklus* (p. 57)[11] anfallen, wahr. Wenn ein Plug-In eine neue Klasse bereitstellt, die `Plugin` erweitert, so kann sie in den überschriebenen Methoden `start()` und `stop()` die für sein Funktionieren notwendigen Initialisierungen vornehmen und Ressourcen reservieren und diese bei seiner Deaktivierung auch wieder freigeben.

Besondere Beachtung wurde bei der Entwicklung von Eclipse auch der Frage geschenkt, wann ein Plug-In aktiviert werden soll. Eine Eclipse- Installation kann je nach Anzahl der installierten Erweiterungen hunderte von Plug-Ins enthalten (p. 35)[11], von denen aber nicht alle auch tatsächlich vom Benutzer verwendet werden. Würde die Plattform beim Start grundsätzlich alle Plug-Ins laden, so könnte der Startvorgang eine erheblich Zeit in Anspruch nehmen (p. 35)[11]. Um dies zu vermeiden und der Tatsache Rechnung zu tragen, dass die sofortige Aktivierung aller Plug-Ins in der Regel nicht notwendig ist, verwendet die Plattform zum Laden der Plug-Ins eine Technik, die als *lazy loading* bezeichnet wird. Gamma und Beck formulieren dies sogar als Grundsatz für die Plug-In-Entwicklung (p. 36)[11]: *Erweiterungen werden nur geladen, wenn sie benötigt werden.*

Zeitpunkt der Aktivierung Klassen, die von einem Plug-In in die Plattform eingeführt werden, können nicht so ohne Weiteres instanziiert werden, wie dies beispielsweise bei Klassen aus der Standard-Java-Laufzeitbibliothek der Fall ist. Da die Plattform *lazy loading* anwendet, lädt sie die Klassen eines Plug-Ins nicht, bevor dieses nicht gebraucht wird. Wenn aber der Zeitpunkt gekommen ist, dass eine Instanz einer Klasse aus einem bisher nicht benutzten Plug-In erzeugt werden soll, so kann dies nicht direkt über den Aufruf eines Konstruktors der Klasse

geschehen – denn die Klasse und damit der Konstruktor sind vielleicht noch nicht geladen worden. Die Instanziierung wird daher über Methoden von Klassen der Plattform-Runtime vorgenommen, die zunächst prüfen, ob das gefragte Plug-In schon aktiviert worden ist. Falls nicht, wird die Aktivierung vorgenommen und das angeforderte Objekt zurückgeliefert. Aktivierung eines Plug-Ins heißt dabei, dass *alle* Klassen, die zu diesem Plug-In gehören, geladen werden. Noch vorher werden allerdings alle in der `plugin.xml` als Abhängigkeit angegebenen Plug-Ins aktiviert. Schließlich ruft die Runtime zum Abschluss der Aktivierung die `start`-Methode der `Plugin`-Klasse auf.

Trennung von Deklaration und Implementierung. Um den Grundsatz des *lazy loading* verwirklichen zu können, werden in Eclipse die Deklaration und die Implementierung eines Plug-Ins strikt getrennt (p. 36)[11]. Die Implementierung findet natürlich in den Java-Klassendateien statt und wird vom Entwickler des Plug-Ins ausprogrammiert. Der Ort der Deklaration hingegen ist die `plugin.xml`.

Diese genug Informationen, um die Struktur des Plug-Ins deutlich zu machen: Die Manifest-Datei teilt der Platform mit, welche Extension-Points das Plug-In erweitern will, von welchen Plug-Ins es abhängt und welche Extension-Points es selbst zur Anwendung hinzufügt. Dadurch wird es der Plattform-Runtime ermöglicht, einen Überblick über die vorhandenen Plug-Ins zu erhalten und festzustellen, was zu tun ist, wenn der Benutzer die Aktivierung eines zusätzlichen Plug-Ins anfordert. Dies wird in der Regel geschehen, indem der Benutzer eine Aktion auf der Benutzeroberfläche, etwa die Auswahl eines Menüeintrags, ausführt, welche die Aktivierung eines Plug-Ins nach sich zieht.

Um dies zu verdeutlichen, greifen wir ein weiteres Mal das Beispiel auf, in dem ein neuer Befehl in ein Menü oder eine Werkzeugleiste eingefügt wird. Wählt der Benutzer die neue Aktion an, so verlangt er damit eine Aktion seiner Anwendung. An diesem Punkt sind die Möglichkeiten der Deklaration erschöpft, denn das `org.eclipse.ui`-Plug-In weiß nicht, wie es den Menübefehl, den ein fremdes Plug-In seiner Deklaration zu Folge anbietet, verarbeiten soll. Die Benutzerschnittstelle weiß jedoch aus der `plugin.xml` desjenigen Plug-Ins, das den Menübefehl eingefügt hat, welche Klasse sie instanziieren muss, um ein Callback-Objekt für diesen Menübefehl zu erhalten. Die Erzeugung dieses Callback-Objektes geschieht wie im letzten Abschnitt beschrieben durch Aufrufen einer Methode der Plattform, die bei Bedarf die Aktivierung des betreffenden Plug-Ins veranlasst.

An diesem Beispiel wird deutlich, wie es die Trennung von Deklaration und Implementierung ermöglicht, den *lazy loading*-Grundsatz einzuhalten: Das Manifest liefert genug Informationen, um die Aktivierung eines Plug-Ins so lange wie möglich hinauszögern zu können. Beispielsweise ist im Falle eines neuen Menübefehls alles, was im Benutzerinterface angezeigt werden soll, also Pfad im Menü, Text und Icons, im Manifest eingetragen und kann auf diesem Weg der Plattform zugänglich gemacht werden, ohne irgendwelchen Code aus dem Plug-In ausführen zu müssen.

2 Vorgehen bei der Erstellung eines Plug-Ins

Nachdem im ersten Teil dieser Arbeit die Grundzüge der Eclipse-Plug-In-Architektur dargelegt wurden, behandelt dieser zweite Teil nun die praktische Anwendung dieses Wissens an Hand einiger Beispiele. Dieser Teil der Arbeit kann keineswegs alle Fälle, die bei der Entwicklung für die Eclipse-Plattform auftreten können, abhandeln. Das hier vorgestellt Beispiel soll verdeutlichen, wie die grundlegenden Mechanismen zur Erweiterung von Eclipse - Extensions und Extension Points - zusammen eingesetzt werden müssen, um ein neues Plug-In in die Plattform zu integrieren.

Als Beispiel dient hier ein "Hello World"-Plug-In. Dieses Plug-In greift den im ersten Abschnitt schon als Beispiel herangezogenen Anwendungsfall, dass von einem Plug-In ein neuer Befehl in das Menü der Workbench eingefügt werden soll, wieder auf: Das Beispiel- Plug-In basiert im Wesentlichen auf dem "Hello-World"-Plug-In, das die Plug-In-Entwicklungsumgebung des Eclipse-Projektes, die sog. *PDE* (*Plug-in Development Environment*) generiert. Das Beispiel-Plug-In erzeugt ein neues Menü "Sample Menu", das als einzigen Eintrag "Sample Action" enthält. Außerdem wird die "Sample Action" noch in die Werkzeugleiste eingefügt. Als Reaktion auf die Aktivierung dieses Menübefehls zeigt das Plug-In eine Nachrichtenbox, welche die Hallo-Welt-Nachricht enthält.

Unsere Erweiterung des von der PDE generierten Basis-Plug-Ins besteht darin, einen Extension-Point `messageProviders` anzulegen, um das Erzeugen der anzuzeigenden Nachricht an weitere Plug-Ins zu delegieren. Alle Extensions müssen das Interface `IMessageProvider` implementieren, über dessen Methode `getMessage` die Nachricht, die in der Nachrichtenbox erscheinen soll, abgefragt wird.

Die Definition des Extension-Points `messageProviders` in der `plugin.xml`, die den neuen Extension-Point im System bekannt macht, sieht so aus:

```
<extension-point id="messageProviders" name="Message Providers"
  schema="schema/messageProviders.exsd"/>
```

Dabei ist der Wert des Attributs `id` der String, mit dem der Extension-Point von den erweiternden Plug-Ins referenziert wird. Das Attribut `schema` verweist, wie bereits aus dem ersten Abschnitt bekannt, auf eine XML-Schema-Datei, die das Format der Deklarationen der Extensions spezifiziert.

Damit ein neuer Eintrag ins Menü eingefügt wird, muss, wie bereits gesagt, der Extension-Point `org.eclipse.ui.actionSets` erweitert werden. Dies schlägt sich in der `plugin.xml` folgendermaßen nieder:

```
<extension
      point="org.eclipse.ui.actionSets">
   <actionSet
      <menu
            label="Sample &amp;amp;Menu"
            id="sampleMenu">
         <separator
               name="sampleGroup">
```

```
            </separator>
         </menu>
         <action
               label="&amp;Sample Action"
               icon="icons/sample.gif"
               class="helloworld.HelloWorldAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               toolbarPath="sampleGroup"
               id="helloworld.HelloWorldAction">
         </action>
      </actionSet>
   </extension>
```

Dieser XML-Code definiert die Erscheinung des neuen Menübefehls (Menüpfad, Icon, Label usw.) und gibt zudem noch die Klasse für das Callback-Objekt (`helloworld.HelloWorldAction`) bekannt.

Nun muss der Extension-Point implementiert werden, um Extensions zu laden. Dazu wird in der Datei `HelloWorldAction.java` neben einigen anderen Methoden die Methode `run()` definiert, die aufgerufen wird, wenn der Benutzer das Menüelement "Sample Action" anklickt.

Unser Code dafür sieht folgendermaßen aus:

```
public void run(IAction action) {
    Vector<IMessageProvider> providers;

    providers = getMessageProviders();
    for( IMessageProvider p : providers) {
        // for each extension, display its message:
        MessageDialog.openInformation(
            window.getShell(),
            "HelloWorld Plug-in",
            p.getMessage());
    }
}
```

Die `run()`-Methode ruft zunächst `getMessageProviders()` auf, um eine Liste von Objekten zu erhalten, die Nachrichten liefern können. Die Methode `getMessageProviders()` ist die Implementierung des Extension-Points `messageProviders`. Sie fragt alle Extensions ab, die für unseren Extension-Point angemeldet sind, und erzeugt die von ihnen bereitgestellten Callback- Objekte. Danach zeigt `run()` (um alle Extensions gleich zu behandeln) für jedes gefundene Callback-Objekt ein eigenes Fenster an, in dem die jeweilige Nachricht angezeigt wird.

Die Implementierung von `getMessageProviders()` benutzt dabei die Plugin-Registry, die von der Plattform bereitgestellt wird und Informationen über alle in der Eclipse-Installation vorhandenen Plug-Ins vorhält. Diese Informationen gewinnt die Plattform-Runtime, indem sie die Manifest-Dateien aller Plug-Ins (die alle im Ordner `plugins` und dessen Unterordnern liegen müssen) ein-

liest (p. 64)[11]. Von der Registry fragt die `getMessageProviders()`-Methode alle Elemente ab, die Kindelemente des `<extension>`-Tags zum Extension-Point `messageProviders` sind, und erzeugt mit dem passenden Element ein Callback-Objekt über die Methode `createExecutableExtension()`.

Der im Folgenden gezeigte Code für `getMessageProviders()` basiert auf einem Listing in (p. 449)[2]:

```
private Vector<IMessageProvider> getMessageProviders() {
    IExtensionRegistry registry;
    IConfigurationElement[] elements;
    Vector<IMessageProvider> providers =
                    new Vector<IMessageProvider>();
    registry = Platform.getExtensionRegistry();
    elements = registry.getConfigurationElementsFor(
                     "HelloWorld.messageProviders");
    for(IConfigurationElement elem : elements) {
        if(elem.getName().equals("message-provider")) {
            Object mp;
            try {
                mp = elem.createExecutableExtension("class");
            }
            catch(CoreException e) {
                mp = null;
            }
            if(mp instanceof IMessageProvider) {
            providers.add((IMessageProvider) mp);
        }
    }
    return(providers);
}
```

Damit sichergestellt ist, dass `getMessageProviders` mindestens einen String liefert, gibt es in unserem Plug-In natürlich auch eine Default-Implementierung eines Message-Providers, d.h. das Plug-In definiert selbst eine Extension zu seinem eigenen Extension-Point. Diese Default-Implementierung ist jedoch so simpel (sie liefert lediglich den String “Hello World” zurück), dass wir sie hier nicht abdrucken.

3 Fazit

Zum Abschluss soll nun die Plug-In-Architektur noch kurz aus einem anderen Blickwinkel als bisher betrachtet werden. Im letzten Abschnitt dieser Arbeit soll es nicht mehr darum gehen, was ein Eclipse-Entwickler wissen muss, um neue Plug-Ins zu schreiben, sondern um Vor- und Nachteile der Eclipse-Architektur.

Zunächst muss gesagt werden, dass Eclipse eine extrem mächtige Plattform ist. Auf der Grundlage von Eclipse können sehr unterschiedliche und sehr mächtige Anwendungen entwickelt werden. Die vom einfachen Texteditor (als zugegebenermaßen etwas pathologisches Beispiel) bis zur vollausgestatteten IDE.

Aber auch gänzlich andere Anwendungen wie beispielsweise ein Webbrowser [8] können realisiert werden. Manche halten die Architektur von Eclipse sogar für Bildbearbeitungsprogramme oder Tonstudios geeignet (p. 363)[3].

Mit dieser Mächtigkeit und Flexibilität einher geht eine enorme Nützlichkeit, die wahrscheinlich einen großen Teil des Erfolges des Eclipse-Projektes ausmacht. Am intensivsten wird Eclipse im Bereich der Software-Entwicklung eingesetzt. Mit dem JDT-Paket steht dafür bereits eine ausgezeichnete Grundlage zur Verfügung, die sich viele Entwickler zu Nutze machen, indem sie eigene Tools in die bestehende IDE integrieren. Dies macht den Vorzug einer offenen Plattform gegenüber einer monolithischen Entwicklungsumgebung aus, dass man nicht auf den bei Auslieferung vorhandenen Funktionsumfang beschränkt bleibt, sondern sein Entwicklungswerkzeug an die eigenen Bedürfnisse anpassen kann.

Da die Architektur von Eclipse nicht darauf ausgelegt ist, eine Anwendung bloß "hinterher", d.h. nachdem die Entwicklung abgeschlossen ist, zu erweitern, sondern Anwendungen komplett aus Plug-Ins aufzubauen, ist das Plug-In-System fest im Kern einer Eclipse-Anwendung verzahnt (um nicht gar zu sagen, es *ist* der Kern der Anwendung). Dies erlaubt auch tiefgreifende Änderungen an vorhandenen Anwendungen. Bei dem bereits in der Einleitung genannten Texteditor Gedit [12] ist dies nur eingeschränkt möglich. Da dieser Editor als monolithische Anwendung konzipiert wurde und irgendwann ein Plug-In-System "nachgerüstet" wurde, besteht leidlich wenig Koordination zwischen dem Kern der Anwendung und den Plug-Ins: Über die Plug-In-API hat ein Plug-In Zugriff auf die Datenstrukturen des Programms. So wird beispielsweise ein neuer Menübefehl eingefügt, in dem man sich einen Zeiger auf das Menüobjekt holt und und das neu erzeugte Item einfach einfügt. Genauso wäre es aber möglich vom Plug-In aus das Menüobjekt freizugeben und somit das ganze Menü einfach verschwinden zu lassen.

In Eclipse wird so etwas durch das Konzept der explizit veröffentlichen API weitgehend unterbunden. Sicherheit gegenüber den Erweiterungen ist somit ein weiterer Pluspunkt für Eclipse. Wie in Abschnitt 2 bereits angedeutet, ist in der Eclipse-Plattform nicht nur Sicherheit vor fehlerhaften Erweiterungen, sondern auch eine gewisse Versionssicherheit verwirklicht, indem Interface und Implementierung einer Schnittstelle getrennt werden: Durch die Trennung ist es für die Entwickler leichter, ihre Implementierung ohne Änderung der veröffentlichten Schnittstelle zu ändern, als es der Fall wäre, wenn man die ganze Implementierung öffentlich machen würde.

Mit der Art und Weise, wie diese Trennung von Interface und Implementierung in Eclipse durchgeführt wurde, hat man sich aber auf Java als einzige Programmiersprache für Erweiterungen festgelegt: Da Java-Interfaces das einzige Mittel sind, das Interface zwischen zwei Plug-Ins festzulegen, kann es keine Plug-Ins ohne Java geben. Dies muss aber nicht so sein. Beispielsweise bietet das CORBA-Framework [13] die Möglichkeit, Komponenten, die in unterschiedlichsten Programmiersprachen implementiert sind, zusammenarbeiten zu lassen.

