AJEER: Eclipse und Security Aspekt-orientiert

Security Does Matter

Quellcode auf CD

>> PETER FRIESE, MARTIN LIPPERT UND HEIKO SEEBERGER

Mehr und mehr Anwendungen werden auf Basis von Eclipse RCP entwickelt. Der Anteil der Anwendungen, die im Unternehmensumfeld eingesetzt werden und im Bereich Security besondere Anforderungen stellen, nimmt ebenfalls zu. Bisher bietet Eclipse jedoch keine native Unterstützung für dieses wichtige Thema. Dieser Artikel stellt einen Ansatz vor, der Aspekt-orientierte Techniken nutzt, um eine Funktions-orientierte Autorisierung in einem RCP Frontend zu ermöglichen.

Security hat viele Facetten: Unter diesen Begriff fallen Themen wie Authentifizierung („Ist der Anwender tatsächlich die Person, für die er sich ausgibt?"), Autorisierung („Welche Funktionen darf der Anwender ausführen, und welche Daten darf er sehen?") und Sicherheit des Programmcodes (insbes. Schutz vor Manipulation des Codes). Für all diese Themen gibt es im Java-Universum diverse Frameworks, deren Einsatzmöglichkeit teilweise jedoch von der eingesetzten Basistechnologie abhängt. So ist das Thema Authentifizierung und Autorisierung im Web-Umfeld bereits so gut abgedeckt, dass es eigentlich kein Problem darstellt, eine Webapplikation auf geeignete Weise und mit vertretbarem Aufwand abzusichern. Für Rich Clients sieht das ein wenig anders aus. Insbesondere für Eclipse-RCP-Applikationen existiert bislang keine Lösung, die out-of-the-box verwendbar ist.

Bevor wir einen Überblick über die derzeit diskutierten Möglichkeiten geben, ist eine kurze Aufstellung der typischen Anforderungen angebracht: Zunächst ist es unerlässlich, dass der Anwender sich an der Applikation anmeldet – üblicherweise mit einem Passwort-Dialog. Die erfragten Anmeldeinformationen werden dann an ein Backend-System übergeben, das die Gültigkeit dieser Informationen überprüft und im Erfolgsfall ein Session Token ausgibt, das im weiteren Verlauf der Kommunikation zwischen Client und Backend verwendet wird. Das Token lässt idealerweise keinen Rückschluss auf das Passwort des Benutzers zu, um ein Hacken der Verbindung zu erschweren.

Nachdem also die Identität des Benutzers festgestellt worden ist, sollen einerseits die Daten eingeschränkt werden, die er sehen darf, und andererseits die Funktionen, die er ausführen darf. Wir wollen uns hier nicht mit dem Filtern der sichtbaren Daten beschäftigen (diese Aufgabe fällt eher in den Bereich des Backends), sondern mit der Frage, wie man die vom Benutzer ausführbaren Funktionen effektiv einschränken kann.

Die vom Benutzer ausführbaren Funktionen werden in Eclipse durch Actions ausgelöst. Actions implementieren entweder das Interface *IAction* oder das Interface *IActionDelegate*. Das Interface *IAction* definiert die Methoden *run()* und *runWithEvent(Event event)*, in denen die Funktion dann ausgeführt wird. Außerdem können Actions aktiv oder inaktiv sein – dies wird über die Property *enabled* gesteuert. Sowohl die *run()-/runWithEvent()*-Methoden als auch die *enabled* Property stellen gute Kandidaten für die Berechtigungssteuerung dar.

