



Verteilte Client/Serveranwendungen auf Basis von Eclipse/Equinox

Eclipse-Riena-Projekt

 Quellcode
 auf CD!

>> CHRISTIAN CAMPO

Eclipse ist nicht nur eine der beliebtesten Softwareentwicklungsplattformen (IDEs), sondern in Form von Eclipse RCP auch eine beliebte Plattform zum Erstellen von Java-Clients. Zunehmend wird der Runtime Kernel Equinox auch als Basis für serverbasierte Anwendungen verwendet. Beispiele dafür sind die Projekte WTP [1] und RAP [2]. Riena [3] setzt auf diesen bisherigen Erfahrungen auf und positioniert sich als Plattform für verteilte Multi-Tier-Client/Serveranwendungen.

Riena, ein relativ neues Eclipse-Projekt, liefert eine Basis für Client/Serveranwendungen für Endbenutzer im Unternehmensumfeld auf Grundlage von Eclipse-Technologien. Ein Schwerpunkt ist der Verteilungsaspekt von Komponenten, der bisher im Eclipse-Umfeld durch bestehende Projekte nicht ausreichend abgedeckt wurde. Dieser wird im Folgenden im Detail be-

handelt. Ein weiterer Schwerpunkt, der in einem eigenen Artikel beschrieben wird [4], betrifft das auf die Bedürfnisse von Endbenutzern abgestimmte User Interface und Navigationskonzept.

Komponenten und Services

Equinox, die Referenzimplementierung der OSGi-Spezifikation, ist die Basis aller Eclipse-Anwendungen und liefert ein

leichtgewichtiges und flexibles Komponentenmodell, das sich sowohl für client- als auch für serverbasierte Anwendungen eignet. Komponenten – oder in OSGi-Terminologie Bundles – definieren nicht nur ihre Abhängigkeiten zu anderen Bundles, sondern auch die von ihnen angebotenen Schnittstellen und Erweiterungen in ihrem Manifest. Zusätzlich ist für jedes Bundle ein Lifecycle-Management definiert. Diese Definition macht die Implementierung eines Bundles etwas aufwändiger, erleichtert aber die Strukturierung und die Wiederverwendung von Bundles.

Zusätzlich bietet OSGi ein Konzept für Services, die von Komponenten (Bundles) angeboten werden und eine weitere Entkopplung der Komponenten gestatten. Ein Service besteht aus einem Interface und einer Instanz, die zusammen zur Laufzeit in einem globalen Verzeichnis (OSGi Registry) registriert werden. Die Verbindung der Services untereinander (also Service A benutzt Service B) geschieht entweder per API durch den aufrufenden Service

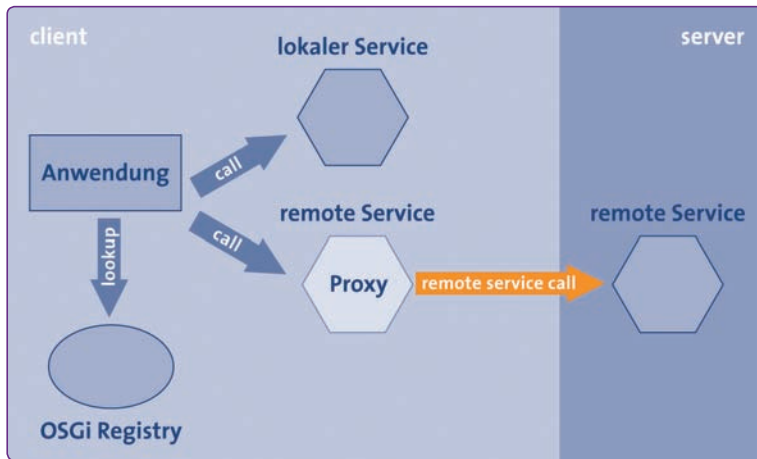


Abb. 1: Lokale und Remote-OSGi-Services

anderen POJOs abhängig, die jeweils eine bestimmte Schnittstelle implementieren. Neben dem Vorteil der Unabhängigkeit lassen sich diese Objekte einfacher testen, weil ihnen zum Testablauf Mock-Objekte statt der eigentlichen Serviceimplementierungen übergeben werden (also Mock B wird Service A als Referenz übergeben).