CORBA lässt Eclipse noch in einem weiteren Punkt hinter sich: Eclipse ist lokal. Da Plug-Ins nur gefunden werden, wenn sie in einem bestimmten Verzeich-

nis liegen, kann folglich kein Plug-In geladen werden, das nicht auf demselben Rechner gespeichert ist, wie die Instanz der Eclipse-Runtime, die es ausführen soll. CORBA hingegen bietet ein verteiltes Protokoll, was bedeutet, dass die Implementierung eines bestehenden Interface von einer Komponente, die irgendwo in einem erreichbaren Netzwerk liegt, verwendet werden kann. Dies bedeutet noch einmal wesentlich mehr Flexibilität, als sie die Eclipse-Plattform bieten kann.

Zum Abschluss dieser Vergleiche muss man jedoch sagen, dass alle drei Systeme - Gedit, Eclipse und CORBA - mit unterschiedlichen Zielsetzungen entwickelt worden sind: Gedit sollte nur kleine Erweiterungen für eine bestehende Software ermöglichen, während Eclipse darauf ausgelegt wurde, ganze Software-Projekte erst mit Hilfe der Frameworks aus der Eclipse-Plattform aufzubauen. CORBA dagegen wurde als ein einzelnes Framework angelegt, das nicht gleichzeitig einen Nährboden für komplette Anwendungen bieten kann, wie es Eclipse tut, indem Prozesskontrolle, Dateisystemzugriffe und vieles weitere von der Plattform übernommen wird.

Literatur

1. Bolour, A.: Notes on the eclipse plug-in architecture.
   http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
   letzter Zugriff: 05.12.2006
2. Brüssau, K., Widder, O., eds.: Eclipse – Die Plattform: Enterprise-Java mit Eclipse 3.1. 2 entwickler.press, Frankfurt (2006)
3. Daum, B.: Java-Entwicklung mit Eclipse 3.1: Anwendungen, Plug-Ins und Rich Clients. dpunkt-Verlag, Heidelberg (2005)
4. Eclipse-Dokumentation (v. 3.2.1).
   http://help.eclipse.org/help32/index.jsp
   letzter Zugriff: 21. 02. 07 Platform Plug-In Developer Guide/Programmer's Guide/Welcome to Eclipse/What is Eclipse?
5. Eclipse-Dokumentation (v. 3.2.1). Platform Plug-In Developer Guide/Programmer's Guide/Platform overview
6. Eclipse-Dokumentation (v. 3.2.1). Platform Plug-in Developer Guide/Programmer's Guide/Runtime Overview/The runtime plug-in model
7. Eclipse-Dokumentation (v. 3.2.1). Platform Plug-In Developer Guide/Programmer's Guide/Packaging and delivering Eclipse based products
8. Eclipse-Dokumentation (v. 3.2.1). Platform Plug-in Developer Guide/Programmer's Guide/Building a Rich Client Platform Application/The browser example
9. Eclipse Plattform Technical Overview, International Business Machines Corp., 2006
   http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf
   letzter Zugriff: 20. 02. 07
10. Homepage des Eclipse-Projektes
    http://www.eclipse.org/platform/overview.php
    letzter Zugriff: 20. 01. 07
11. Gamma, E., Beck, K.: Eclipse erweitern: Prinzipien, Patterns und Plug-Ins. Addison-Wesley, München (2004)

12. Homepage des Gedit-Projektes
http://www.gnome.org/projects/gedit
letzter Zugriff: 20. 01. 07
13. The Object Management Group: Corba basics.
http://www.omg.org/gettingstarted/corbafaq.htm
letzter Zugriff: 20.12.06

BIRT

Business Intelligence and Reporting Tools für Eclipse

Tobias Dencker

Fakultät für Informatik
Universität Karlsruhe

Zusammenfassung Die zunehmende Daten- und Informationsflut ist ohne den Einsatz von Business Intelligence Werkzeugen kaum zu bewältigen. Als solches bietet Eclipse BIRT eine effiziente Lösung zur Ausstattung einer Java Anwendung mit Reporting-Funktionalität. Nutzbarkeit und Wert vieler Anwendungen hängen oft unmittelbar mit der strukturierten und übersichtlichen Informationsausgabe zusammen. Mussten Software-Entwickler während der letzten Jahre noch Ihre eigenen Werkzeuge für das Erstellen von Berichten aufwendig entwickeln, ist dies nun mit Hilfe von Werkzeugen wie BIRT wesentlich leichter zu bewerkstelligen. Dieser Artikel bietet eine eingehende Untersuchung sowie kritische Betrachtung des Eclipse Business Intelligence Werkzeugs BIRT.

1 Einführung

1.1 Erläuterung des Begriffes „Business Intelligence“

Der Begriff der so genannten „Business Intelligence“ [1] ist erst in den letzten Jahren aus dem Fachbereich der Wirtschaftsinformatik hervorgegangen und unterliegt deswegen noch keiner einheitlichen Definition. Er befasst sich mit der Problematik der stetig wachsenden Daten- und Informationsflut im Unternehmensbereich.

Als grundlegende Eigenschaft der Business Intelligence ist die Komprimierung unübersichtlicher Datenberge zu aussagekräftigen Dateneinheiten z.B. in Form von Geschäftsberichten hervorzuheben. In Kombination mit der zeitnahen Verfügbarkeit solcher Berichte können Unternehmen mögliche Trends und Probleme in ihrem Geschäftsfeld schon frühzeitig erkennen und auf Grundlage der so gewonnenen Daten schlüssige und nachvollziehbare Entscheidungen treffen. Gerade in Unternehmensbereichen wie „Corporate Finance“ oder auch dem „Customer Relationship Management“ ist die schnelle Verfügbarkeit von Geschäftsberichten von großem Nutzen, wenn nicht sogar notwendig für die Wettbewerbsfähigkeit des Unternehmens.

1.2 Werkzeuge für Business Intelligence Reporting

Business Intelligence bedeutet demnach stark vereinfacht, die Fähigkeit der Abstraktion großer Datenmengen mit dem Ziel, eine fundierte Wissensbasis für die

Entscheidungsfindung zu schaffen. Durch Anwendung von Business Intelligence erhält man sinnvolle und strukturierte Informationen, im Grunde Berichte. Folglich ist es nicht verwunderlich, dass der Ruf in Unternehmen nach Software zur weitgehenden automatisierten Generierung von Geschäftsberichten immer lauter wird.

Die Berichtsdaten müssen strukturiert, Geschäftslogik eingefügt und das Layout angepasst werden. Mussten Software-Entwickler während der letzten Jahre noch Ihre eigenen Werkzeuge für das Erstellen von Berichten aufwendig entwickeln, ist dies nun mit Hilfe von Werkzeugen wie *Eclipse BIRT* [2] wesentlich leichter zu bewerkstelligen. Neben der grundsätzlichen Forderung nach einfacher Handhabung und hoher Kompatibilität von Software sind bei Berichtswerkzeugen die folgenden Merkmale als wichtige Kriterien einzuordnen:

- Hohe Flexibilität und Freiheit in der Berichtgestaltung (neue Anforderungen lassen sich schnell und ohne großen Aufwand realisieren),
- Neben tabellarischer Darstellung auch die Möglichkeit der grafischen Aufbereitung von Daten,
- Integrationsmöglichkeiten in bestehende Systeme (z.B. Nutzung der bestehenden Dateninfrastruktur),
- zeitnahe Verfügbarkeit der Berichte (z.B. Abrufbarkeit über Webserver) und
- zeitnahe Aktualisierung von Berichtsinhalten (z.B. direkte Anbindung an Datenserver).

Bevor im 3. Abschnitt näher auf die zentralen Merkmale des BIRT Plugins (Prozessfluss, Designer Perspektive, Laufzeitkomponenten und Erweiterbarkeit) eingegangen wird, wird im Folgenden die Rolle des BIRT-Projekts im größeren Rahmen der *Eclipse Foundation*, der Organisation für die Leitung und Entwicklung von Eclipse, geklärt. Im letzten Abschnitt folgt eine Zusammenfassung der Vor- und Nachteile von BIRT sowie ein abschließendes Fazit.

2 Eclipse BIRT Projekt

2.1 Eclipse Plattform

Das *Eclipse Projekt* [3] ist mit dem Ziel entstanden, seinem Benutzer eine robuste und umfassende Entwicklungsplattform zur Verfügung zu stellen. War Eclipse in früheren Versionen noch enger mit der Java IDE (Integrated Development Environement) verknüpft, versteht man mittlerweile unter der Eclipse Plattform nur noch den Kern mit einem rudimentären Funktionsumfang. Der Einsatz der Plattform ist jedoch aufgrund der enormen Erweiterbarkeit durch Plugins gemäß dem Rich-Client-Plattform Konzepts in den verschiedensten Bereichen denkbar, wobei die Java Entwicklungsumgebung wohl nach wie vor die bekannteste Anwendung von Eclipse ist.

Eclipse ist in Java programmiert und basiert auf dem Open Services Gateway Initiative (OSGi) Standard. Zudem wird die graphische Oberfläche durch das Standard Widget Toolkit (SWT), einem GUI Framework, gebildet, welches

die nativen GUI-Komponenten des zugrunde liegenden Betriebssystems benutzt. Somit ist Eclipse auf allen gängigen Betriebssystemen einsetzbar.

Über eine integrierte Update-Funktionalität lassen sich eine Vielzahl von Plugins ohne weiteres über ein beliebigen Update-Server herunterladen und installieren. Mittlerweile sind mehrere Eclipse Distributionen wie Callisto verfügbar, die nach der Installation bereits eine Auswahl an Plugins mitliefern.

2.2 Top-Level Projekt BIRT

Bei einem dieser Plugins für die Eclipse Plattform handelt es sich um **BIRT - B**usiness **I**ntelligence and **R**eporting **T**ools. BIRT wird mit dem Leitgedanken entwickelt dem Eclipse-Entwickler ein mächtiges Werkzeug zur Ausstattung einer Java oder Java 2 Enterprise Edition (J2EE) Web-Applikation mit Reporting-Funktionalität zur Seite zu stellen. BIRT gilt als eines von 10 Top-Level Projekten der Eclipse Foundation [4], die mehrere kleinere Projekte unter einem Dach vereinen und besondere Aufmerksamkeit bei Entwicklungsarbeit von Eclipse erhalten. Seine Zielsetzung ist es den Business Intelligence Bereich für Eclipse Anwender noch leichter zugänglich zu machen. Inzwischen hat die Eclipse Foundation die Version 2.1.1 des Eclipse BIRT-Projektes veröffentlicht.

Ausgehend von einer Initiative des kommerziellen Business Reporting Anbieters Actuate wurde BIRT Mitte 2004 als erstes Eclipse Projekt im Business Intelligence Bereich ins Leben gerufen. Bewusst wurde BIRT im Rahmen der Eclipse Foundation als Open-Source Projekt etabliert, was die Software und dessen Quelltext kostenlos und frei verfügbar macht.

Unter der Führung von Actuate wird das Eclipse-BIRT-Projekt unter anderem von IBM, Pentaho, Scapa Technologies, Zend und InetSoft vorangetrieben. Zudem besteht eine Kooperation mit weiteren Eclipse Projekten wie z.B. dem Test und Performance Tools Platform-(TPTP-)Projekt. Insbesondere in diesem Rahmen hat Intel angekündigt, dass das TPTP-Projekt auf BIRT zurückgreifen wird und man Erweiterungen dafür entwickeln möchte.

Trotz des starken Einflusses von Actuate profitiert die Entwicklung des BIRT Plugins vor allem von der Mitarbeit und dem Feedback der sehr aktiven Eclipse Community, ohne die der bisherige Erfolg nicht denkbar gewesen wäre.

Zur Frage nach der Motivation eines kommerziellen Unternehmens zur Beteiligung an einem Open-Source Projekt führte *Actuate Chef Mike Thomas in einem Interview mit dem Eclipse Magazin* [5] die folgenden Argumente an:

- Zugewinn bei Reporting-Tools-Anwendern schafft einen größeren Markt, auf dem Actuate seine kommerziellen Produkte anbieten kann.
- Zur Akzeptanz der Open-Source Software bietet sich Actuate als kommerzieller Service-Partner an, der Schadensersatz sowie die Instandhaltung und Pflege des BIRT-Codes garantiert.
- BIRT Beteiligung dient als „Übergang“ zum Actuate-Produktportfolio

3 Hauptmerkmale von BIRT

3.1 Prozessfluss

Im Arbeitsprozess mit BIRT nimmt der Report Designer (1) die zentrale Stellung ein, indem er mehrere Plugins innerhalb von Eclipse zu einer Perspektive vereint, die für die Berichtgestaltung die notwendigen Struktur- und Layout-Werkzeuge zur Verfügung stellen. Ein weiteres wichtiges Modul ist der Chart Builder, der für die graphische Darstellung von Daten in Form von Diagrammen nutzbar ist. Zunächst wird das Bericht Design (RPTDesign) im XML Format erstellt und auch in derselben Form abgespeichert, was durch BIRT Design Engine (DE) erledigt wird.

Aus der XML Datei kann dann mit der Report Engine (2) der Bericht generiert und in mehreren Formaten z.B. HTML oder PDF ausgegeben werden. Beim Erstellen des Berichts durch die Report Engine (RE) wird eine Verbindung zu den im XML Report Design definierten Datenquellen geöffnet und darüber die Berichtselemente mit den verknüpften Datensätzen gefüllt. Die Diagrammanfertigung (Charting Services) (3) wird von der Chart Engine (CE) übernommen und das fertige Diagramm an die Report Engine zum Einbinden in das Berichtsdokument weitergeleitet.

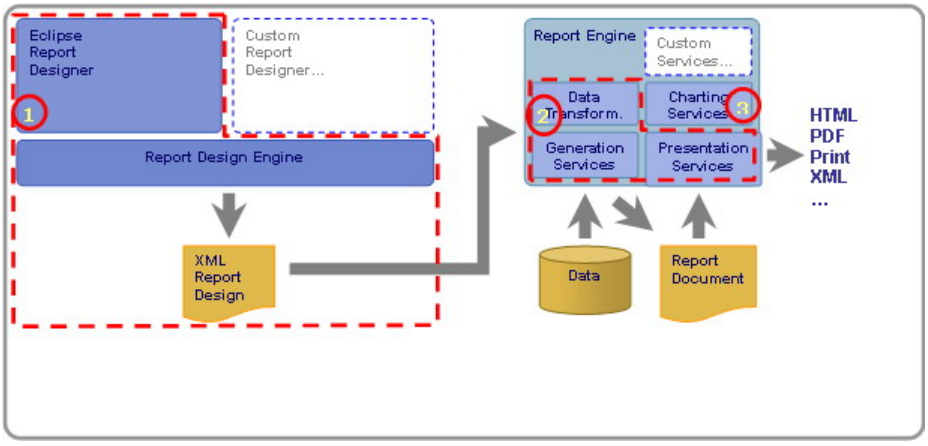


Abbildung 1. BIRT Arbeitsprozess, Quelle: Actuate [6]

Mit welcher Funktionalität und Spezifikation diese Komponenten aufwarten und inwieweit sie sich in andere Java Applikationen integrieren lassen, wird in den folgenden Abschnitten genauer behandelt.

3.2 Design Perspektive

Die BIRT Design Perspektive wird innerhalb von Eclipse zum Aufbau und zur Gestaltung von Berichten im XML Format (RPTDesign) verwendet. Dazu werden folgende APIs genutzt:

- Design Engine (DE) für den Report Designer
- Chart Engine (CE) für den Chart Builder

Perspektive in Eclipse

Nach der Installation des BIRT Plugins lassen sich keine offensichtlichen Veränderungen in Eclipse feststellen. Erst beim Erstellen eines „Report Projekt“ präsentiert sich Eclipse BIRT mit seiner Design Perspektive, die ansonsten auch über „Fenster“>“Perspektive öffnen“ erreichbar ist. Zu Beginn eines Projektes hat der Entwickler die Möglichkeit mit einem Blanko-Bericht neu zu starten, ein Template aus einer Liste mit Standard-Reports auszuwählen oder ein bereits erstellten Bericht wieder zu verwenden. Mehrere Berichtstypen werden unterstützt: Listen, Tabellen, Diagramme, Briefe, Dokumente, gemischte Berichte.

Die Grundfunktionen für Eclipse Projekte sind auch in einem Report Projekt verfügbar: Ein Navigationsfenster zur Anzeige der einzelnen Dateien des Projekts wird bereit gestellt. Ein Gliederung in Baumstruktur verschafft Übersicht über die einzelnen Berichtsbausteine.