Diskussion der möglichen Ansätze

- *Die Deaktivierung ganzer Plug-ins* (z.B. mit Eclipse JAAS; André Oosthuizen hat dazu zahlreiche Beiträge in seinem Blog veröffentlicht [1]): Sie ist nur dann sinnvoll, wenn die Funktionalität der Anwendung so gekapselt wird, dass sie schön auf einzelne Plug-ins verteilt ist und der Anwender durch das Abschalten der Funktionen nicht zu sehr eingeschränkt wird. Das Abschalten des Wertpapierverwaltungs-Plug-ins wäre z.B. für einen Bankangestellten, der an der Kasse arbeitet, noch sinnvoll. Kontraproduktiv wäre jedoch das Abschalten des Personalverwaltungs-Plug-ins für einen Mitarbeiter in der Rentenabteilung. D.h., dass es nicht ausreicht, die Plug-ins entlang der fachlichen Entitäten zu schneiden. Man muss noch feingranularer schneiden und sowohl die Entitäten als auch das, was man damit machen will, beachten. Ergo: Dieser Ansatz ist mit viel Nachdenken bzw. aufwendigem Refactoring verbunden und führt zu einer schwer wartbaren Zersplitterung der Anwendung.
- *Capabilities*: Kim Horne hat dazu auf der EclipseCon 2005 im Rahmen ihrer Session „Addressing UI Scalability Issues in Eclipse" [2] einen Vorschlag gemacht. Prinzipiell ist das keine schlechte Idee, allerdings kann der Anwender ja problemlos eine deaktivierte Capability wieder anschalten.
- *Die Einführung einer eigenen Oberklasse* für alle Actions, die autorisiert werden müssen: An sich ein vernünftiger Ansatz, allerdings bekommt man die Default Actions der Eclipse-Plattform nicht zu fassen.
- Der hier vorgestellte Weg nutzt *Aspekt-orientierte Programmierung (AOP)*, um die Berechtigung des Benutzers auf dem Client durchzusetzen. Es werden ausschließlich Aktionen, für die der Benutzer die erforderliche Berechtigung besitzt, angezeigt bzw. ausgeführt.

AOP in OSGi-Anwendungen

Naheliegenderweise verwenden wir als Aspekt-orientiertes Hilfsmittel AspectJ [3], weil AspectJ der De-facto-Standard

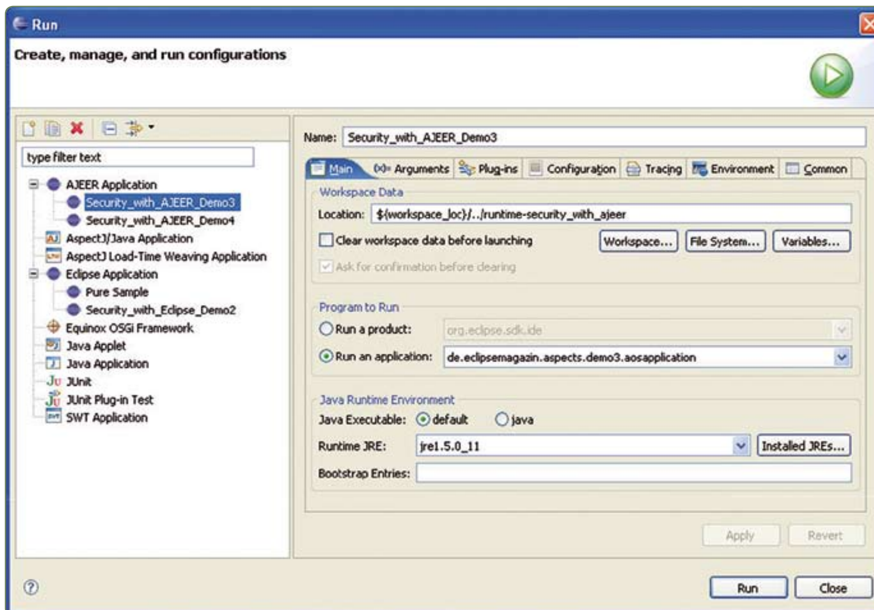

Abb. 1: AJEER Launcher

für AOP mit Java und ebenfalls ein Eclipse-Projekt ist, sodass mit AJDT [4] eine schöne Integration in die Entwicklungsumgebung genutzt werden kann. Allerdings ist das Aspect Weaving von AspectJ zunächst einmal mit Blick auf „normale" Java-Anwendungen implementiert und harmoniert nicht ohne Weiteres mit dem Plug-in-Konzept von Eclipse. Rufen wir uns daher ins Gedächtnis zurück, wie RCP-Anwendungen aufgebaut sind:

Applikationen auf Basis von Eclipse bzw. OSGi [5] bestehen aus Plug-ins genannten Modulen (in der OSGi-Welt werden die Module als Bundles bezeichnet). Bundles definieren im so genannten Bundle-Manifest (META-INF/MANIFEST.MF), welche Packages zum externen API gehören und somit für andere Bundles sichtbar sind. Weiterhin wird festgelegt, welche Packages importiert werden bzw. zu welchen anderen Bundles Abhängigkeiten bestehen. Diese Informationen werden von der OSGi Runtime zur Laufzeit ausgewertet und beim Classloading berücksichtigt. Letztendlich besitzt jedes Bundle einen eigenen ClassLoader, der nur die eigenen Klassen und die importierten Klassen sieht. Dadurch wird für jedes Bundle ein eigener, abgeschotteter Classpath definiert.

Listing 1

```
<extension point="org.eclipse.ajeer.
                         weavingruntime.aspects">
  <aspect class="org.eclipse.ajeer.examples.hello.
                         aspects.HelloAspect"/>
</extension>
```

Listing 2

```
osgi.framework.extensions=org.eclipse.ajeer.
                         bytecodetransformer
osgi.bundles=...,org.eclipse.ajeer.
                         weavingruntime@:start,
      org.eclipse.ajeer.bytecodetransformer
...
```

Listing 3

```
pointcut enablement(IAction action):
  target(action) && execution(public boolean
                         IAction.isEnabled());
```

Soll AspectJ innerhalb eines Bundles eingesetzt werden, kann direkt das AspectJ Tooling und das Compile-Time Weaving genutzt werden. Das Bundle wird vom AspectJ Compiler kompiliert, und die Aspekte werden in den Code des Bundles eingewoben. Soweit alles kein Problem. Was passiert aber, wenn ein Aspekt in einem Bundle implementiert werden soll, der andere bereits existierende Bundles beeinflusst? Genau das wollen wir ja schließlich in unserem Security-Anwendungsfall nutzen.

Die Antwort heißt natürlich zunächst Load-Time Weaving. Leider lässt sich durch den sehr speziellen Classloading-Mechanismus von OSGi das Standard Load-Time Weaving von AspectJ nicht ohne Weiteres nutzen. Das Hauptproblem besteht darin, dass durch das Aspect Weaving der Code der verwobenen Klasse verändert wird. Dabei werden Abhängigkeiten zu dem eingewobenen Aspekt in den Bytecode der Klasse eingefügt. In einer gängigen Java-Anwendung ist dies kein Problem, weil der Aspekt und die verwobene Klasse im gleichen Classpath liegen und die zusätzlichen Abhängigkeiten damit keine Probleme verursachen. In der OSGi-Welt entstehen aber Abhängigkeiten, die im OSGi-Manifest nicht berücksichtigt sind. Können sie ja auch nicht, denn wir wollen mit unserem Aspekt ja auch solche Plug-ins beeinflussen, die wir NICHT implementiert haben.

Load-Time Weaving für OSGi

Glücklicherweise ermöglicht die Eclipse-OSGi-Implementierung (Equinox-) Anpassungen am Classloading-Standardverhalten. Seit Eclipse 3.2 gibt es dafür den Hookable Adaptor [6]. Über so genannte Framework Extension Fragments, d.h. Fragment Bundles für das System Bundle, können *ClassLoadingHook*s beigesteuert werden. Dieser Mechanismus kann beispielsweise ausgenutzt werden, um der OSGi Runtime selbst das Load-Time Weaving beizubringen.

Eine entsprechende Erweiterung für die Equinox-OSGi-Implementierung existiert bereits, sogar in zwei unterschiedlichen Facetten: Zum einen gibt es das Equinox-Incubator-Projekt „Aspects", welches eine entsprechende OSGi-Extension für AspectJ bereitstellt. Zum anderen existiert ein Sourceforge-Open-Source-Projekt namens AJEER [7], welches ebenfalls eine solche Erweiterung implementiert. Beide Projekte setzen auf der Original-AspectJ-Weaving-Implementierung auf und binden dieses Weaving geeignet in die OSGi Runtime ein.