Verteilte Services

Das durch OSGi spezifizierte Konzept des Zusammenspiels von Komponenten und Services ist vielfältig anwendbar. Dabei spielt es keine Rolle, ob es sich um eine Clientanwendung handelt, die auf Basis von RCP gebaut ist, oder ob Equinox auf dem Server eingesetzt wird, um eine Webanwendung, zum Beispiel auf Basis von WTP oder RAP, zu betreiben. OSGi spezifiziert bisher allerdings keine Möglichkeiten, wie Services, die verteilt auf Client und auf Server existieren, untereinander verbunden werden können. Hier setzt Riena ein, indem es für OSGi-Services auf der Basis von Web Services einen transparenten Remote-Service-Aufruf implementiert. Im Vordergrund für Riena steht hierbei die Transparenz bei der Benutzung von lokalen und entfernten Services. Anwendungsentwickler sollen sich auf ihre fachliche Funktionalität konzentrieren können und gleichermaßen Services auf Client und Server benutzen können. Dabei kümmert sich Riena nicht nur um die Serialisierung aller Parameter und das Dispatching der Aufrufe, sondern um die Übertragung der Benutzersession. Durch diesen Mechanismus kann man auf jeder Plattform sowohl auf die Identität des Benutzers als auch auf seine damit verknüpften Rechte zugreifen. Die notwendige Unterscheidung nach Client (Single-user) und Server (Multiuser) erfolgt innerhalb von Riena und nicht innerhalb der Anwendung (Abb. 1).

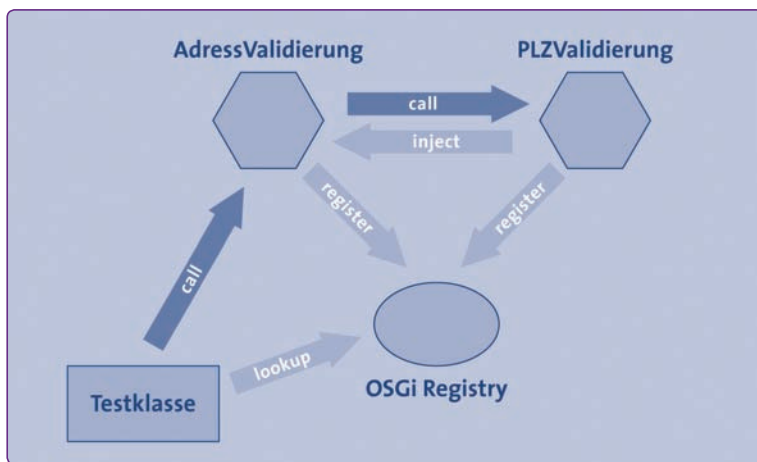


Abb. 2: Lokaler OSGi-Service

(Service A sucht Service B) oder viel besser per Dependency Injection (Service B wird Service A als Referenz übergeben). Die konsequente Benutzung von Depend-

cy Injection gestattet die Reduzierung der fachlichen Objekte auf einfache POJOs ohne Abhängigkeiten zum Laufzeitcontainer. Diese POJOs sind dann nur noch von

Listing 1

Implementierungen

```

public class PlzValidierungImpl implements
    IPlzValidierung {
    System.out.println("AdressValidierung Service
    start");
}

public PlzValidierungImpl() {
    System.out.println("PlzValidierung Service start");
}

private IPlzValidierung plzValidierung;

public boolean testeAdresse (Adresse adresse) {
    System.out.println("testeAdresse");
    return plzValidierung.testePLZ(adresse.getPlz());
}

public void bind(IPlzValidierung plzValidierung) {
    System.out.println("bind PlzValidierung");
    this.plzValidierung = plzValidierung;
}

public void unbind(IPlzValidierung plzValidierung) {
    System.out.println("unbind PlzValidierung");
    this.plzValidierung = null;
}

public class AdressValidierungImpl implements
    IAdressValidierung {
    System.out.println("AdressValidierung Service
    start");
}

public AdressValidierungImpl() {
    System.out.println("AdressValidierung Service start");
}

private IPlzValidierung plzValidierung;

public boolean testeAdresse (Adresse adresse) {
    System.out.println("testeAdresse");
    return plzValidierung.testePLZ(adresse.getPlz());
}

public void bind(IPlzValidierung plzValidierung) {
    System.out.println("bind PlzValidierung");
    this.plzValidierung = plzValidierung;
}

public void unbind(IPlzValidierung plzValidierung) {
    System.out.println("unbind PlzValidierung");
    this.plzValidierung = null;
}
    
```