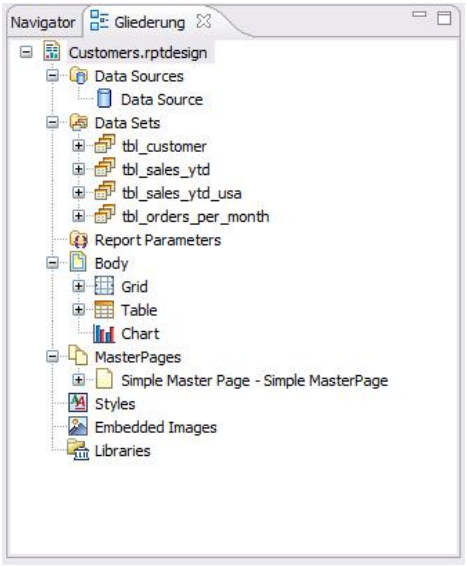


Abbildung 2. Gliederung eines Report Projektes

Grundstruktur eines Berichtes

Ein BIRT Bericht besteht aus verschiedenen Elementen und Funktionen - siehe vorherige Abbildung. Die Berichtsdaten können aus mehreren beliebigen Quellen stammen. Zudem können sie durch Datentransformation und Geschäftslogik in das gewünschte Format gebracht werden, bevor sie schließlich übersichtlich und ansprechend in einem Bericht dargestellt werden. Die Berichtsstruktur wird - vereinfacht ausgedrückt - durch zwei Komponenten beeinflusst, die sich auch in den einzelnen Berichtsbausteinen wieder finden:

Datenkomponente bestimmt die im Bericht erscheinenden Datensätze und deren Verarbeitung

Layoutkomponente bestimmt die Berichtgestaltung, also die Art und Weise der Präsentation der Datensätze

Im Hintergrund befindet sich eine **„programmierbare Ebene" zum Einbinden von Geschäftslogik** in den Bericht. Daten- und Layoutkomponten des Berichts können somit fast beliebig angepasst werden.

3.3 Report Designer

BIRT ermöglicht es unter Berücksichtigung dieser Sachlage die Daten- und Layoutkomponente zunächst getrennt voneinander zu spezifizieren, indem mittels des Report Designers entsprechende Werkzeuge zur Verfügung gestellt werden. So wird für ein Höchstmaß an Ordnung und Übersichtlichkeit gesorgt, aber gleichzeitig auch Flexibilität und Freiheit beim Gestalten eines Berichtes garantiert: Daten- und Layoutkomponenten lassen sich effektiv verbinden, aber auch schnell an neue geschäftliche Gegebenheiten anpassen. Die Informationen zu den Komponenten der Berichtsbausteine werden als Report Design im XML Format in einer RPTDesign Datei gespeichert.

Data Explorer

Zur **Verwaltung der Datenkomponenten** wird dem Benutzer der Data Explorer als Werkzeug zur Seite gestellt:

Data Sources definieren Datenquellen mit den dazugehörigen Verbindungen. BIRT unterstützt folgende Datenquellen:

- Java DataBase Connectivity (JDBC) Datenbanken
- XML Datenquellen
- Java Objekte aus bestehenden Applikationen
- Extraktion aus flachen Dateien (z.B. Comma Separated Values (CVS))
- Durch Scripting verfügbar gemachte Datenquellen

Nach dem Einbinden in den Data Explorer sind diese Datenquellen verfügbar, müssen aber für die Verknüpfung mit dem Bericht erst noch in Datensätze unterteilt werden.

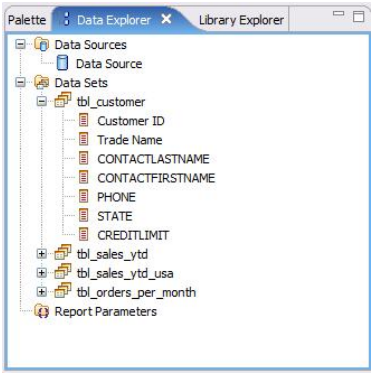


Abbildung 3. Data Explorer

Data Sets legen die auszulesenden Datensätze fest. Mit Hilfe des Data Set Editors lassen sich die Data Sets erstellen oder verändern. Auch die vom Benutzer zu bestimmenden Parameter lassen sich hier festlegen. Abhängig von der Datenquelle gibt es verschiedene Methoden um die Informationen der Datenquellen in die gewünschte Form zu bringen.

- Bei Datenbanken ist es meist möglich, einen Großteil der Datenbearbeitung auf dem Server mittels SQL-Queries zu bewältigen.
- Da bei Quellen wie Java Objects oder CSV Dateien meist kein Datenbanksystem die Datentransformationsaufgaben übernehmen kann, beherrscht BIRT die gängigen Operationen zur Datenstrukturierung: Summieren(Aggregieren), Berechnen des Durchschnitts, Gruppieren, Sortieren, Filtern, Bestimmen von Min Max, Berechnung von Prozentwerten etc. Für ausgefallene Berechnungen kann der Entwickler zudem auf eine programmierbare Ebene in Form eines *Expression Builders* für kleinere Ausdrücke sowie eines Quelltext Editors (*Script Ansicht*) für erweiterte Scripting-Funktionalität zurückgreifen, die ebenfalls bei der optischen Gestaltung eines Berichts (z.B. bedingte Formatierung) einsetzbar sind.

Data Sets sind wiederverwendbar, so dass mehrere Berichtsbausteine mit demselben Data Set verknüpft werden können. Die Unterscheidung zwischen der Verbindung zur Quelle und der Datenabfrage kommt der Übersichtlichkeit und Produktivität beim Erstellen von Berichten zugute.

Layout Editor

Für **Struktur und Gestaltung der Layoutkomponente** steht ein leistungsfähiger WYSIWYG (What You See Is What You Get) Editor zur Verfügung. Verschiedene Berichtselemente lassen sich per Drag&Drop Methode direkt im Berichtlayout verschieben oder aus einer Liste mit Standardbausteinen - beispielsweise eine Tabelle - hinzufügen. Hierzu bietet BIRT ein Werkzeugfenster mit einer Element-Palette. Auch die Data Sets aus dem Data Explorers

sind in das Layoutfenster verschiebbar, wobei die Datensätze standardmäßig in eine Tabelle eingebunden werden.

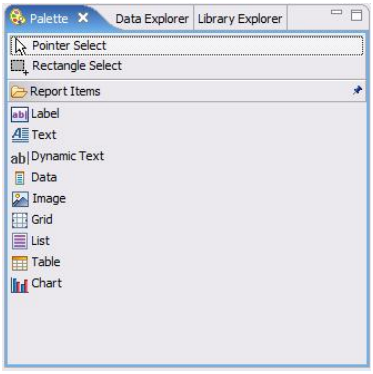


Abbildung 4. Palette mit Reportbausteinen

Der Layout Editor verfügt zudem über eine Reihe von nützlichen Spezialansichten, die den Entwickler beim Erstellen und Bearbeiten von Berichten unterstützen:

Layout Dies ist die Grundansicht zum Editieren eines Berichts. Elemente von der Palette oder dem Data Explorer lassen sich einfach in das Berichtdesign einfügen. Textfelder können durch Anklicken editiert werden.

Master Page Es handelt sich um eine spezielle Layoutansicht zum Bearbeiten der Master Page. Gerade bei größeren Berichten ist es überlegenswert eine Master Page mit einheitlichem Design zu erstellen, so dass man nicht jede Berichtsseite einzeln formatieren muss.

Script Der Code Editor bietet für JavaScript oder Java Programmierung die Standard-Werkzeuge der Eclipse Umgebung wie Syntax-Einfärbung, Auto-Vervollständigung. Durch Einsatz von Scripting-Funktionalität wird der Datenzugriff, die Berichterstellung und die Präsentation an die Geschäftslogik angepasst. Für kleinere Scripting-Tätigkeiten steht auch der Expression Builder zur Verfügung.

XML Source Eine Editoransicht des XML Quellcodes der RPTDesign Datei

Preview Vorschaufunktion mit der sich das Design mit Echtdaten begutachten lässt, ebenso werden Diagramme angezeigt. Diese Funktion ist sehr nützlich zur Überprüfung des aktuellen Berichtdesigns. Eventuelle Fehler im Design werden schon im Aufbauprozess erkennbar - auch Fehler in der Datenverarbeitung oder bei der Datenbankabfrage werden angezeigt.

Expression Builder

Ausdrücke, die mit dem Expression Builder erstellt werden, sind einfache Funktionen, die nur einen Wert zurückliefern. Diese Ausdrücke werden

in Verbindung mit Wertzuweisungen zu Berichtsbausteinen (z.B. im Property Editor), Einfügen von Bildern, dynamische Hyperlinks, für Standardparameter usw. verwendet.

Property Editor

Die am häufigsten genutzten **Eigenschaften von Berichtsbausteinen** lassen sich durch den Property Editior bearbeiten. Bei den meisten Berichtselementen trifft man neben der Layout- auch auf die Datenkomponente:

Properties zum Editieren von Sichtbarkeit, Schrift, Rahmen etc.
Binding legt fest, welcher Datensatz mit dem Element verknüpft ist.
Filters filtert den Datensatz auf beliebige Werte.
Sorting sortiert die Daten wie gewünscht.
Groups fassen Daten zu Gruppen zusammen.
Highlights heben abhängig vom Wert Daten speziell hervor.
Map ersetzt einen speziellen Datenwert durch einen anderen Ausdruck.

Durch die Verknüpfung mit Eclipse hat der Entwickler zudem Zugriff auf das Eclipse Fenster „Eigenschaften“, das eine noch detailliertere Beschreibung der einzelnen Berichtsbausteine bietet. Der Entwickler kann hier ebenfalls auf die programmierbare Ebene in Form des Expression Builders für einfache Ausdrücke sowie eines Code Editors für erweiterte Scripting-Funktionalität zurückgreifen, um Berichtsbausteine noch flexibler zu gestalten.

Library Explorer

Mit dem Library Explorer erhält der Benutzer Zugriff auf eine automatische generierte Bibliothek, in der bereits erstellte Berichtsbausteine wie z.B. Datenquellen, Tabellen oder JavaScripts verwaltet werden. Diese Objekte lassen sich wiederverwenden, indem sie einfach mittels Drag&Drop in den aktuellen Report verschoben werden.

Chart Builder

Mit dem Chart Builder lassen sich **Diagramme (Charts) in den Bericht einbinden** und deren Eigenschaften editieren. Zunächst wird über den Layout Editor ein Diagramm Element in dem Berichtlayout platziert. Darauf wird ein Wizard gestartet, der den Benutzer schrittweise durch die Designstufen des Diagramms führt. Der Wizard verfügt über ein Vorschaufenster, das dem Benutzer das endgültige Diagramm bereits in vereinfachter Fassung darstellt:

1. Zunächst wird der Diagrammtyp bestimmt. Unter anderem stehen folgende Typen zur Wahl, ein Teil davon auch in 3D:
   - Tortendiagramm
   - Funktionsgraph
   - Balkendiagramm

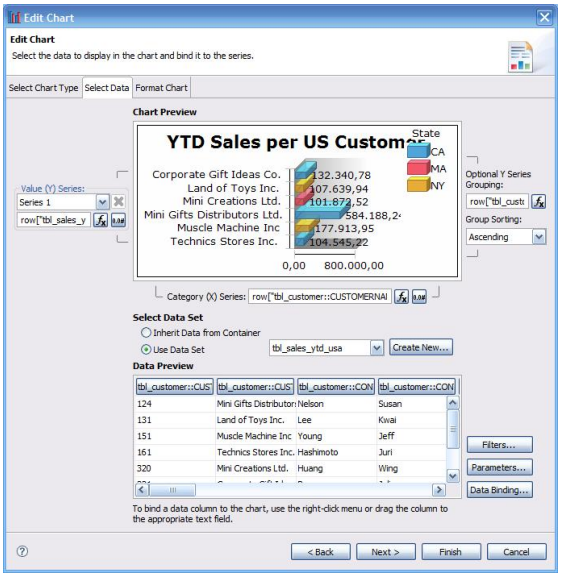


Abbildung 5. Chart Builder

   - Flächendiagramm
2. Dann wird die Datenstruktur abhängig vom Diagrammtyp eingerichtet:
   - Daten sind durch Data Sets aus dem Data Explorer verfügbar
   - Datensätze werden mit den gewünschten Achsen oder Flächen verknüpft
   - Legende kann zur farblichen Gruppierung genutzt werden
   - Parameter zur benutzerabhängigen Darstellung lassen sich festlegen
3. Zuletzt wird das Diagramm optisch angepasst:
   - Achsenbeschriftung, Textformatierung, Ausrichtung etc.
   - Sortierung, Balkenform, Graphenfarbe

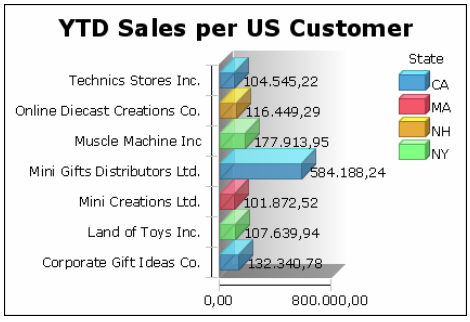


Abbildung 6. Einfaches Balkendiagramm erstellt mit BIRT

3.4 Laufzeitkomponenten

Die Vorschaufunktion der Designer Perspektive basiert bereits auf der BIRT Laufzeitumgebung, die das in XML formatierte RPTDesign interpretiert, daraus den Bericht generiert und im gewünschten Format ausgibt. Die Laufzeit setzt sich aus drei Komponenten zusammen:

Report Engine Die RE kann als das Herzstück der Laufzeit betrachtet werden. Die Engine besitzt zwei Grundfunktionen:

Bericht Generierung fasst folgende Operationen zusammen:

- Auswerten der für einen Report definierten Parametersets und Auslesen der Paramterwerte.
- Zusammenführen der im RPTDesign spezifizierten Datensätze(Data Sets) aus den entsprechenden Datenquellen(Data Sources)
- Ausführung von Datentransformationen und eingebundener Geschäftslogik.
- Bilddateien oder Diagramme in den Bericht einbinden

Bericht Ausgabe kann in unterschiedlichen Formaten erfolgen. Unterstützt werden:

- HTML Format / seitenweise HTML (paginated)
- PDF Format
- CSV Format

Abhängig vom Ausgabeformat und von den Vorgaben des Report Designs wird auch ein Inhaltsverzeichnis erstellt sowie der Bericht mit Lesezeichen und Markierungen versehen.

Beim Auftreten von Diagrammen greift die RE zusätzlich auf die Funktionen der Chart Engine zurück, die das Diagramm als Ergebnis zurückliefert. Die Report Engine bindet das Diagramm darauf in das endgültige Berichtsdokument ein.

Chart Engine Die CE leistet in der Laufzeit die Erstellung und Ausgabe von Diagrammen in verschiedenen Formaten. Die Chart Engine verfügt somit über die gleichen Grundfunktionen wie die RE. Die Generierung der Diagramme erfolgt ausgabeformatunabhängig über das EMF (Eclipse Modeling Framework). Hierbei werden ebenfalls die Vorgaben im RPTDesign ausgelesen. Die CE unterstützt verschiedene Bildformate: PNG, GIF, JPG, BMP und SVG für Vektorgrafiken. Für Java wird SWING, SWT unterstützt.

Report Viewer Zur Darstellung eines Berichtes innerhalb von Eclipse bietet BIRT den Report Viewer. Dieser nutzt die Funktionalität der Report- und Chart Engine zur Generierung der Berichte und präsentiert diese in einem Browserfenster. Der Report Viewer ist web-basiert und führt Berichte im HTML oder PDF Format aus. Zusätzlich zur bloßen Darstellung ermöglicht der Viewer durch Abfrage von Parametern, die zuvor im RPTDesign definiert wurden, die Erstellung von benutzerspezifischen Berichten. Darüber hinaus werden mittels Inhaltsverzeichnis, Navigationselementen und Lesezeichen praktische Bedienungselemente zum Betrachten von Berichten geboten, sodass sich ein Bericht leichter erschließen lässt. Dazu müssen allerdings die entsprechenden Anpassungen in der Designphase stattgefunden haben. Auch

lässt sich der Bericht mit dem Viewer in verschiedenen Formaten speichern und drucken.

3.5 Erweiterungs- und Integrationsmöglichkeiten

BIRT ist von Grund auf für die Integration in J2EE Web-Applikationen konzipiert. Zu diesem Zweck fusst das Plugin auf mehreren Application Programming Interfaces (APIs) sowie einem RPTDesign im transportablen XML Format. Die Erweiterbarkeit und Integrationsfähigkeit von BIRT wird insbesondere durch den Einsatz von APIs garantiert. Neben der **API Struktur** finden sie in Abbildung 7 noch einmal das Grundgerüst des Prozessfluss von BIRT (s.3.1) veranschaulicht:

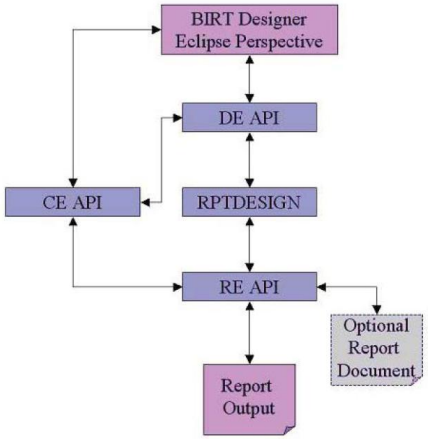


Abbildung 7. BIRT APIs, Quelle: Weathersby [7]

Die Engines lassen sich über diese APIs durch zusätzliche Funktionen erweitern und dadurch gut in bestehende Anwendungen integrieren:

DE API Neue Berichtsbausteine lassen sich dem Report-Object-Model (ROM) der *Design Engine* hinzufügen. Darüber hinaus lässt sich der Report Designer für die eigenen Bedürfnisse umgestalten und somit in die eigene Anwendung integrieren.