Das Prinzip bei beiden Projekten ist sehr ähnlich: Bundles können Aspekte enthalten und deklarieren, dass diese Aspekte mit anderen Bundles verwoben werden sollen. Im Falle von AJEER wird dies über den Extension-Point-Mechanismus realisiert. Mit einer passenden Extension kann also ein Plug-in einen Aspekt der Runtime „bekanntmachen". Dieser Aspekt wird grundsätzlich in alle anderen Bundles des Systems eingewoben, wenn Klassen geladen werden. Einzig die Definition des Aspekts selbst (genauer gesagt, die enthaltenen Pointcuts) entscheidet, ob der Aspekt die gerade zu ladende Klasse beeinflusst oder nicht.

Im Equinox-Incubator-Projekt kann sehr viel genauer gesteuert werden, wie

der Aspekt verwoben werden soll. Dazu können die von AspectJ bereits bekannten *aop.xml*-Dateien verwenden werden. Es ist aber auch möglich, eine wie bei AJEER sehr globale Regel einzustellen.

Die OSGi Runtime Extension sorgt nun in beiden Projekten dafür, dass die so bekannt gemachten Aspekte beim Classloading mit den entsprechenden Klassen verwoben werden und dadurch entstehende Abhängigkeiten zwischen den Bundles korrekt behandelt werden. Damit ist man in der Lage, Aspekte innerhalb von Bundles modular zur Verfügung zu stellen und zur Laufzeit in das System weben zu lassen. Für das Beispiel verwenden wir der Einfachheit halber die AJEER-Variante.

AJEER verwenden

Die Installation von AJEER erfolgt am besten über die AJEER Update Site [8], wobei unbedingt darauf zu achten ist, dass das AJEER Feature, welches das Framework Extension Fragment enthält, in das Installationsverzeichnis des Eclipse SDK installiert wird und nicht in eine separate Extension Location. Nur wenn die Framework Extension im selben Verzeichnis wie das System Bundle (*org.eclipse.osgi*) liegt, kann sie gefunden und zum Framework hinzugefügt werden.

Das Plug-in *org.eclipse.ajeer.weavingruntime* definiert den Extension Point *aspects*, der dazu verwendet wird, Aspekte zu registrieren, die beim Load-Time Weaving berücksichtigt werden sollen (Listing 1).

Um der Eclipse Runtime die Verwendung von Framework Extensions bekannt zu machen, muss die System Property *osgi.framework.extensions* gesetzt und mit dem *Bundle-SymbolicName* des Framework Extension Bundle versehen werden. Weiterhin ist es nötig, das Plug-in *org.eclipse.ajeer.weavingruntime* zu starten. Beides wird in der Datei *config.ini* vorgenommen, die für jede Eclipse-Applikation existiert (Listing 2).

Zur Vereinfachung der Verwendung innerhalb des Eclipse SDK liefert das AJEER Tools Fragment einen AJEER Launcher, der auch diese Einstellungen in die automatisch generierte *config.ini* einträgt.

Design und Implementierung einer konkreten Lösung für unser Problem

Nun zurück zum eigentlichen Problem. Als Beispielapplikation dient eine mithilfe des Eclipse PDE Template *RCP application with a view* erzeugte Miniapplikation, die um ein ActionSet inklusive Beispiel Action sowie eine View Action erweitert wurde. Diese Applikation soll nun um die notwendigen Sicherheitsfeatures erweitert werden.

Den Zugriff auf eine Benutzerdatenbank inklusive Berechtigungsschema wird mithilfe der Klassen *Authorizer* und *AuthorizationDAO* sowie einer Properties-Datei (*authorizations.txt*) simuliert, welche die Berechtigungen enthält.

Wie bereits beschrieben, sollen Actions möglichst nur dann als aktiviert dargestellt werden, wenn der Benutzer berechtigt ist, sie auszuführen. Ziel ist es also, den *enablement* State aller Actions zu steuern. Ein Blick in das API offenbart folgende Signatur für *IAction*s: *public boolean isEnabled()*. Um alle Aufrufe dieser Methode abfangen und entsprechend der Benutzerberechtigung reagieren zu können, wird ein AspectJ Pointcut benötigt, der alle Aufrufe einer Methode *isEnabled* mit einem Boole'schen Parameter auf Subklassen des Interface *IAction* markiert (Listing 3). Die Überprüfung der Benutzerberechtigung wird dann mit einem recht einfachen Advice (Listing 4) durchgeführt.