OSGi-Service everywhere

Allein durch die Verwendung von Equinox auf Client und Server ergibt sich automatisch, dass viele Basisservices wie Logging, aber auch Konfiguration auf beiden Plattformen identisch benutzt werden können. Das erlaubt es, Komponenten ohne direkte Abhängigkeiten zur Plattform gemeinsam zu entwickeln. Eine Abhängigkeit ergibt sich natürlich, falls eine GUI (Client) oder eine Datenbank (Server) angesteuert wird. An-

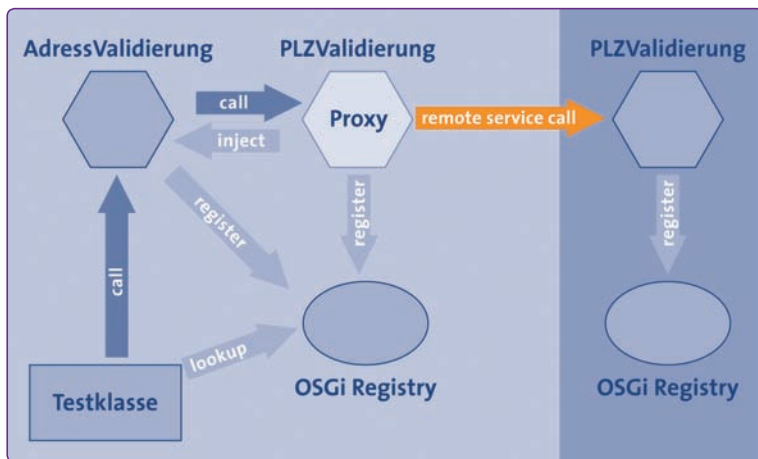


Abb. 3: Remote-OSGi-Service

Und nun werden alle Bundles zusammen als Typ *OSGi Framework* in Eclipse gestartet. Nach dem Start werden zuerst alle Bundles aktiviert. Das erzeugt folgende Ausgabe:

```
osgi> PlzValidierung Service start
AdressValidierungImplService start
bind PlzValidierung
testeAdresse
testPLZ
60329 is valid=true
```

Das heißt: Zuerst wird das Bundle mit der *PlzValidierung* gestartet, dann die *AdressValidierung*. Diese bekommt dann die *PlzValidierung* über die *bind*-Methode injected und danach läuft der Test. Durch die Benutzung des Parameters *-console* kann man sich mit der Kommandozeile einen Überblick über die laufenden Bundles und Services verschaffen und einzelne Bundles stoppen oder starten. Die Liste der geladenen Bundles und Services sieht in gekürzter Form folgendermaßen aus:

```
osgi> ss

Framework is launched.

Id State Bundle
11 ACTIVE em.plz_1.0.0
15 ACTIVE em.common_1.0.0
19 ACTIVE em.adress_1.0.0
20 ACTIVE em.test_1.0.0
35 ACTIVE org.eclipse.riena.core_1.0.0.M2

osgi> status
Framework is launched.

{em.common.IPlzValidierung}={service.id=35}
{em.common.IAdressValidierung}={service.id=36}
```

Stoppt und startet man einzelne Bundles, lässt sich erkennen, wie dynamisch Services entbunden oder wieder neu gebunden werden (Abb. 2):

```
osgi> stop 11
unbind PlzValidierung

osgi> start 11
PlzValidierung Service start
bind PlzValidierung
```

Soweit ein einfaches Beispiel aus mehreren Komponenten mit Dependency Injection, in dem fast ausschließlich Standard-OSGi-Funktionalität benutzt wird. Die bisher einzige Riena-Spezialität ist die *Inject*-Klasse, die Unterstützung innerhalb der *AdressValidierungs*-Komponente bietet.

Remote-OSGi-Services

Eines der zentralen Anliegen von Riena ist die Verteilung von Komponenten zwischen Client und Server. Daher wird das Beispiel so geändert, dass die *PlzValidierungs*-Komponente auf den Server verschoben wird. Das heißt, es werden zwei JVMs erzeugt, in denen jeweils eine OSGi Runtime gestartet wird. Auf dem „Server“ wird das Commons Bundle (mit den Interfaces) und die *PlzValidierung* installiert. Auf dem „Client“ wird Commons, die *AdressValidierung* und die *Testklasse* installiert. Die Bundles sind unabhängig voneinander und die *PlzValidierung* wird sofort auch in einem Server als lokaler OSGi-Service funktionieren. Nun muss Riena mitgeteilt werden, dass dieser OSGi-Service auch als Web-Service-Endpunkt remote zu erreichen ist. Das geschieht bei der Registrierung mithilfe von