RE API Unterstützung zusätzlicher Dateiformate kann durch die Erweiterung mit entsprechenden Konvertern implementiert werden. Zudem können Berichte direkt über Java Code ausgeführt werden. Die *Report Engine* lässt sich damit in Java Anwendungen einbinden, oder ist als Servlet sowie in eine RCP Anwendungen integrierbar.

CE API Neue Diagrammtypen und Bildformate lassen sich der *Chart Engine* hinzufügen. Zudem kann die CE unabhängig von der RE in beliebige Java Anwendungen integriert werden.

Report Viewer Der Viewer kann als *eigenständige J2EE Web-Applikation* zum Ausführen und Darstellen von BIRT Berichten genutzt werden. Durch seine Erweiterbarkeit kann das Werkzeug mit zusätzlichen Funktionen ausgestattet werden und so als Ausgangspunkt für die eigene Applikation zum Darstellen der Berichte dienen. Zudem findet man im Viewer ein praktisches Beispiel für die Entwicklung eines Front-End für die Report Engine.

Die Integrationsfähigkeit in bestehende Anwendungen geht Hand in Hand mit der Erweiterbarkeit. In der Architektur von BIRT finden sich daher noch **zusätzliche Erweiterungs- bzw. Integrationspunkte**:

„programmierbare Ebene“ Wie bereits in der Beschreibung der Designer Perspektive erwähnt, können BIRT Berichte durch die Verwendung von Scripting-Funktionalität weitgehend unabhängig von den Beschränkungen durch die Design Engine gestaltet werden. So lässt sich Geschäftslogik überall im Bericht einbinden. Hauptsächlich wird das Scripting in JavaScript realisiert, worüber auch bestehende Java-Applikationen aufgerufen werden können. Zusätzlich verfügt BIRT über einen Java Event Handler, dessen Ereignisse (Events) in Java Code implementiert werden. Genaue Informationen zum Eventmodell findet man auf der BIRT Homepage [8].

Datenmodellunterstützung Neben der Unterstützung der bisher genannten Datenquellen wie JDBC Datenbanken gestattet das BIRT Open Data Access (ODA) Framework die Integration jeder Art von tabellarischen Daten durch die Einbindung entsprechender Treiber. Es ermöglicht auch den Aufbau einer neuen Bedienschnittstelle sowie von Laufzeitfunktionen für die zugehörigen Datenquellen. Wenn die Entwicklung entsprechender Treiber zu aufwendig erscheint, lassen sich Datenquellen auch einfach durch Scripting verfügbar machen (Scripted Data Sources).

Cascading Style Sheets BIRT unterstützt den Einsatz und den Import von Cascading Style Sheets (CSS) für den Report Design. Bekannt aus der Anwendung mit HTML Seiten lassen sich CSS ebenso effektiv in Verbindung mit BIRT Berichten benutzten und vereinfachen so das Erstellen eines einheitlichen Designs.

Internationalisierung BIRT verwendet Unicode und lässt sich gut an andere Sprachen und Kulturen anpassen. Allerdings gibt es keine Sprachunterstützung für rechts nach links gerichtete Sprachen.

Trotz des Fokus auf J2EE Applikationen lässt sich BIRT auch in andere Umgebungen integrieren. Ein Beispiel dafür liefert der Report Viewer, mit dem Berichte nur durch den Aufruf einer URL ausgeführt werden können. Auf diese Weise ist die Engine auch mit PHP, Perl oder sogar auf statischen Webseiten einsetzbar.

4 Schlussfolgerung

4.1 Vor- und Nachteile von BIRT

Gemessen an den zu Beginn des Artikels genannten Kriterien für ein Berichtswerkzeug finden wir alle wesentlichen Merkmale in BIRT vertreten:

- Hohe Flexibilität und Freiheit in der Berichtgestaltung wird durch einen sehr effizienten Report Designer garantiert. Auch **Benutzer ohne tiefer gehende Programmierkenntnisse können schnell und einfach Berichte erstellen**. Eine klare Trennung von Daten- und Layoutkomponenten ermöglicht es zudem, den Bericht bis ins Detail anzupassen. Die getrennte Definition von Datenquelle und Datensatz erlaubt gleichfalls einen flexiblen Umgang mit den Berichtsdaten. Datenstrukturen, unterschiedlich in Genauigkeit und Umfang, lassen sich beliebig miteinander kombinieren. Sind Berichtsbausteine einmal erstellt, lassen sich viele nach dem Baukastenprinzip wiederverwenden.
- Neben tabellarischer Darstellung sorgt die Chart Engine auch für die Möglichkeit der grafischen Aufbereitung von Daten.
- Möglichkeit der Integration in bestehende Systeme wird durch die API Struktur von BIRT und weitere Integrationspunkte gewährleistet. Geschäftslogik lässt sich überall im Bericht durch einfache Ausdrücke oder Code einfügen.
- Die zeitnahe Verfügbarkeit der Berichte ist bei einer Integration in eine J2EE Applikation gegeben. Berichte lassen sich über den Browser abrufen, und über Parameterwerte kann der Benutzer die Darstellung steuern.
- Die zeitnahe Aktualisierung von Berichtsinhalten wird durch die Kompatibilität von BIRT mit nahezu jedem Datenmodell garantiert. So kann BIRT die Daten für einen Bericht direkt aus einer Datenbank auslesen, transformieren und im Bericht darstellen.

Abgesehen von softwaretechnischen Merkmalen bietet BIRT durch sein Umfeld weitere Vorzüge:

- BIRT ist eine Open-Source-Lösung, wobei der Softwareanbieter Actuate als kommerzieller Service-Anbieter verfügbar ist.
- Das Projekt basiert auf Eclipse und ist voll in die IDE integriert.
- Mit der Eclipse Foundation hat BIRT einen starken Partner an seiner Seite.
- Das BIRT Projekt ist im Aufschwung und entwickelt sich rasch weiter.
- BIRT befindet sich mit anderen Eclipse Projekten im regen Austausch.

Die fehlende Unterstützung wichtiger Ausgabeformate wie XLS für Excel oder DOC/RTF für Word zählt zu den großen Nachteilen von BIRT. Im Arbeitsalltag sind diese Formate Standard und werden sehr häufig zum Informationsaustausch eingesetzt. Insbesondere können Dokumente in diesen Formaten leichter bearbeitet und eingebunden werden (z.B. zum Erstellen einer Präsentation), was für die Benutzer dieser Berichte von großer Bedeutung ist. Desweitern sind wie berichtet Integrationspunkte (siehe 3.5) für BIRT reichlich vorhanden;

allerdings fehlt es teilweise noch an der nötigen Dokumentation, was eine Integration von BIRT unnötig erschwert und viele Entwickler von einem Umstieg auf das „schwer integrierbare“ BIRT abschreckt, wie es in Diskussionen über den Einsatz von BIRT regelmäßig angeführt wird [9].

4.2 Fazit

Ein allgemeines Votum für oder gegen BIRT macht wenig Sinn. Die Entscheidung wird immer von den im Anwendungsfall ausschlaggebenden Faktoren abhängen. Dennoch sollte hervorgehoben werden, dass BIRT als Eclipse Top-Level-Projekt ein Berichtswerkzeug mit großem Zukunftspotenzial ist, das mehr und mehr seine Kinderkrankeiten hinter sich lässt. Ein Blick in den *Projektfahrplan* [10] für die kommenden Versionen zeigt, dass von dem BIRT Projekt noch einiges zu erwarten ist:

- Kreuztabellen (mehrdimensionale Datenauswertung) als neue Berichtsbausteine
- Unterstützung der Ausgabeformaten von Microsoft Word und Excel sowie XML und PostScript
- Unterstützung weiterer Diagrammtypen
- Ein grafischer Query Builder für erleichterte Erstellung von Datensätzen

Somit bleibt auch in Zeiten zunehmender Konkurrenz bei den Berichtswerkzeugen für Java (z.B.: JasperReport bietet mehrere Designer unabhängig von der Eclipse Plattform an, und mittlerweile hat sogar Crystal Reports, stark bei Microsofts Visual Studio, einen Designer für die Eclipse Plattform veröffentlicht) die Zukunft von BIRT ungefährdet.

Literatur

1. Strauch, B.: Business Intelligence *http://www.symposion.de/wm-ph/wm-ph_26.htm.*
2. Eclipse.org: Eclipse BIRT Homepage *http://www.eclipse.org/birt/.*
3. Eclipse.org: Eclipse Homepage *http://www.eclipse.org/.*
4. Eclipse.org: Eclipse Top-Level Projekte *http://www.eclipse.org/projects/.*
5. Hohmann, T.: Bereit zum Report. Eclipse-Magazin **4** (2005) 66–71
6. Actuate.com: Actuate BIRT Homepage *http://www.actuate.com/products/javareporting/birt-reporting.asp.*
7. Weathersby, J.: Introduction to Eclipse BIRT 2.1. Eclipse Magazine **1** (2006) 1–7
8. BIRT-Team: Event Handler Documentation *http://www.eclipse.org/birt/phoenix/deploy/reportScripting.php.*
9. Dewanto, L.: Das Muster Projekt. Eclipse-Magazin **9** (2006) 28–30
10. BIRT-Team: BIRT Project Plan *http://www.eclipse.org/birt/phoenix/project/project_plan_R2_2_0.php.*

Eclipse Rich Client Platform

Martin Krogmann

Universität Karlsruhe

Zusammenfassung Diese Ausarbeitung zum Proseminar *Software Entwicklung mit Eclipse* soll einen Überblick über die Eclipse Rich Client Platform schaffen. Die Entwicklung einer Rich Client Anwendung erfordert immer wieder die Entwicklung von Lösungen für wiederkehrende Grundprobleme. Die RCP versucht mit ihrer Struktur, viele gemeinsame Möglichkeiten und Eigenschaften solcher Anwendungen in einem Framework zu vereinheitlichen, um so die Entwicklung auf das Wesentliche zurückzukürzen. Die Charakteristiken, Bestandteile und deren Beziehung zueinander wird in dieser Ausarbeitung erläutert. Einen Einstieg in die tatsächliche Verwendung wird zudem angedeutet und anhand eines kurzen Beispielprojekts veranschaulicht. Dabei zeigt sich, dass die RCP ein vielversprechender Ansatz für verschiedenste Anwendungen ist.

Key words: Eclipse, RCP, Rich Client Platform

1 Einführung

1.1 Was ist die Rich Client Platform?

Mit dem Entwurf der Version 3 des als integrierte Entwicklungsumgebung (IDE) bekannt gewordenen Eclipse, sind viele der grundlegenden Komponenten und Hilfswerkzeuge, die für Eclipse entstanden sind, zu keinem Maße mehr spezifisch für IDEs zu sehen. Dies ist letztendlich eine Folge des Prinzips, dass Komponenten für eine möglichst flexible Anwendbarkeit nach Möglichkeit eine kleine, fest abgesteckte Aufgabe zu erfüllen haben. Denn klar ist: Eine Lösung für ein spezielles Problem kann nur durch Abstraktion von dafür unwichtigen Details auch eine Lösung für andere, ähnliche Probleme sein. Ausgehend von dieser Beobachtung der Vielseitigkeit und Allgemeinverwendbarkeit, fasste man deshalb die Kernkomponenten unter dem Begriff *Eclipse Rich Client Platform* zusammen.

Die Eclipse Rich Client Platform (im Weiteren: RCP) ist ein Framework, das die Grundlage für verschiedenste Java Applikationen bieten will. Diese Anwendungen müssen wenig mit der bekannten Eclipse IDE gemeinsam haben, können jedoch auf eine Fülle von Basisfunktionalitäten zurückgreifen. Um eine Anwendung beim Anwender bereitzustellen, bieten sich dem Entwickler verschiedene Optionen. Im Wesentlichen (es ließe sich auch feiner unterscheiden) sind dies Rich Clients und Thin Clients. *Rich Clients* sind komplette, selbstständige Anwendungen, die beim Benutzer bereitgestellt werden können. Sie müssen auf jedem Rechner lokal installiert sein, zusammen mit unter Umständen benötigten Bibliotheken und anderem (man denke an die Java Runtime Environment).

Eine Aktualisierung kann hier erneut einen ähnlichen Aufwand erfordern, wie die Erstinstallation, denn sie muss erneut auf allen Clients durchgeführt werden. Dagegen haben Rich Clients jedoch auch die Möglichkeit, die Ressourcen optimal zu nutzen. Damit sind reaktionsstarke und performante Anwendungen möglich, die auch dann funktionieren, wenn der Rechner nicht mit dem Netzwerk verbunden ist. Dies schließt natürlich nicht die Möglichkeit aus, dass solche Anwendungen auch Netzwerk-Ressourcen nutzen können. Im Gegensatz dazu stehen *Thin Clients*. Hier will man die meist schon bestehende Infrastruktur nutzen, um Anwendungen mit möglichst wenig Bereitstellungs- und Wartungsaufwand zur Verfügung zu stellen. Dies geschieht im Allgemeinen über eine Client-/Serverarchitektur, zum Beispiel über einen Browser. Damit muss für gewöhnlich nichts installiert werden, denn Browser sind auf allen modernen Rechnern bereits vorinstalliert. Dies erleichtert nicht nur die erste Inbetriebnahme, sondern ermöglicht auch ein zentrales Aktualisieren an nur einer Stelle (dem Server). Daher erfreuten sich Thin Clients zwischenzeitlich steigender Popularität, zumal ein Zustand erreicht ist, an dem nahezu alle aktuellen Browser Technologien wie AJAX (Asynchronous JavaScript and XML) unterstützen und damit recht aufwendige Anwendungen möglich sind. Browser sind hier nur eine Variante, denn Netzwerkdienste (oder *Web-Services*) lassen sich auf verschiedenste denkbare Weisen hierfür benutzen. Auch hardwareschwache Rechner sind Zielplattformen für Thin Clients. Großer Nachteil dieser Methode ist, dass diese Anwendungen auch nur dann gebraucht werden können, wenn ein Netzwerk ständig verfügbar ist. Die RCP zielt, wie der Name impliziert, auf den erstgenannten Ansatz hin. Sie verfolgt dabei einen stark Plugin-basierten Ansatz. Anstelle einer einzigen, monolithischen Basis, lassen sich einzelne Komponenten zur Anwendung hinzuziehen oder auch auslassen. Damit müssen in dem Paket, das dem Anwender zur Verfügung gestellt wird, auch nur die verwendeten Komponenten vorhanden sein. Insgesamt existiert ein umfangreiches Spektrum an Plugins, das dem geneigten Entwickler bei bestimmten Aufgaben eine Ausgangsbasis bietet. Mehr zum Aufbau der RCP im Kapitel 2 zur Struktur der RCP.

1.2 Warum RCP verwenden?

Bei der Planung einer neuen Anwendung stehen oft vielfältige Anforderungen auf der Liste. Dies kann Performanz, Feature-Reichtum, Plattformunabhängigkeit, Systemintegration, Erweiterbarkeit oder weitere denkbare Kriterien beinhalten. Viele dieser Möglichkeiten sind sehr allgemein, kosten jedoch viel Planungsbedarf, wenn eine Anwendung von Grund auf eine Eigenimplementierung sein soll. Womöglich müssen ganze Anwendungen umstrukturiert werden, da ein Anwendungsentwurf eine vorher nicht gesehene Anforderung nicht erfüllen kann. Der Wunsch nach einer bereits erprobten und aktiv entwickelten Plattform erscheint im Angesicht solcher Szenarien nur natürlich. An dieser Stelle seien auf der Seite von Eclipse prominente Unterstützer aus der Industrie – unter anderem IBM, Intel, HP oder SAP – genannt, um nur eine kleine Auswahl zu geben. Das Vorwort zu (1) handelt von Maestro, einer von der NASA auf RCP-Basis entwickelten Software. Dies verspricht dem interessierten Entwickler insgesamt, dass

nicht nur das aktuelle Produkt Eclipse RCP an der Wirklichkeit erprobt, sondern auch die Weiterentwicklung sicher gestellt ist. Die Eclipse RCP erspart dem Entwickler also einige, mit einer von Grund auf eigenen Entwicklung verbundene Schwierigkeiten und ermöglicht einen auf die wirklich gesuchte Problemlösung ausgerichteten Entwicklungsprozess. Dazu steht ihm mit der Eclipse IDE eine dazu prädestinierte Entwicklungsumgebung zur Verfügung, denn wie Kapitel 4 aufzeigen wird, sind hier viele Hilfestellungen angeboten. Welche Eigenschaften die Eclipse Rich Client Platform insgesamt charakterisiert, soll die Ausarbeitung im Folgenden nahebringen.