Sowohl der Aspekt als auch die Hilfsklassen *Authorizer* und *AuthorizationDAO* werden in einem separaten Plug-in-Projekt mit dem Namen *de.eclipsemagazin.aspects.demo.security* abgelegt. Mittels des Menüeintrags ASPECTJ TOOLS | CONVERT TO ASPECTJ PROJECT wird das Projekt in ein AspectJ-Projekt umgewandelt. Im Plug-in-Manifest müssen die Plug-ins *org.eclipse.ajeer.weavingruntime* und *org.aspectj.runtime* als Dependencies hinzugefügt werden.

Zum Starten der Beispielapplikation mit aktivierter Sicherheit muss im Launch-Dialog eine neue AspectJ Launch-Konfiguration angelegt werden. Die auszuführenden Plug-ins sind *de.eclipsemagazin.aspects.demo3* und *de.eclipsemagazin.aspects.demo3.security*, die erforderlichen Abhängigkeiten können durch Klicken auf ADD REQUIRED PLUG-INS hinzugefügt werden. Auf der Seite *Main* muss als zu startende Applikation *de.eclipsemagazin.aspects.demo3.aosapplication* ausgewählt werden.

Wenn alles richtig läuft, lässt sich die Anwendung starten, und die Berechtigung lässt sich für die Actions über die Properties-Datei steuern. Das Format für die Berechtigungen lautet *action.[Benutzername.]Klassenname der Action=enabled*, also z.B. *action.Peter.org.eclipse.ui.internal.QuitAction=true* oder *action.de. eclipsemagazin.aspects.demo.SampleAction=false*.

Ein kurzer Test offenbart, dass die Actions zwar aktiviert bzw. deaktiviert werden, allerdings ist es nicht möglich, im laufenden Betrieb die Autorisierung für eine Action zu ändern. Dies liegt daran, dass Eclipse den *enablement* State einer Action im Normalfall nur einmal prüft. Eine Ausnahme bilden die View Actions: Ihr *enablement* State wird überprüft, sofern die Auswahl im zugehörigen View geändert wurde. Allerdings wird diese Prüfung nicht direkt im Anschluss an den Selektionswechsel ausgeführt, sondern erst direkt bevor die Action aufgerufen wird – und dazwischen können Minuten liegen!

Listing 4

```
boolean around(IAction action):
enablement(action) {
  String function = action.getClass().getName();

  System.out.println("Getting enablement state for ["
                                   + function + "]");
  boolean enabled = isAuthorized(function);
  if (enabled) {
    System.out.println("The user is authorized to
                                execute [" + function +
    "], but maybe the action is diabled nevertheless?");
    enabled = proceed(action);
  }
  System.out.println("Enablement state: [" + enabled
                                           + "].");
  return enabled;
}
```

Listing 5

```
pointcut actionDelegateInvocation(IActionDelegate
                             delegate, IAction action):
  target(delegate) && args(action) &&
  execution(void IActionDelegate.run(IAction));
```

Listing 6

```
pointcut actionInvocation(IAction action):
  target(action) && execution(void IAction.run());
```

Da diese erste Herangehensweise (Actions aktivieren/deaktivieren je nach Berechtigung) nicht wie gewünscht funktioniert, hat die Stunde für den alternativen Lösungsansatz (Ausführung der Actions abfangen) geschlagen: Wie bereits angemerkt, müssen dazu sowohl alle Aufrufe der Methode *public void IAction.run()* als auch der Methode *public void IActionDelegate.run(Action action)* abgefangen werden. Die dazu benötigten Pointcuts finden sich in Listing 5 bzw. 6.

Die Überprüfung der Benutzerberechtigung wird in den zugehörigen Advices durchgeführt (Listing 7 und 8).

Ein nochmaliges Ausführen der Applikation zeigt, dass die Actions nun nicht mehr disabled werden. Verfügt der aktuelle Benutzer allerdings über die notwendigen Rechte, so wird eine Warnmeldung ausgegeben.

Diese Lösung ist noch nicht perfekt, zeigt aber gut, wie sich mit Aspekt-orientierter Programmierung und Load-Time Weaving für OSGi pragmatische und elegante Lösungen implementieren lassen.