Service-Properties. Das heißt, die Klasse *PlzValidierungsImpl* bleibt unverändert, aber der Activator ändert sich:

```
public class Activator {
    public void start(BundleContext context) {
        Hashtable props = new Hashtable();
        props.put("riena.remote", "true");
        props.put("riena.remote.protocol", "hessian");
        props.put("riena.remote.path", "/"
            + PLZValidationService");
        context.registerService(IPlzValidierung.class,
            getName(), new PlzValidierungImpl(), props);
    }
}
```

Startet man dieses Bundle zusammen mit einigen Bundles aus Riena, die für den Server vorgesehen sind, so sieht man auf der Konsole eine Info, dass der Service mit einer URL verbunden wurde:

```
osgi>
PlzValidierung Service start
... DEBUG [Start Level Event Dispatcher] org.eclipse.
riena.internal.communication.publisher.hessian.
HessianRemoteServicePublisher published web service.
protocol=hessian, url=http://localhost/hessian/
hessian/PLZValidationService, interface=em.common.
IPlzValidierung
```

Eine Riena-Komponente (der Publisher) hat hier einen lokalen OSGi-Service mit bestimmten Properties erkannt und automatisch für das Hessian Protocol registriert und publiziert. Wird der lokale OSGi-Service später wieder gestoppt, kann er auch nicht mehr remote aufgerufen werden. Die fachliche Klasse merkt von alledem nichts. Remote-Service-Aufrufe sind für sie dasselbe wie lokale Aufrufe.

Noch fehlt das Erzeugen eines Remote-Service-Proxies auf dem Client, der alle Aufrufe auf die Remote-Instanz weiterleitet. Dafür gibt es in Riena eine Factory, die genau diese Aufgabe erfüllt und den Proxy als lokalen OSGi-Service registriert. Die Original-*PlzValidierung*-Komponente wird nicht mehr auf dem Client installiert. Das ist nicht nötig, da jetzt die *remote*-Komponente aufgerufen werden soll. Dafür benötigt man nur folgenden Code:

```
new RemoteServiceFactory().createAndRegisterProxy
(IPlzValidierung.class,
"http://localhost/hessian/
PLZValidationService",
"hessian");
```




Die Variable *appserver* in diesem Beispiel kann nun in einem eigenen umgebungsabhängigen Bundle über eine Extension definiert werden. Die umgebungsabhängigen Parameter werden so in einem Bundle zentral gehalten. Durch den Austausch dieses Bundles kann die gesamte Anwendung für verschiedene Umgebungen wie Integration, Abnahme oder Produktion konfiguriert werden.

Security

Ein weiterer wichtiger Baustein für eine verteilte Architektur ist der jeweilige Kontext des Benutzers. In Unternehmensanwendungen wird man nicht umhinkommen, sich mehr oder weniger aufwendig zu authentifizieren und diese Authentifizierungsinformation jedem beteiligten Service zur Verfügung zu stellen. Da darf es keine Rolle spielen, ob der Service auf dem Client oder auf einem der Server liegt. Der Kontext beinhaltet:

- Sessioninformationen (SessionId)
- Subject bzw Principal (Benutzer)
- Permissions (Rechte)

Funktionalitäten im Umfeld von Security in Riena sind damit:

- Überprüfen der Credentials und Erzeugen einer Session
- Jederzeitiger Zugriff (auf Client und Server) auf den „aktuellen“ User (Principal)
- Jederzeitiger Zugriff auf die „aktuellen“ Permissions und Überprüfung dieser
- SSL-Zertifikate für sichere Web-Service-Kommunikation

All diese Themen sind natürlich in Java bereits durch eigene APIs präsent. Zur Authentifizierung gibt es *LoginContext*, *CallbackHandlers*, *Principal*, *Subject* und *JavaPermission*. Der Verteilungsaspekt von Riena fügt diesen Komponenten allerdings eine zusätzliche Komplexität hinzu:

- Eingabe der Credentials auf dem Client, Überprüfung auf dem Server
- Zugriff auf den Principal auf dem Client (Singleton) und auf dem Server (im Kontext eines Requests)
- Zugriff auf die Permissions analog zum Zugriff auf den Principal auf Client und Server

Riena enthält Services, die das Thema Security erleichtern. Dazu zählen mehrere Remote-Services zur Authentifizierung und zur Ermittlung der Benutzerrechte. Jedem erfolgreich eingeloggten Benutzer wird eine Session zugewiesen, die bei jedem Remote-Service-Aufruf transparent mitgegeben wird. So gelangt die Sessioninformation vom Client zu jedem der beteiligten Server. Über die Session ermittelt nun Riena automatisch den Benutzer und seine Rechte und stellt diese auf dem Server jeder Komponente zur Verfügung.