2 Struktur der RCP

2.1 RCP Architektur

Die RCP Architektur entstand vor allem mit wenigen zentralen Entwurfszielen im Sinn, nämlich Vielseitigkeit und vor allem Modularität. Das Ergebnis dieser Bestrebungen ist eine Platform, die zwar sehr konsistent, jedoch weniger ein großes, komplexes System als die Komposition vieler kleinerer Teile, mit spezifischen Funktionen, ist. Dieses Konzept bietet softwaretechnisch mehrere große Vorteile. Gegenüber dem zunächst größeren Planungsaufwand profitiert man davon, dass sich bei der Entwicklung der einzelnen Komponenten die schwierigen Zusammenhänge ausblenden lassen. Durch gut gewählte Schnittstellen bleibt (im Idealfall) die dazu notwendige Flexibilität erhalten. Die einzelne Komponente bleibt aber deutlich leichter wartbar und bei sich drastisch ändernden Bedürfnissen lässt sie sich sogar komplett austauschen. Im komplementären Fall lässt sie sich dann auch in einer völlig anderen Umgebung einsetzen. Für die Rich Client Platform ziehen sich diese Konzepte durch alle Ebenen. Die RCP besteht im Wesentlichen aus fünf aufeinander aufbauenden Hauptkomponenten (betrachte dazu auch Abb. 1).

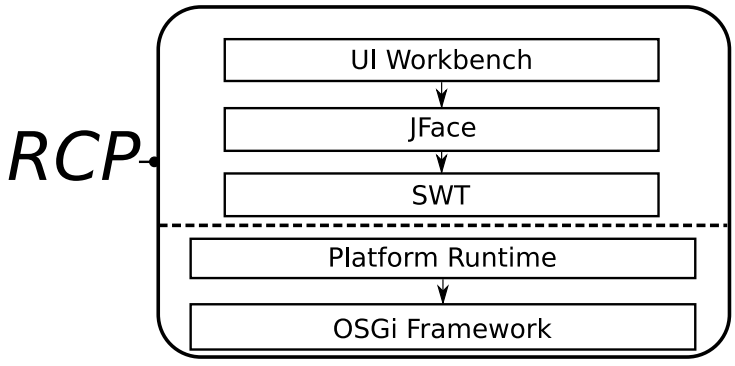


Abbildung 1. Struktur der RCP

Die Basis stellt die Runtime (*Equinox*, OSGi basierend) dar. Mit Eclipse Version 3.2 sind OSGi Implementierung und Runtime als voneinander getrennt anzusehen (In früheren Versionen war eine engere Verwebung vorhanden). Diese Betrachtungsweise soll auch im Folgenden fortgeführt werden. Das OSGi Framework ist eine Spezifikation für ein Komponentenmodell, das nicht Eclipse-spezifisch ist, im Gegensatz zur darauf bauenden Eclipse Runtime. Die in Eclipse verwendete OSGi-Implementierung trägt den Namen Equinox. Die Eclipse Runtime nutzt OSGi, um für Rich Client Anwendungen sinnvolle Erweiterungen zu ermöglichen. Weiterhin bilden SWT, JFace sowie die Generic Workbench (auch *UI Workbench*) nun (mit zunehmender Abstraktionsstufe) das, was dem Anwender als grafische Oberfläche präsentiert wird. SWT stellt *Widgets* – grundlegende Benutzeroberflächenelemente, die die Fenster-Manager der Betriebssysteme bereithalten, wie bspw. Buttons, Textfelder etc. – zur Verfügung. JFace erleichtert unter anderem die Verwendung sich daraus zusammensetzender Elemente wie Dialoge, indem es dem System vor allem das Model-View-Controller Entwurfsmuster hinzufügt. Die UI Workbench steckt schließlich einen Rahmen für die auftretenden Elemente und organisiert, wann etwas wo erscheint.

All diese Bestandteile stehen RCP Anwendungen standardmäßig zur Verfügung. Es ist jedoch auch möglich, die Runtime völlig ohne graphische Benutzeroberfläche zu verwenden. Dies liegt daran, dass sogar die fast immer grundlegende UI Workbench Plugin-Charakter hat. Ein Beispiel für eine reine Textanwendung liefert unter anderem (2). Genau betrachtet, hat man bei der Auswahl der Komponenten, die man zur Anwendungsentwicklung verwenden möchte, annähernd freie Hand, solange man die Abhängigkeiten berücksichtigt. Im weiteren sollen mit dem Begriff *RCP Applikation* – in der gleichen Weise wie (1) – genau die Anwendungen betrachtet werden, die diesen fünfteiligen Basissatz nutzen. Eine kurze Einführung in die genannten Module soll im Folgenden geboten werden.

OSGi Eclipse verwendet ein Komponentenmodell auf der Basis von OSGi. Das *Open Service Gateway Initiative* Framework, kurz OSGi (3), erlaubt es, verschiedene so genannte *Bundels* gleichzeitig in einer JVM (Java Virtuell Machine) zu betreiben. Die Bundles der OSGi-Spezifikation lassen sich für Eclipse gleichsetzen mit Plugins. Sie können zur Laufzeit ge- und entladen, (de-) installiert oder auch aktualisiert werden. Die Aufteilung in Plugins erlaubt es, immer nur die gerade benötigten Komponenten geladen zu halten – dies geschieht durch die *Platform Runtime*. Für Eclipse RCP ermöglicht dieses Komponentenmodell die einfache und flexible Zusammensetzung und Interaktion von Plugins. *Equinox* heißt das OSGi Framework, das mit Eclipse entstanden ist. Die für Eclipse 3.2 aktuelle Version von Equinox implementiert die *OSGi R4 core framework specification* der OSGi Alliance.

Platform Runtime Die *Platform Runtime* ist eine dünne Abstraktionsebene direkt über der Implemetierung des OSGi Frameworks (ursprünglich waren OSGi und Platform Runtime eine einzige Komponente). Durch sie werden einige

Eclipse-spezifische Dinge realisiert. Hierzu gehören Extensions und Extension Points (siehe 2.2), verwaltet in der *Extension Registry*, sowie auch die Definition von *Product* und *Application*. Eine *Application* beschreibt für die Runtime, wie eine Anwendung zu starten ist. Dies ist äquivalent zu sehen zu der Main-Methode in einer gewöhnlichen Java-Anwendung. Ein *Product* ist ein Sammelbegriff für verschiedene Anwendungseigenschaften. Dazu gehören Einstellungen und produktspezifische Anpassungen (*Branding*). Diese Anpassungen sind Anwendungssymbole, Logos und Ähnliches. Ein Veranschaulichung hierfür findet sich auch in Kapitel 4. Extensions legen bestimmte Relationen zwischen Plugins fest, beschrieben durch XML-Dateien (`plugin.xml`). Dies wird im Näheren unter 2.2 erläutert. Die Platform Runtime hat mit der Extension Registry einen Mechanismus, um mit Hilfe der XML-Dateien passende Relationen zu finden und so die Funktionalität zu verbinden. Dies stellt sich als ein mächtiges Integrationswerkzeug heraus. Die Platform Runtime ist auch zuständig für die Speicher- und (System-)Resourcenverwaltung für Fenster und Perspektiven (siehe 2.4). Sie instanziiert diese und gibt nicht mehr benötigte Resourcen frei.

SWT Das *Standard Widget Toolkit* (4) ist eine Graphikbibliothek für die Erzeugung von Benutzeroberflächen. Zu den unterstützten Widgets zählen z.B. Listen, Buttons, Textfelder oder Werkzeugleisten. Das SWT ist ähnlich zu den für Java bekannten Bibliotheken AWT und Swing. Es entstand vor allem mit der Intention, Schwächen dieser anderen beiden Bibliotheken zu vermeiden. So greift SWT, wenn immer möglich, auf die nativen Bedienelemente zurück, um so eine möglichst hohe Plattformintegration zu erreichen. Dazu gehört auch plattformspezifisches Verhalten wie etwa Drag und Drop sowie die Verwendung der Zwischenablage. Für nicht auf der Plattform verfügbare Elemente findet, soweit anwendbar, eine transparente Emulation statt, so dass die verwendete API unabhängig davon die gleiche bleibt. Einen genaueren Einblick bietet hier (5). Zudem zeichnet sich SWT durch die vergleichsweise hohe Zeichengeschwindigkeit aus.

JFace *JFace* bildet eine Werkzeugkiste zur Bewältigung üblicher Aufgaben für die Benutzeroberfläche und ist eine auf SWT aufsetzende Abstraktion, die im Wesentlichen das Model-View-Controller Entwurfsmuster für SWT bereitstellt. Dies bedeutet eine Trennung von Daten (Model), Präsentation (View) und Programmsteuerung (Controller) sowie die wechselseitige Information über Änderungen durch Ereignisse. Im Kern wird dieses Muster durch sogenannte Viewer implementiert. Diese werden z.B. für die Abbildung von Eigenschaften verschiedener Bedienelemente zwecks Synchronisation verwendet. Hierfür lassen sich auch Filter-Mechanismen definieren. Viewer ermöglichen auch Benachrichtigungen über Ereignisse, die in Widgets stattfinden, d.h. Selektionen, Änderungen bestimmter Eigenschaften etc. JFace enthält bereits einige Standard Viewer. Eine Schriftartenverwaltung und Klassen für Eigenschaftendialoge oder Vorlagen für Wizards sind ebenfalls in JFace enthalten. Von den JFace-Klassen bekommen insbesondere auch Actions durch die UI Workbench eine große Relevanz.

Actions werden näher beschrieben im Kapitel 2.4. JFace ist auch in seiner Implementierung nicht mehr plattformspezifisch, d.h. auf allen Plattformen läuft der selbe Code.

UI Workbench Die UI Workbench setzt wiederum auf SWT und JFace auf. Sie stellt die organisierende Struktur dar und definiert die wichtigen Konzepte *Editoren*, *Perspektiven*, *Views* sowie *Action Sets*, d.h. vereinfacht, dass die UI Workbench entscheidet, wo welche Elemente der Benutzoberfläche letztlich dargestellt werden. Sie ist für die Rich Client Platform von vergleichsweise zentraler Bedeutung. Daher findet sich eine detailliertere Beschreibung auch im Abschnitt 2.4.

2.2 Plugins

Die Nutzung von Plugins ist wohl der prägendste Baustein für den Aufbau von RCP Anwendungen. Ein Plugin kapselt jeweils eine (meist kleine) Einheit von Funktionalität und lässt sich einzeln von der Anwendung bzw. der Runtime laden. Plugins können untereinander interagieren, wenn entsprechende Extension Points definiert sind. Größere Projekte können viele Plugins einsetzen, um über deren kombinierte Funktionsvielfalt zu verfügen.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: RCP_TODO Plugin
Bundle-SymbolicName: RCP_TODO; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: rcp_todo.Activator
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
Eclipse-LazyStart: true
```

Abbildung 2. Eine sehr einfache Manifest-Datei aus der Beispielanwendung

Plugins werden für die Runtime durch eine *Manifest*-Datei (`MANIFEST.MF`, für ein Beispiel siehe Abb. 2) definiert. Sie beschreibt Eigenschaften wie Name, Version, aber insbesondere auch Abhängigkeiten des Plugins und Dienste, die das Plugin bereitstellt. Ein Plugin enthält zusätzlich die Datei `plugin.xml`, die zur Deklaration von Extensions verwendet wird. Ein Beispiel hierfür ist in Abb. 2.2 zu sehen. Diese beiden Dateien ermöglichen der Runtime erst, Plugins nur bei Bedarf zu laden. Dies ermöglicht das Szenario, das Plugins auch dann nicht durch die Runtime geladen werden müssen, wenn das Plugin bspw. Beschriftungen und Icons beiträgt. Die Metainformation sind hinreichend, damit die Workbench entsprechende Werkzeugleisten und Menüs erzeugen kann.

Außerdem in Plugins enthalten sein kann Javacode in einer JAR-Datei und verschiedenste Ressourcen (auch native Bibliotheken), die das Plugin verwendet oder beiträgt – dies ist jedoch optional, denn ein Plugin kann bspw. auch nur aus HTML-Dokumentation bestehen.

Extensions und Extension Points Eclipse ermöglicht es Plugins, Extension Points zu definieren, an denen Erweiterbarkeit durch Extensions, also Erweiterungen, vorgesehen ist. Anderen Plugins steht es dadurch offen, sich so zu integrieren und an diesen fest definierten Stellen Funktionen in Form von Extensions hinzuzufügen. Dies können auch Actions sein, die sich in Menüs oder Werkzeugleisten einfügen (siehe auch Kapitel 2.4).