Auswirkungen

Das Load-Time Weaving von Aspekten benötigt in nicht unerheblichem Umfang Speicher- und Prozessorressourcen. Dies resultiert daraus, dass das Aspect Weaving selbst eine nicht-triviale Aufgabe ist und der Bytecode der geladenen Klassen geparst und weitergehend analysiert werden muss. Die Lösung für diese Probleme liegt in einem geschickten Caching der bereits gewobenen Klassen für den nächsten Start des Systems. Das Equinox-Incubator-Projekt hat dafür bereits eine Caching-Implementierung, die sich spezielle Eigenschaften der IBM Java VM zu Nutze macht und sehr effizient ist. Die Performance und der Speicherverbrauch einer bereits gecachten Anwendung lassen sich praktisch nicht mehr von einer Anwendung ohne Load-Time Weaving unterscheiden. Eine Implementierung für VMs, die nicht von IBM stammen, steht noch aus, sollte aber in Zukunft verfügbar sein. AJEER implementiert einen nicht so effizienten Cache, kann dafür aber auch auf diesen VMs eingesetzt werden.

Fazit

Die Ausführungen und Beispiele in diesem Artikel zeigen, dass durch Framework-Erweiterungen AspectJ Load-Time Weaving auch für Eclipse-RCP-Applikationen möglich ist und dass dieser Ansatz hervorragend geeignet ist, um das Thema Security zu adressieren. Die vorgestellte Lösung wird derzeit in einem kommerziellen Projekt produktiv eingesetzt und erlaubt es auf eine elegante Art und Weise, echte Cross Cutting Concerns auch in einer OSGi-Welt zu modularisieren.

Peter Friese ist Senior-Software-Architekt bei Gentleware. Er ist spezialisiert auf Modell-getriebene Softwareentwicklung, Spring sowie Eclipse-Technologien. Peter ist Committer für FindBugs und openArchitectureWare. Kontakt: peter.friese@gentleware.com

Martin Lippert ist Senior-IT-Berater bei it-agile. Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und Eclipse-Technologie und ist Committer im Eclipse Equinox Incubator-Projekt. Kontakt: martin.lippert@it-agile.de

Heiko Seeberger leitet die Market Unit Enterprise Architecture der metafinanz GmbH. Er erstellt seit etwa zehn Jahren Enterprise Applications mit Java, wobei sein aktueller Fokus auf Eclipse und AspectJ liegt. Heiko ist Committer von AJEER und ContractJ. Kontakt: heiko.seeberger@metafinanz.de

>>Links & Literatur

[1] www.ji.co.za/unplugged
[2] www.eclipsecon.org/2005/presentations/EclipseCon2005_7.1Scalability.pdf
[3] www.eclipse.org/aspectj
[4] www.eclipse.org/ajdt
[5] www.osgi.org
[6] wiki.eclipse.org/index.php/Adaptor_Hooks
[7] ajeer.sourceforge.net
[8] ajeer.sourceforge.net/updatesite

Listing 7

```
void around(IActionDelegate delegate, IAction action):
  actionDelegateInvocation(delegate, action) {

  String function = delegate.getClass().getName();
  System.out.println("Action [" + function + "] invoked on delegate ["
      + delegate.getClass().getName() + "]");

  System.out.println("Getting enablement state for [" + function + "]");
  boolean enabled = isAuthorized(function);
  if (enabled) {
    System.out.println("The user is authorized to execute [" +
      function + "], but maybe the action is diabled nevertheless?");
    proceed(delegate, action);
  }
  else {
    MessageDialog.openInformation(null, "Authorization failed",
      "You are not authorized to invoke this function.");
  }
}
```

Listing 8

```
void around(IAction action): actionInvocation(action) {
  String function = action.getClass().getName();
  System.out.println("Action [" + function + "] invoked.");
  System.out.println("Getting enablement state for [" + function + "]");
  boolean enabled = isAuthorized(function);
  if (enabled) {
    System.out.println("The user is authorized to execute [" +
     function + "], but maybe the action is diabled nevertheless?");
    proceed(action);
  }
  else {
    MessageDialog.openInformation(null, "Authorization failed",
     "You are not authorized to invoke this function.");
  }
}
```