Was heißt das konkret? Riena enthält als Beispiel den *SubjectHolderService*. Über die Methode *fetchSubjectHolder()*. *getSubject()* kann sich jede Komponente das *Subject* geben lassen, das einen oder mehrere *Principal* des gerade eingeloggten Benutzers enthält. Das funktioniert auf dem Client, auf dem immer nur ein Benutzer gleichzeitig eingeloggt ist, genauso wie auf dem Server, wo parallel in jedem Thread ein anderer Benutzer aktiv ist. Jede fachliche Komponente kann sich über die *Inject*-Klasse diesen Service injecten und hat damit jederzeit Zugriff auf den Benutzer, für den die jeweilige Funktionalität aufgerufen werden soll. Dieses Beispiel verdeutlicht, wie Riena eine homogene Umgebung auf Client und Server schafft, die die Entwicklung von Komponenten erleichtert.

Der Verteilungsaspekt

Wie das Security-Beispiel zeigt, ist ein wesentliches Ziel von Riena, bestehende und neue Komponenten in verteilten Client/Serverumgebungen zu unterstützen. Um dieses Ziel zu erreichen, ist es manchmal notwendig, zusätzliche Schnittstellen zu definieren oder auch bestehende geschickt zu benutzen.

Die Unterstützung von Persistenz ist hier ein wichtiges Thema. Da der Client selbst keine Abhängigkeiten zu einem Persistenz-Layer hat, müssen alle Daten über Remote-Services geladen werden. Die veränderten Daten werden ebenfalls über Remote-Services gespeichert. Bei großen Anwendungen lohnt es sich, die Datenmenge für das Speichern eines großen Geschäftsvorfalles so klein wie möglich zu halten, da der Client oft nur über eine langsame Datenleitung mit dem Server verbunden ist. Im Idealfall überträgt man dann nur die wesentlich geringere Liste der veränderten Felder zurück auf

den Server. Dort werden die Originaldaten von der Datenbank geladen (was um ein Vielfaches schneller geht) und die Veränderungen werden in diesen Datenbestand übernommen und gespeichert. In Riena wird dieses Konzept durch die Komponente *ObjectTransaction* unterstützt.

Ein weiteres wichtiges Thema sind Softwareupdates. Da Client und Server sowohl von ihren Schnittstellen als auch bei den verwendeten gemeinsamen Datenobjekten eng miteinander verzahnt sind, ist es notwendig, bei jeder neuen Version des Servers auch alle Clients mit einer neuen Version zu versorgen. Das muss gerade im Unternehmensumfeld fehlerfrei und ohne manuelles Eingreifen des Endbenutzers funktionieren.

Es gibt noch viele weitere Themen, in denen Riena bestehende erfolgreiche Komponenten in Zukunft erweitern wird, um diese besser in einem verteilten Umfeld einzusetzen. Beispiele sind Reporting oder Client Monitoring.

Fazit

Client-/Serveranwendungen arbeiten durch ihr verteiltes Komponentenmodell unter besonderen Bedingungen. Die zentrale Aufgabe von Riena ist es, durch zusätzliche Services und APIs eine stabile Infrastruktur für diese Art von verteilten Anwendungen zu erstellen. Und dies auf der bewährten Basis von Equinox/OSGi.



Christian Campo ist als IT-Consultant bei der compeople AG tätig. Er beschäftigt sich vor allem mit der Entwicklung von innovativen Onlineanwendungen und serviceorientierten Architekturen im Java-EE-Umfeld. Bei der compeople AG ist er maßgeblich am Design und der Entwicklung der Smart-Clientplattform beteiligt. Darüber hinaus leitet er zurzeit das Eclipse Technology Project Riena.

>> Links & Literatur

- [1] WTP: www.eclipse.org/webtools
- [2] RAP: www.eclipse.org/rap
- [3] Riena: www.eclipse.org/riena
- [4] Heiko Barth: Riena User Interface, in: Eclipse Magazin Vol. 16, S. 53-58.