```
<extension
point="org.eclipse.ui.views">
    <view
name="Mailboxes"
allowMultiple="true"
icon="icons/sample3.gif"
class="rcp_todo.NavigationView"
id="RCP_TODO.navigationView">
    </view>
</extension>
```

Abbildung 3. Beispiel für eine View-Extension in der Datei plugin.xml

2.3 Nutzbare Plugin Umgebung

Die RCP Anwendungen profitieren oftmals in vielen Bereichen von der Nutzung der Plugins, die RCP Entwicklern zur Verfügung stehen. Diese können zum Beispiel der erweitererten Eclipse Plattform entstammen, die z.B. durch die Entwicklungsumgebung, dem Eclipse SDK, in Gebrauch sind, oder auch speziell für RCP entwickelt sein. Inzwischen haben sich auch kommerzielle Anbieter für Plugins in der Eclipse-Umgebung gebildet. Einige sehr verbreitete sollen hier kurz beleuchtet werden.

Das Hilfesystem Das Hilfesystem bietet ein einheitliches System zum Ablegen und Wiederfinden von Dokumentation. Dazu lassen sich verschiedene, seperate Dokumentationstypen verwenden, zum Beispiel Benutzer- und Entwicklerdokumentation. Die eigentliche Dokumentation wird in Form von HTML-Dateien erwartet, zusätzlich dazu sollen XML-Dateien verwendet werden um eine Navigationsstruktur herzustellen. So lässt sich auch bereits bestehende HTML-Dokumentation in dieses System einfügen. Hilfethemen werden mit Hilfe dieser Informationen in einer Baumstruktur angeordnet. Diese kann erneut offen für

Erweiterungen an festgelegten Stellen sein, was als logische Konsequenz zu sehen ist, denn schließlich können auch die zu dokumentierenden Plugins erweitert werden.

Team Plugin Das Team Plugin stellt eine Infrastruktur für kollaboratives Bearbeiten von Ressourcen zur Verfügung. Dazu gehören Sperrstrategien und die Integration von CVS- bzw. SVN-Servern. Durch Extensions lassen sich hier auch beliebige andere Dienste als so genannte *Service Provider* integrieren. Da diese sehr unterschiedlich geartet sein können, ist die Art und Weise, wie sich diese integrieren, bewusst offen gehalten. Die Service-Provider haben die Möglichkeit, ihre Erweiterungen an einer Reihe von typischen Operationen und Manipulationen auf den Ressourcen zu definieren. Gleiches gilt für entsprechende Bedienelemente. Zwar werden bestimmte Orte für die Erweiterungen vorgegeben – wie die Erweiterung jedoch aussieht liegt beim Service-Provider.

Aktualisierungsmechanismen Eclipse bringt einen ausgereiften Aktualisierungsmechanismus mit, mit dem einzelne Plugins installiert und aktualisiert werden können. Dieser Mechanismus lässt sich auch für die eigenen Komponenten nutzen. Das System kann die Verfügbarkeit von Aktualisierungen online oder lokal bestimmen und die betroffenen Plugins selbstständig in die RCP Anwendung einfügen. Hiermit wird dem RCP-Entwickler ein Mittel an die Hand gegeben, mit dem er den Wartungsaufwand, wie er auch in der Einleitung für Rich Clients erwähnt wurde, geschickt reduzieren kann.

Weitere Viele weitere Plugins stehen zur Verfügung. Diese sollen nur knapp erwähnt sein. Von den frei verfügbaren listet (6) die aus Eclipse stammenden auf, die jedoch nicht das Ende des Spektrums darstellen. Dies sind z.B.:

- Draw2D – Ein Plugin mit Funktionen zum zeichnen von Vektorgraphik
- Forms – Auf SWT bauendes Plugin, das HTML-ähnliche Bedienelemente bereithält, die sich in allen UI Elementen verwenden lassen
- Welcome Page/Intro – Plugin mit dem Zweck, neue Benutzer in eine Anwendung einzuführen
- Cheat Sheets – Zielt darauf, Leitfäden für die Benutzung einer Anwendung zu erstellen.
- Graphical Editing Framework (GEF) – Erlaubt die Erzeugung graphischer Editoren aus gegebenen Anwendungsmodellen
- Eclipse Modeling Framework – Ein Plugin zur Erstellung von Modellen und der Generierung von Quellcode aus solchen Modellen.
- Resources – Ein Modell zur Verwaltung von Dateien, insbesondere Projekten, und Ordnern.
- Text – Plugin für Textbearbeitung mit vielen Funktionen

Kommerzielle Eclipse Plugins Neben den frei verfügbaren Plugins hat sich auch eine kommerzielle Umgebung für Eclipse gebildet. Verschiedene Hersteller versuchen mit ihren Plugins, möglichst attraktive Erleichterungen für unterschiedlichste RCP Anwendungsentwicklungen anzubieten. In (7) erwähnt der Autor hier stellvertretend das Plugin RCP Developer, dass z.B. einen Designer und ein Unit-Testing-Framework sowie andere Vereinfachungen für bereits bestehende Plugins anbietet (Internetpräsenz unter (8)).

2.4 Konzepte der RCP

Die grundlegenden Konzepte der RCP werden praxisnah in (9), oder auch konzeptioneller in (1) sowie vielen anderen Dokumenten dargestellt. Einen umfassenden Überblick aus den verschiedenen Quellen soll im folgenden geboten werden.

Die Workbench Die *Workbench* nutzt SWT und JFace, um eine kohärente Bedienung von RCP Anwendungen zu ermöglichen. Der Schritt, die Benutzeroberfläche in wiederverwendbare und austauschbare Fragmente zu teilen, ist beim Aufbau aller Anwendungen, die man als *Rich Clients* bezeichnet, an einer Stelle notwendig. Die Einführung dieser Abstraktionsebene, die durchdachte, allgemein wiederverwendbare Konzepte zur Verfügung stellt, ist daher eine für die Rich Client Platform zentrale Aufgabe. Die wesentlichen Bestandteile sind unter den folgenden Punkten erklärt. Die Workbench ist die zusammenfassende Struktur für die gesamte Benutzeroberfläche. Als solche enthält sie die verschiedenen Views, Editoren und Menüelemente und wacht über die unterschiedlichen Perspektiven. Dazu stellt sie auch Services bereit, die Ereignisse für gerade aktive Views und Editoren oder neue Auswahlen auslösen. Dies kann von Plugins genutzt werden – sie können sich als *Listener* registrieren – um entsprechend darauf zu reagieren. Die Workbench bietet Extension Points an, mit denen sich neue Editoren, Views und Perspektiven durch Plugins beitragen lassen.

Perspektiven *Perspektiven* erlauben es, je nach zu erledigender Aufgabe, die *Views* und *Editoren* zu organisieren sowie festzulegen, welche *Actions* zur Verfügung stehen. Dies kann sehr sinnvoll sein, um nicht relevante Bedienelemente zu verstecken und die Übersicht zu waren. Perspektiven bilden somit den Grundbaustein für eine problemorientierte Oberfläche. In der Workbench können mehrere Perpektiven definiert sein, wovon aber jeweils nur eine angezeigt wird. Wenn die Anwendung dies zulässt lassen sie sich durch den Benutzer auch individualisieren, um eine für ihn produktive Oberfläche zu erhalten. Für Perspektiven sind Extension Points definiert, mit denen sich neue Kurzbefehle (Shortcut Keys), Views und Action Sets hinzufügen lassen. Eine RCP Applikation muss mindestens eine Perspektive definieren.

Editoren Editoren zeichnen sich durch die Möglichkeit zum Öffnen, Speichern und Schließen von bestimmten Objekten aus. Das Speichern erfolgt hier für

gewöhnlich nicht sofort wärend des Editierens, sodass sich Änderungen erst bei explizitem Speichern auswirken. Editoren erlauben aber auch weitere Workbench-Integration. So können sie spezielle Aktionen zu den Menüs und Werkzeugleisten beitragen, was dann sinnvoll ist, wenn diese Aktionen genau auf die Manipulation des gerade zu bearbeitenden Objekts abzielen. Die RCP stellt einen Standardeditor für Textdateien bereit. Speziellere Editoren lassen sich dann durch verschiedene Plugins beitragen. Editoren lassen sich auch ihrerseits durch Extensions erweitern. Dafür sind Extension Points für Actions gegeben, so dass sich bspw. das Kontextmenü erweitern lässt.

Views *Views* eignen sich besonders zur Darstellung von kontextsensitiven Informationen. So können sie Eigenschaften auflisten und bearbeitbar machen oder auch Informationen über Objekte anzeigen. Dabei können sie sich auf verwendete Editoren, aber auch auf andere Views beziehen. Änderungen, die in Views vorgenommen werden, wirken sich unmittelbar aus, d.h. sie übertragen sich direkt auf andere relevante Views oder auch Editoren. Hier ist kein eigener Speichermechanismus vorgesehen, wie er für Editoren existiert. Views bieten Extension Points, um Actions in die lokale Werkzeugleiste oder in das Kontextmenü einzufügen.

Actions *Actions* sind Befehle, die per Entwurf nicht an eine bestimmte Stelle in der Benutzeroberfläche gebunden sind. Statt dessen stellt jede festgelegte Action alle Eigenschaften bereit, um an verschiedensten Stellen eingebunden zu werden, sei dies in Menüs, als Icon in der Werkzeugleiste oder als Button. Dies ist kann immer auch dort sein, wo durch andere Plugins speziell hierfür Extension Points anboten werden. Eigenschaften die zur Definition einer Action festgelegt werden müssen sind zum Beispiel Titeltext (Label), Symbol (Icon) und Tooltips.

3 RCP aus Entwicklersicht

Für den Anwendungsentwickler sind vor allem drei zentrale Advisor-Klassen von Bedeutung. Dies sind abstrakte Klassen die das grundlegende Verhalten der Anwendung vorgeben. Der Entwickler muss zwingend einige wenige Funktionen implementieren und hat zusätzlich die Möglichkeit, nach Wunsch weitere Funktionen zu überschreiben, um so seine Anwendung zu charakterisieren. Die Advisor Klassen sind im einzelnen:

- Workbench Advisor
- Workbench Window Advisor
- ActionBar Advisor

Die RCP kapselt mit den Advisorn alle Optionen zur Einstellung der Workbench und hierarchisch niedriger liegender Elemente, d.h. den einzelnen Fenstern und deren Benutzerschnittstellen, *Views* und *Perspektiven*. Zusätzlich kann noch ein Plugin-Singleton verwendet werden, um in der Anwendung übergreifende Funktionen zur Verfügung zu stellen.

3.1 Workbench Advisor

Der Workbench Advisor stellt Möglichkeiten bereit, den übergeordneten Status der Workbench zu überwachen, Eigenschaften der Workbench wie die Standard-Perspektive festzulegen und auf Ereignisse zu reagieren. Dies sind unter anderem Initialisierung-, Start- und Beendigungsereignisse, aber auch Fehlerbehandlungen und Leerlaufverwertungen. WorkbenchAdvisor ist eine abstrakte Klasse, die durch den Entwickler implementiert werden muss. Dabei sind aber mit Ausnahme der Methoden `createWorkbenchWindowAdvisor()` und `getInitialWindowPerspectiveId()` die Implementierungen optional. Dabei reicht `createWorkbenchWindowAdvisor()` auch einen Configurer an den WorkbenchWindowAdvisor weiter. Siehe hierzu auch Abschnitt 3.2. Die weiteren Methoden werden aufgerufen, bieten aber Standardimplementierungen, oft mit wenig signifikanter Funktionalität. Ein Teil dieser Methoden beziehen sich auf den Lebenszyklus der Workbench. Dies sind die Methoden `initialize()`, `preStartup()`, `postStartup()`, `preShutdown()`, `postShutdown()`. Hier kann man beliebige Funktionalität an den entsprechenden Zuständen zwischenschalten, um Vorbereitungs- oder Aufräumarbeiten vorzunehmen. Genauere Informationen (auch für die folgenden Advisor-Klassen) erhält man aus der API-Dokumentation und der Hilfe (10). Weiterhin gibts es zwei Methoden die sich in Loops einschalten, die zum einen bei Leerlauf der Anwendung (`eventLoopIdle()`), zum anderen bei Fehlerbehandlung (`eventLoopException()`) auftreten. Methoden, die neben der Startperspektive (s.o.) zusätzliche anzeigebezogene Anwendungseigenschaften abfragen, lassen sich ebenfalls überschreiben. Dies sind `getDefaultPageInput()` und `getMainPreferencePageId()`. Für eine genauere Kontrolle über die Erzeugung der Fenster steht die Methode `openWindows()` bereit. Standardmäßig wird der zuletzt vorhandene Workbench-Status wiederhergestellt.

3.2 Workbench Window Advisor

Mit Hilfe des Workbench Window Advisors lassen sich unter anderem Eigenschaften der Benutzeroberfläche anpassen und das Verhalten über den Lebenszyklus einzelner Fenster hinweg. Hierfür erwartet der Advisor an allen Stellen einen `Configurer`. Über diese Klasse lassen sich dann die Fenstereigenschaften festlegen. Eigenschaften sind zum Beispiel die Größe, der Titel, ob Coolbars oder Statusleisten angezeigt werden sollen, etc. Ähnlich wie für den WorkbenchAdvisor gibt es auch für den WorkbenchWindowAdvisor einige Möglichkeiten eigene Methodenimplementierungen zu festen Zeitpunkten aufrufen zu lassen. Dazu überschreibt man die Methoden `preWindowopen()`, `postWindowRestore()`, `postWindowCreate()`, `openIntro()`, `postWindowOpen()`, `preWindowShellClose()` oder `postWindowClose()`. Weitere Kontrollmöglichkeiten über das Layout bieten `createWindowContents()` und `createEmptyWindowContents()`.

3.3 ActionBar Advisor

Der ActionBar Advisor verwaltet Aktionen, die auf "ActionBars" – dies meint Elemente wie Statusleisten, Menüs oder Werkzeugleisten – definiert werden kön-

nen. Neben der explizit programmatischen Definition ist dies im Übrigen auch wieder durch Extensions und die `plugin.xml`-Datei möglich. Unter Eclipse ist auch der Begriff Coolbar geläufig. Gemeint sind hiermit Werkzeugleisten, die aus mehreren einzelnen Werkzeugleisten zusammengesetzt sind, die sich darin anordnen lassen. Der Advisor erzeugt die Actionbars, fügt sie zu gegebenem Zeitpunkt hinzu und entfernt sie (was ermöglicht, den Speicher wieder freizugeben), falls sie nicht mehr benötigt werden. Um nun Actions anzulegen, erledigt man dies durch das Überschreiben der Methoden `makeActions()`, `fillMenuBar()`, `fillCoolBar()` und `fillStatusLine()` und das Ablegen der dort ersichtlich zugehörigen Elemente. Außerdem vorhanden ist die OLE-spezifische Methode `isApplicationMenu()`, die auf Windowssystemen angibt, ob das Menü einer mit Mircosofts OLE eingebundenen Fremdanwendung entstammt. Erneut sei hier auf die API hingewiesen.

3.4 Plugin Klasse

Die Pluginklasse ist ein optionale Singleton-Klasse, in der man beispielsweise Werkzeuge für die gesamte Anwendung zur Verfügung stellen kann. Dies ist ein sinnvoller Ort, um die Funktionalität, die man mit seiner Anwendung entwickelt, für die verschiedenen Klassen auf einheitliche Art und Weise bereit zu stellen. Außer den zum Überschreiben vorgesehenen Funktionen `start()`, `startUp()` und `stop()`, hat der Entwickler relativ freie Hand, ob und wie er diese Klasse nutzt. Üblich – wie in der unter Kapitel 4 vorgestellten Beispielanwendung – ist auch die Nutzung der `AbstractUIPlugin`-Klasse. Die `AbstractUIPlugin`-Klasse erbt von Plugin und erweitert diese unter anderem um eine zentrale Ablage für Bilddaten, die `ImageRegistry`, und um zugehörige Methoden.

4 Beispiel einer Eclipse RCP Anwendung

Eclipse stellt weitreichende, von Version zu Version erweiterte Starthilfen für das Erstellen von RCP Plugin Projekten zur Verfügung. Dadurch lassen sich lauffähige Anwendungsgerüste sehr schnell erstellen, wie auch (9), (11) bis (12) zeigt. Neben einem minimalistischem Beispiel existiert z.B. auch ein Mailclient in Rohform – natürlich ohne wirkliche Mailfähigkeiten (siehe z.B. (12)). Außerdem bietet die Eclipse IDE für die Konfigurationsdateien eine komfortable Oberfläche, mit der man schnell gewünschte Änderungen vornehmen kann.

Im Rahmen des Prosemininars wurde eine knappe, aber lauffähige Anwendung zur Demonstration von RCP Fähigkeiten entwickelt. Selbstgestecktes Ziel hierfür war eine einfache Todo-Listen Verwaltung. Natürlich mag die Verwendung von RCP an dieser Stelle übertrieben scheinen. Andererseits lässt sich eine solche Anwendung schnell in eine viel umfassendere Umgebung einfassen – vor allem da die Funktionalität letztlich ohnehin als Plugin realisiert wird und sich somit, vorrausgesetzt es ist sauber definiert, von anderen RCP-Anwendungen nutzen lässt. So wäre es denkbar das Todo Beispielprogramm in einen vollwertigen Personal Information Manager (PIM) einzugliedern, oder das Teamplugin

zu nutzen, um hieraus eine kollaborative Aufgabenverteilung zu bilden. Letztlich sind der Vorstellung hier also nur wenige Grenzen gesetzt.

Ausgangspunkt für das Beispielprojekt war die oben erwähnte Mailclient-Vorlage. Hier sind bereits alle notwendigen Dinge definiert. Das heisst im einzelnen:

- Der Klassenname für den Haupteinstiegspunkt `org.eclipse.core.runtime.applications` ist durch eine Extension festgelegt (durch Verweis auf die `run`-Methode der Application Klasse, die mittels der Runtime die Workbench erzeugt).
- Eine Perspektive ist festgelegt. Die Schnittstelle `IPerspectiveFactory` verlangt hierfür mindestens die Implementierung von `createInitialLayout()`.
- Der WorkbenchAdvisor enthält Implementierungen für `createWorkbenchWindowAdvisor()` und `getInitialWindowPerspectiveId()`.
- Der WorkbenchWindowAdvisor enthält ebenso die Standard-Implementierungen sowie leichte Anpassungen durch `preWindowOpen()`.
- Eine Plugin-Klasse, d.h. hier eine AbstractUIPlugin-Klasse ist implementiert als die Klasse `Activator`.

Insgesamt wird schnell klar, dass die vorhandenen Mechanismen für vieles bereits weithin hinreichend Funktion bieten, sodass sich die Entwicklung der Anwendung zum Großteil wirklich auf das Versprochene hinausläuft - nämlich der Festlegung gewünschten Verhaltens und dem Anbieten eigener Services.

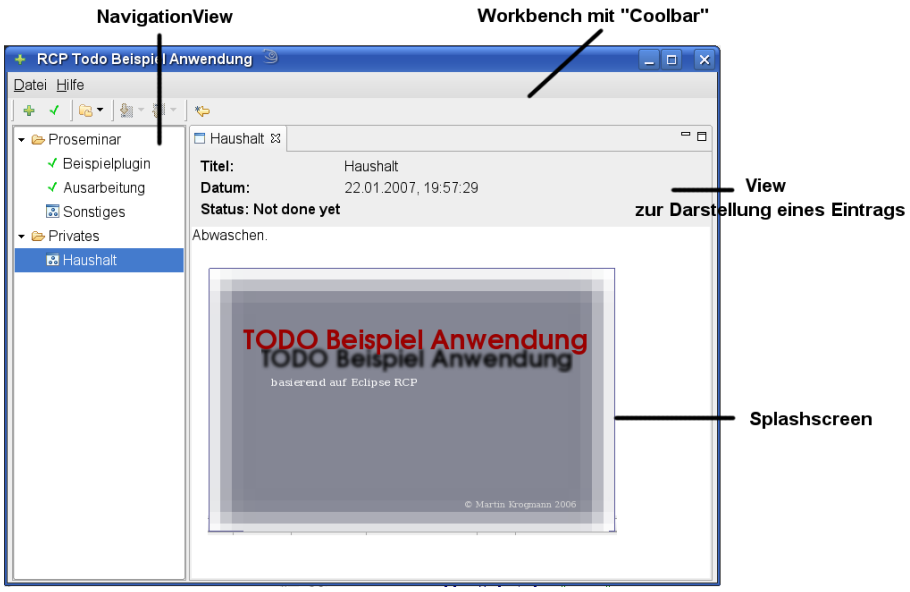


Abbildung 4. Screenshot der erstellten Anwendung

Die Abbildung 4 zeigt die entstandene Anwendung. Die Workbench enthält hier einerseits eine Coolbar mit darin befindlichen Actions – teilweise selbstdefiniert (Button 1 und 2 von links) und teilweise durch ein testweise hinzugezogenes Editor-Plugin, das Teil des Eclipse SDKs ist. Andererseits enthält die Workbench noch die zwei definierten Views *NavigationView* und *View*. Mit ersterem lassen sich verschiedene Einträge in der Baumansicht auswählen und in zweiterem werden diese dann angezeigt. Realisiert wird dies dadurch, dass für den NavigationView in der Methode `createPartControl()` der Aufruf `getSite().SetSelectionProvider()` hinzuzufügen ist, wobei als Parameter die verwendete TreeViewer Klasse verwendet wird. Im Hauptview kann dann ein ISelectionListener implementiert werden, d.h. insbesondere die Methode `void selectionChanged( IWorkbenchPart part, ISelection selection)`, in der man dann auf Änderung der Auswahl reagieren kann. Dazu trägt man diese Implementierung durch den Aufruf von `getViewSite().getPage().addSelectionListener(this);` als Listener ein, etwa in der eigenen `createPartControl()` Methode. Der NavigationView selbst entsteht durch eine Viewer-Klasse, nämlich die Klasse `TreeViewer`. Dieser definiert man einen `IContentProvider` – für die Repräsentation der Baumstruktur – und einen `ILabelProvider` für die Beschriftung der einzelnen Knoten. Für die ganze `createPartControl()`-Methode betrachte man Abb. 4. Der View zur Anzeige eines Eintrags definiert sich hauptsächlich durch gewöhnliche SWT-Elemente und sei hier ausgelassen. Insgesamt liegt es an dem Entwickler, seine eigene Programmlogik einzubinden, um zu einer sinnvollen RCP-Anwendung zu gelangen.

Erstellt man eine Produkt-Konfiguration (.product), so sieht Eclipse auch die Option vor, dem erstellten Produkt auf einfache Art und Weise ein Profil zu geben (*Branding*). Das bedeutet man kann einen Splash-Screen verwenden, der die Anwendung beim Laden ankündigt und optional auch Ladetexte und Fortschrittsbalken anzeigt. Der Splash-Screen lässt sich auch direkt in einem Plugin über vorgegebene Schnittstellen erweitern (siehe hierzu auch die Abb 4). Außerdem kann man den "About"-Dialog personalisieren, um dabei z.B. ein eigenes Impressum und Logo anzugeben. Mit der Produkt-Konfiguration können auch plattformspezifische Startargumente (auch für die JVM) und Dateisymbole festgelegt werden.

Ein Nachteil, den die eigenen Tests an den Tag legten, ist, das die von Eclipse bereitsgestellten Wizards durchaus differenziert zu betrachten sind. Neben dem offensichtlichen schnellen Einstieg in die Erweiterung der Anwendung, kann auch ein gegenläufiges Resultat eintreten. Nämlich das eine vorher lauffähige Anwendung ihren Dienst verweigert. Wizards ersparen dem Entwickler demnach im Allgemeinen nicht, sich mit dem darunterliegenden System auseinanderzusetzen und verstecken auf der anderen Seite durchaus auch Flexibilität – wobei sie natürlich immer optional betrachtet werden müssen.

```
/**
* This is a callback that will allow us to create the viewer and
* initialize it.
*/
public void createPartControl(Composite parent) {
    viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL
                                    | SWT.V_SCROLL | SWT.BORDER);
    ViewContentProvider vcProvider = new ViewContentProvider();
    viewer.setContentProvider(vcProvider);
    ViewLabelProvider vlProvider = new ViewLabelProvider();
    viewer.setLabelProvider(vlProvider);
    viewer.setInput(createTreeModel()); // createTreeModel erstellt
                                        // den Baum
    viewer.expandAll();
    viewer.setSelection(new StructuredSelection(
                  viewer.getTree().getItem(0).getItem(0).getData()),
                                                              true);
    getSite().setSelectionProvider(viewer); // propagate selection
                                            // changing events
    Activator.getDefault().getTodoItemManager().
                        addPropertyChangeListener(new TreeUpdater());
}
```

Abbildung 5. Die createPartControl() Methode des NavigationView

5 Fazit

Für Anwendungsentwickler hat die Verwendung von RCP einige nicht zu unterschätzende Vorteile. Er kann zunächst einmal von der Betriebssystemunabhängigkeit profitieren. Bei der Auswahl der Entwicklungsplattform erscheint hier Java als naheliegende Wahl, da hier ausgereifte Bibliotheken zur Verfügung stehen und bereits eine relativ hohe Verbreitung der Java Runtime Environment (JRE) vorhanden ist. Mit der Verfügbarkeit der RCP wird diese Wahl nocheinmal attraktiver.

Mit RCP bedarf es nur weniger oder keiner Anpassungen, um für mehrere Zielplattformen oder gar Geräte gleichzeitig zu entwickeln. Gleichwohl muss der betroffene Entwickler sich dabei nicht auf einen kleinsten gemeinsamen Nenner beschränken, sondern kann dem Benutzer eine in die Plattform integrierte Benutzerschnittstelle durch native Benutzeroberflächen-Elemente, Drag & Drop Unterstützung und mehr bieten.

Dank dem komponentenbasierten Ansatz muss die Anwendung auch kein für sich alleine stehender Komplex bleiben, sondern kann auch in mehrstufige Architekturen eingebunden werden, wie dies sehr oft der Fall ist (man denke an die Verknüpfung mit Datenbanken etc., siehe auch (13)). Weiterhin ist die Erweiterbarkeit ein Wesensmerkmal von Eclipse Applikationen (siehe dazu auch Kapitel 2.2), zusätzlich forciert durch einen integrierten Aktualisierungsmechanismus.

Nicht zuletzt ist auch der große Pool von bestehenden Plugins, die unabhängig voneinander eingebunden werden, eine attraktive Möglichkeit, schnell qualitativ hochwertige Anwendungen zu produzieren. Durch die gezielte Hinzunahme solcher Plugins, die für eine spezifische Problemstellung hilfreich sind, kann der Entwicklungsaufwand – und hierzu gehört auch das Testen solcher Komponenten – drastisch reduziert werden, da sich bereits bestehende Plugins oftmals schon in anderen Anwendungen bewährt haben. Zudem bietet es sich an, eigene Plugins mit dem Extension-Mechanismus erweiterbar zu halten, um so die Anwendung, je nach Bedarf, in mehreren Stufen den Bedürfnissen gerecht werden zu lassen.

Dem gegenüber werden Programmierer mit einem recht komplexen System konfrontiert, das natürlich nicht ohne einen vergleichsweise hohen Einarbeitungsaufwand zu bewältigen ist. Irritierend sind hier auch einige Eclipse spezifische Namenskonventionen die von allgemeiner gebräuchlicheren Begriffen abweichen (es wird etwa der Begriff Plugin sehr weit gefasst). Dies wird erleichtert durch die hohe Verfügbarkeit von Dokumentation, denn neben der Javadoc API Dokumentation und einer Vielzahl von Artikeln exisitieren auch ganze Bücher, die sich nur mit der RCP beschäftigen.

Die Eclipse Rich Client Platform bewegt sich hin zu der Standardplattform für Rich Client Entwicklungen und seine Bedeutung wird zweifelsohne auch in absehbarer Zukunft noch steigen. Die offene Basis und die breite Unterstützung bieten gute Aussichten für die kommende Entwicklung. Insgesamt lässt sich somit wenig Negatives an der RCP finden. Dennoch kann die RCP natürlich kein Wunderwerkzeug für alle Typen von Anwendungen sein und es gibt weiterhin Argumente die dafür sprechen, weniger Abstraktionsstufen zu wählen und plattformnäher zu programmieren. Doch ist sie ein Werkzeug, das für sehr viele unterschiedliche Typen von Projekten attraktive Perspektiven eröffnet.

Literaturverzeichnis

[1] Jeff McAffer, Jean-Michel Lemieux: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications. Addison-Wesley Professional (Oktober 2005)

[2] Glenn Schaare: Varianten-freundliches Software-Design: Die Eclipse-Rich-Client-Platform. (2005) URL: http://www-wi.uni-muenster.de/pi/lehre/ws0506/skiseminar/VariantenManagementEclipse_Schaare.pdf.

[3] OSGi Alliance: http://osgi.org.

[4] Standard Widget Toolkit: http://www.eclipse.org/swt/.

[5] IBM: Eclipse Platform Technical Overview. (April 2006) URL: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[6] wiki.eclipse.org: RCP FAQ. (2006) URL: http://wiki.eclipse.org/index.php/RCP_FAQ.

[7] Dan Rubel: The Heart of Eclipse. (Oktober 2006) URL: http://www.acmqueue.org/modules.php?name=Content&pa=printer_friendly&pid=425&page=1.

[8] Kommerzielles RCP Plugin: http://www.instantiations.com/rcpdeveloper/.

[9] Ed Burnette: Rich Client Tutorial Part 1. (Februar 2006) URL: http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html.

[10] Eclipse.org: Eclipse help system URL: http://help.eclipse.org.

[11] Ed Burnette: Rich Client Tutorial Part 2. (Februar 2006) URL: http://www.eclipse.org/articles/Article-RCP-2/tutorial2.html.

[12] Ed Burnette: Rich Client Tutorial Part 3. (Februar 2006) URL: http://www.eclipse.org/articles/Article-RCP-3/tutorial3.html.

[13] Wayne Beaton, Jeff McAffer: Eclipse Rich Client Platform. (November 2006) URL: http://download.eclipse.org/technology/phoenix/talks/What-is-Eclipse-and-Eclipse-RCP-3.2.6.pdf.

Software Development with Eclipse OSGI

Nikola Veber

UNI Karlsruhe (TH)

Abstract. This article should provide an Overview of the OSGi platform. It should explain the reasons which lead to the creation of OSGi. Frther on, the architecture of the OSGi and some of the application scenarios will be described. The article also gives a brief overview of the Eclipse Equinox project.

Key words: OSGi, Eclipse

1 Introduction

1.1 What is OSGi

OSGi (Open Service Gateway Initiative) specification defines a "standardized, component oriented, computing environment for networked services that is the foundation of an enhanced service oriented architecture."
Using the OSGi specification allows the software components to be installed and managed remotely on a variety of networked devices. These actions can be done without disturbing the function of the device at any moment.
OSGi forms a thin layer on top of the Java Virtual Machine and allows multiple applications to coexist and cooperate in a single JVM. This approach has its problems, like possible security problems or consistency problems, but OSGi offers efficient tools to overcome these difficulties.

2 Technology

2.1 Platforms

The OSGi specification was originally meant to be used in Home Internet gateways and in similar Home Automation scenarios. However, the standard proved to be interesting for other areas of use, for example mobile phones or automotive industry.

Enterprise Software OSGi can be, and is used, in the domain of enterprise software. Enterprise software benefits from OSGi's component model (live deployment for example), security features, etc. Example: IBM WebSphere Everyplace Deployment. In further sections it will be discussed how the Eclipse Equinox can work in a Application Server environment.

Smart Phones OSGi should provide a better platform for business applications running on mobile phones.

Ubiquitous computing and Sensor networks OSGi and its support for remote administrations can be used in scenarios involving a large number of small devices that build a (sensor-) network and cooperate. OSGi can be used to deploy software on the devices. With OSGi, it would be possible to produce almost universal miniature sensor-equipped computers that could act and interact in any given way (in a way defined by a off-the-shelf software that could work on a number of mentioned devices).

Home appliances Certain manufacturers of household devices already support home automation. The devices, like microwave oven or a television, are connected via the power-lines (no new cabling required) and are able of being programmed. Having an unique, standarized platform like OSGi makes it easier to deploy/maintain different applications on the physical devices. It also makes it possible to create standard products, that would fit any given product of a certain type, regardless of its manufacturer.

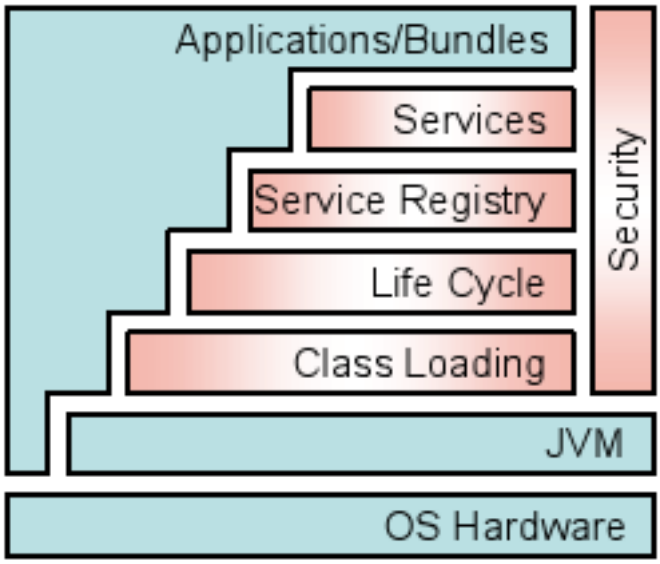
Automotive industry Software is playing a ever-increasing role in a modern car. Since the car manufacturers sell cars on the global level, OSGi could provide an efficient way of adjusting the software to the location specific needs (different traffic rules in different countries). Software could be adjusted in the local branches of companies and then deployed in the actual vehicles.

Eclipse Eclipse is an open source Integrated Development Environment (IDE) which evolved in a Runtime platform, based on a very small runtime core. As noted in [2], OSGi was introduced to Eclipse to meet the requirement the version 2.0 did not meet: plugins couldn't be loaded without a restart. Even though it was possible to develop alternative methods for dynamic plugin loading, Eclipse adopted OSGi specification in order to take advantage of its relative maturity.

2.2 Architecture

The Architecture of OSGi consists of following layers:

- Class Loading
- Life Cycle
- Service Registry
- Services
- Security



Those layers come on top of the JVM. Application bundles are built on top of the layers mentioned above. The layers also represent the major phases in the life cycle of an application and tasks in the platform itself, so those phases will be explained in more detail in the context of the architecture layer being responsible for them.

Class Loading is a critical part of application deployment. This task requres the strict rules of the class loading model to be followed. "The L1 Modules layer defines the class loading policies. The OSGi Framework is a powerful and rigidly specified class-loading model. It is based on top of Java but adds modularization. In Java, there is normally a single classpath that contains all the classes and resources. The OSGi Modules layer adds private classes for a module as well as controlled linking between modules. The module layer is fully integrated with the security architecture, enabling the option to deploy closed systems, walled gardens, or completely user managed systems at the discretion of the manufacturer", reads the OSGi Technology overview [10]

Application bundles consist of Class files, which are organised in packages. The first problem the OSGi Framework must handle is how to treat these classes with respect to other bundles. Should these classes be kept private or should they be shared with other bundles?
In OSGi it is possible to use private bundles, but that results with the larger footprint and decreases the code reusability. In order to avoid these problems, an efficient mechanism for class sharing had to be developed.
In OSGi every bundle can export and import packages. Every package is exported with its unique version, which makes it possible for different versions of

the same package to exist without collisions.
[11] provides more reading on Class Loading in general.

Life Cycle is responsible for the installation, execution and deinstallation of bundles.
Bundles are stored in JAR archives, which contain Manifest. A manifest stores information about the archive. There is a installBundle method available to every Bundle, which basically means that only Bundles can install other Bundles. After the installation, a bundle needs to be resolved before it can be installed. Resolving a bundle means taking care of the package dependencies between bundles.

Security is an important aspect of OSGi, since the concept of using (e.g. Of-The-Shelf) modules from various sources is prone to security problems. Therefore, several security mechanisms are available in OSGi:

- Java 2 Code Security
- Minimized bundle content exposure
- Managed communication links between bundles

Those mechanisms are used to prevent problems that could occure on the level of: Ressources management (files, I/O etc), Code access (classes, methods, packages) and Services.

Service Registry was developed to make the problems of dynamic environment of the OSGi platform easy to manage. It dynamically links different bundles together while tracking their state and dependencies.
With the Service Registry, bundles can:

- Register objects with the Service Registry.
- Search the Service Registry for matching objects.
- Receive notifications when services become registered or unregistered.

As [2] reads: "Objects registered with the Service Registry are called services. Services are always registered with an interface name and a set of properties. The interface name represents the intended usage of the service. The properties describe the service to its audience. For example, the OSGi Log Service would be registered with the org.osgi.service.log.LogService interface name and could have properties such as vendor=acme.
The Service Registry is an in-memory registry. Registrations are dynamic and are dependent on the execution state of the bundles. The OSGi Framework automatically unregisters all services from a stopped bundle, notifying all its dependents. The Service Registry is designed to have very low cost per registered

service. The use of the Declarative Service specification can reduce even this low cost by delaying loading classes from a bundle until services from the bundle are actually used; often achieving an efficiency that a more traditional environment cannot.
The Service Registry enables the OSGi Service Platform to support applications built using a service oriented architecture (SOA). It allows application programmers to develop small and loosely coupled components, which can adapt to the changing environment in real time. The platform operator uses these small components to compose larger systems. The Service Registry is the glue that binds these components seamlessly together."

2.3 Standard Services

Standard services in the OSGi platform are divided in 5 groups:

- Framework Services
- System Services
- Protocol Services
- Miscellaneous Services
- Programming Support

Framework Services include a Permission Admin service, a Conditional Permission Admin service, a Package Admin service, a URL Handler service, and a Start Level service. Those services are optional and are not requred by the specification.

System Services include commonly required services, like Log Service, Configuration Admin service, Device Access service, User Admin service, IO Connector service, Event Admin service, Preferences Service... Bundles save memory footprint on the physical device by using centralised services instead of their own. It is possible to reduce development time by using there well tested and highly reliable components.

- Log Service is used for the logging of diverse information, like warnings, debugging information, errors and so on. It works on the subscriber principle, where bundles can subscribe to receive certain log entries. In that way it is possible to enhance the cooperation between applications and make the whole environment more stable (advanced exception handling).

- Configuration Admin service is used for managing configuration data (in config files or a database).

- Event Admin provides a general and flexible publish and subscribe event mechanism. Supports both synchronous and asynchronous delivery of events.

- Device Access Device Access is the mechanism to match a driver to a new device and automatically download a bundle implementing this driver. This is used for Plug and Play scenarios.

- User Admin This service uses a database with user information (private and public) for authentication and authorization purposes.

- O Connector The IO Connector service allows bundles to provide new and alternative protocol schemes for the Generic Connection Framework.

- Preferences Service A service that provides access to hierarchical database of properties. Similar to the Windows Registry or the Java Preferences class.

Protocol Services are used as an interface between OSGi services and communication protocols used by devices. In that way, it is possible to achieve a higher abstraction level. The Protocol Services are:

- Http Service The Http Service is, among other things, a servlet runner. Bundles can provide servlets, which become available over HTTP. The dynamic update facility of the OSGi Service Platform makes the Http Service a very attractive web server that can be updated with new servlets, remotely if necessary, without requiring a restart.

- UPnP Universal Plug and Play (UPnP) is an emerging standard for consumer electronics. The OSGi UPnP service maps devices on a UPnP network to the Service Registry. Alternatively, it can map OSGi services to the UPnP network.

- DMT Admin The Open Mobile Alliance (OMA) provides a comprehensive specification for mobile device management on the concept of a Device Management Tree (DMT). The DMT Admin service defines how this tree can be accessed and/or extended in an OSGi Service Platform.

Miscellaneous Services

- Wire Admin Normally bundles establish the rules to find services that they want to work with. However, in many cases this should be a deployment decision. The Wire Admin service therefore connects different services together as defined by a configuration. The Wire Admin service uses the concept of a Consumer and Producer service that interchange objects over a wire. The Wire Admin service is supported with the Position, Measurement and State utility classes.

– XML Parser The XML Parser service allows a bundle to locate a parser with desired properties and compatibility with JAXP.

– Initial Provisioning - Initial provisioning of an OSGi device is a multi step process that culminates with the installation and execution of the initial management agent. At each step of the process, information is collected for the next step. Multiple bundles may be involved and this service provides a means for these bundles to exchange information. It also provides a means for the initial Management Bundle to get its initial configuration information.

– Foreign Application Access - A service that servers as an interface between OSGi platform and applications running in it on the one hand, and the applications running outside of the platform on the other.

3 Eclipse and OSGi

As the [9] reads: "The OSGi framework specification forms the basis of the Eclipse Runtime. As of Eclipse 3.0, the Runtime is fully based on the OSGi notion of bundle (equivalent to Eclipse plug-ins). The OSGi framework implementation in Eclipse 3.0 is fully compliant with the OSGi framework specification R3.0. The OSGi implementation in Eclipse 3.1.2 and the upcoming Eclipse 3.2 fully implement the OSGi R4.0 framework specification."
OSGi makes it possible to build Eclipse on top of a very small kernel, making it extremely flexible. The Eclipse Rich Client Platform (RCP) evolved from this architecture. RCP is a minimal set of plug-ins needed to build a rich client application.

4 Equinox

Exuinox is the name of the OSGi R4 core framework specification used in the Eclipse project. Exuinox is a set of Bundles providing standard services, as well as some additional services (for example the Servlet API) to the developers.
OSGi has its obvious advantages on the client-side. Recent developments, however, show that server-side applications can benefit from OSGi, especialy since the adaprion of Eclipse on the server-side. Both scenarios, Client- and Server-Side, are going to be described.

4.1 Client-Side

OSGi is used on the client side to provide the Eclipse RCP basic functionality. Each time Eclipse is started, the required bundles are installed and started automatically. This is, however, not the only way to use OSGi with Eclipse.
As stated in [7] , Equinox is a part of Eclipse RCP, but it is available as a

standalone application, which can be donloaded from the Eclipse download site. Once downloaded, it can be started from the command line:

java -jar org.eclipse.osgi.jar -console

Once running the application, you can obtain list of available commands by typing "?" in the command line.

The most interesting commands for getting started are:

- install [bundle URL] - Installs the bundle from the given URL
- start [bundle or bundle name] - Starts the bundle with the given numeric or symbolic id
- stop [bundle or bundle name] - Stops the bundle with the given numeric or symbolic id
- ss - Reports a summary status of all installed bundles
- diag [bundle or bundle name] - Reports any resolution problems for the bundle with the given numeric or symbolic id

Installing and starting bundles manually needs to be done only once, since the Equinox remembers which bundles were installed and started, and installs and starts them on each startup.
The command line is not the only way of configuring Equinox. It is possible to automate the process of bundle installation and execution by creating "configuration(s)". A Configuration is a set of Bundles and information on how to handle them. In order to create a configuration that includes two Bundles, B1 and B2, you first need a directory structure like this:

```
somedir/
    org.eclipse.osgi.jar
    B1.jar
    B2.jar
    configuration/
        config.ini
```

As you can see, this directory contains the Equinox itself, both of your Bundles and a configuration file, a Java properties file containing information how to use the bundles. An example config.ini could have the following commands:

```
osgi.bundles=B1.jar@start, B2.jar@start eclipse.ignoreApp=true
```

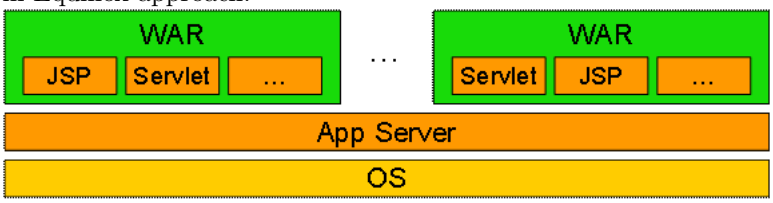
By using certain Eclipse Bundles, like org.eclipse.equinox.common and org.eclipse.update.configurator, it is possible to use features like automatic discovery and installation of bundles.

4.2 Server-Side

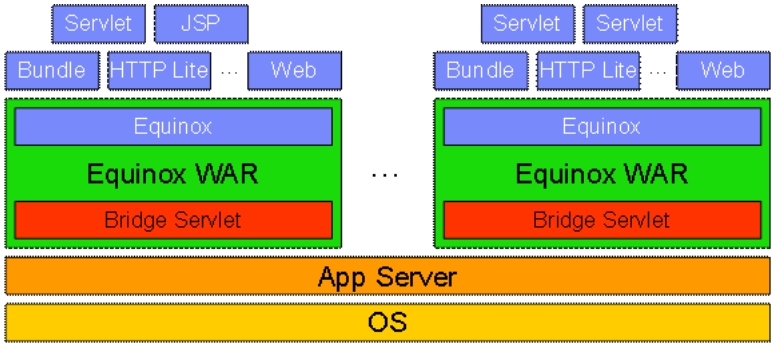
Although OSGi was originaly ment to be used with embedded systems, its dynamic module capabilities made it suitable to be used in standard application

server scenarios. As shown in [6], there are two main approaches; embedding Equinox in the servlet container or embedding the servlet container in Equinox. To a large extent the choice made here does not impact your server application or functionality. The choice is more a function of the infrastructure needs of your environment. For example, are there other standard web applications running, are you using clustering or other infrastructure,

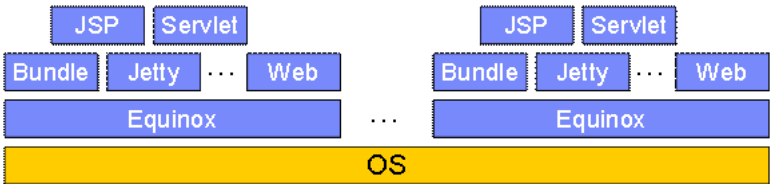
The following three diagrams illustrate the traditional web application server setup and contrasts it with the Equinox in Container approach and the Container in Equinox approach.



Traditional approach



Equinox is embedded in an existing servlet container as shown in Figure 2 by installing a very thin web application. This application contains a Bridge Servlet that launches an Equinox instance inside the application and then forwards all requests to the Equinox-based application (servlets, JSPs,) via a very lightweight HTTP service.



The approach is turned around and Equinox runs an application container (e.g., embedded Jetty) and provides some glue code that wires Equinox-based applications into the application container.

Further reading on how to use Equinox with Application Servers is available at [8],

5 Other OSGi implementations

There is a number of products following the OSGi specification. They could be divided into several gorups, depending on the area of use.

5.1 Enterprise

Siemens Communications Inc. expanded its leadership role in enterprise communications by providing development-friendly, business-focused communication applications across a framework of Service Oriented Architecture (SOA) standards and associated Web services interfaces based on OSGi technology. The Siemens Communications SOA initiative, which began in 2004, is planned to ultimately encompass all key Siemens enterprise innovations and services - including past and future communication solutions – to help maximize the use of reusable communication components with any network or business. Enterprise customers, for example, are utilizing Siemens' SOA-built HiPath OpenScape Software Developer Kit (SDK), which exposes the underlying functions of the HiPath OpenScape solution's presence-smart features and helps to allow developers to easily embed these features directly into enterprise applications.

Paremus' Infiniflow product familys foundations are built upon OSGi technology. The Infiniflow Distributed Service Framework (DSF) enables autonomic deployment, resource optimization and self-healing of OSGi / SCA compliant POJO and Spring based business services. Infiniflow Enterprise Service Grid (ESG) and Infiniflow Enterprise Service Fabric (ESF) provide massive distributed scalability, complex event-driven and high throughput transactional business processing.

5.2 Open Source

Apache Felix Project has entered Incubation at the Apache Software Foundation and will be the foundation's implementation of the OSGi specifications. Felix is a community effort to implement the OSGi Service Platform Release 4, which includes the OSGi framework and standard services, as well as providing and supporting other interesting OSGi-related technologies. The ultimate goal is to provide a fully compliant implementation of the OSGi framework and standard services and to support a community around this technology. Felix currently implements a large portion of the OSGi specifications, but additional work is necessary for full compliance. Despite this fact, the OSGi framework functionality provided by Felix is very stable.

Eclipse Exuinox , as mentioned above, is a OSGi Framework used by Eclipse. It is mentioned again in order to help the reader understand its position on the market.

Knopfelerfish Knopflerfish project has the goal to develop and distribute easy-to-use open source code and build tools and applications related to OSGi technology. Gatespace Telematics, an OSGi Alliance member, maintains and sponsors the Knopflerfish project and has several developers assigned to develop and maintain the Knopflerfish OSGi distribution.

Newton Project is a distributed runtime framework for the dynamic instantiation and subsequent management of complex OSGi / SCA Systems within enterprise environments. Based on a SCA System description, Newton dynamically deploys and maintains availability of relevant OSGi service bundles, and dynamically wires these together across a distributed set of heterogeneous compute resource. The Newton Project has been donated to Open Source by Paremus (www.paremus.com).

5.3 Automotive Electronics

BMW Research (BMW Technik und Forschung GmbH) designs and develops innovative infotainment- and telematic-systems in various automotive projects, such as 3GT, Ertico GST or stadtinfokln. To meet the challenges of a vehicle infotainment system, which connects the vehicle to the outside world, an open and modular platform is necessary that is based on a standardized software architecture. BMW Research uses ProSyst's OSGi-certified service delivery platform, mBedded Server (prosyst.com), as an enabling technology for the development and the deployment of applications to the vehicle infotainment platform and other devices to allow seamless information exchange.

Bombardier Transportation utilizes ProSyst Software's OSGi-based product mBedded Server (prosyst.com) as well as the JVM J9 from IBM OTI for its Remote Diagnosis System (RDS), a wireless, remote data transmission system that improves and extends the maintenance, administration and fleet management functionality for rail vehicles. Thanks to RDS, railway companies can carry out remote monitoring and diagnosis of their trains and access important data to improve the operation of their vehicles and fleet. One of the first applications of RDS is the 24 electric locomotives ALP 46, that will be delivered to the American New Jersey Transit.

Ertico GST - The EC-supported GST project is focused on creating a Global System for Telematics (GST) enabling on-line safety services. Organized into seven subprojects and seven test sites, the GST consortium is developing an open and standardized framework architecture for end-to-end telematics. GST

is offering a reference implementation using an OSGi Service Platform as the execution environment.

Other developments in this area include the work of Volvo, Siemens VDO Automotive and other companies.

5.4 Smart Home

Blekinge Tekniska Hgskola - SOCLAB - Using the OSGi compliant Gatespace Telematics' Ubiserv, Societies of Computation Laboratory at Blekinge Institute of Technology (Ronneby, Sweden) has developed a Service-Oriented Layered Architecture for Communicating Entities (SOLACE). In essence, the SOLACE platform provides support for pivotal abstraction layers of open computational systems, i.e., mediation fabric, system interaction, and cognitive domains. These platform features manifest themselves by means of spontaneous network management, peer-to-peer system interoperability, and multidimensional system modeling. Examples of application domains include network-centric warfare, distributed home healthcare, and sustainable infrastructures.

BSH (Bosch und Siemens Hausgerte GmbH) - BSH, Europe's largest white goods manufacturer, presented a full range of built-in and freestanding appliances with networking capabilities at the HomeTech 2002, European's leading household fair. The heart of the system is a service gateway with ProSysts open, scalable and modular service delivery platform and service packages based on OSGi technology (www.prosyst.com). Serve@Home products follow a simple "plug--use" system, where the end-user can easily expand, remotely manage and use the products and services via mobile phone, PDA or computer. The Serve@Home kitchen product line is part of the overall smart home concept. It provides the consumer with secure, convenient and individual residence monitoring and controlling from anywhere.

Cisco - Gatespace Telematics OSGi Service Platform-based technology is deployed in the CiscoWorks 2000 Service Management Solution. The CiscoWorks 2000 Service Management Solution manages service levels between enterprises and internal or external service providers to ensure quality-of-service. Gatespace Telematics Distributed Service Platform (GDSP) provides the Service Level Manager (SLM) Collection Manager, a component of the Service Management Solution, with remote management functionality enabling dynamic deployment and life-cycle management of Cisco's Java-based control agents.

Other OSGi-based products are available on this market, and range from small home electronics (small home gateways), to the management of diverse end-user services in the vertical branches of electricity commercialization, home automation, energy management and conservation, used by Electricit de France (EDF) insidte their M@jordom project.

6 Conclusion

The OSGi specification is still not a widely adopted technology, but the benefits of using it in above mentioned scenarios make it reasonable to believe that the adoption of OSGi tends to increase. It is to be expected, that a number of portable devices (smart phones, single board computers, home appliances...) that have a JVM and are able of supporting OSGi, increases. That results with a stronger demand for remote administration and maintanance standards, like OSGi.

7 Resources

Links to websites used as references:

- [6] "Eclipse, Equinox and OSGi", Jeff McAffer, Simon Kaegi, January 2007, http://www.theserverside.com/tt/articles/article.tss?l=EclipseEquinoxOSGiasrc=EM$_N LN_8$74425$uid =$ 5581085
- [7] "Equinox quickstart guide", http://www.eclipse.org/equinox/documents/quickstart.php
- [8] "Equinox Server-side Quickstart", http://www.eclipse.org/equinox/server/http$_q uickstart.php$
- [9] "osgi, the footings of the foundation of the platform", http://www.eclipse.org/osgi
- [10] "OSGi Technology", http://www.osgi.org/osgi$_t echnology/?section = 2$
- [11] "Inside Class Loaders", Andreas Schaefer, http://www.onjava.com/lpt/a/4337

References

1. McAffer / Lemieux: "Eclipse Rich Client Platform"
2. OSGi Alliance: "About the OSGi Service Platform", Technical Whitepaper, Revision 4.1, 11 November 2005
3. "Implementation of Initial Provisioning Function for Home Gateway Based on Open Service Gateway Initiative Platform", SangOk, HoJin Park Electronics and Telecommunications Research Institute
4. "The Eclipse 3.0 platform: Adopting OSGi technology", O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, T. Watson
5. "Leveraging OSGi Technology", A Business Whitepaper, Revision 1.0, 5 October 2005
6. "Eclipse, Equinox and OSGi", Jeff McAffer, Simon Kaegi, January 2007
7. "Equinox quickstart guide"
8. "Equinox Server-side Quickstart"
9. "osgi, the footings of the foundation of the platform"
10. "OSGi Technology"
11. "Inside Class Loaders", Andreas Schaefer"