Orientation in Objects

Acegi

Method based Security für Spring

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Soenke Sothmann
Orientation in Objects GmbH

Veröffentlicht am: 1.11.2006

ACEGI SECURITY

"Wer die Freiheit aufgibt, um Sicherheit zu gewinnen, wird am Ende beides verlieren." (Benjamin Franklin)

Sicherheit ist ein zentrales Thema in der Entwicklung vieler Softwaresysteme. Moderne Softwareentwicklung mit Java zeichnet sich dadurch aus, dass Technologie- und Infrastrukturcode weitgehend von der Business-Logik separiert werden. Das Domain orientierte Design kann somit im Vordergrund stehen und die Software trotzdem um technische Aspekte erweiterbar bleiben. Vorreiter und treibende Kraft auf diesem Gebiet ist das Spring Framework, das mit dem Acegi Security Framework nun über eine Sicherheitslösung verfügt, die diesen Prinzipien folgt.

Durch den Einsatz des Acegi Security Frameworks kann eine Anwendung entwickelt werden, ohne Sicherheitsaspekte von Anfang an berücksichtigen zu müssen. Die Sicherheit wird anschließend hinzukonfiguriert. Somit bleibt die Anwendung unabhängig von den verwendeten Sicherheitsmechanismen und -technologien.

Dieser Artikel gibt eine Einführung in das Acegi Framework und beleuchtet dessen Architektur, wobei der Fokus auf der Absicherung auf Methodenebene (z.B. bei Rich Client und Backend Systemen) liegt. Anhand eines Beispiels wird der Einsatz des Frameworks erläutert.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

EINFÜHRUNG ACEGI

Nachdem das Spring Framework sich immer größerer Beliebtheit in der Entwickler-Gemeinschaft erfreute, kam der Wunsch nach einer Sicherheitslösung auf. Dieser Wunsch wurde von einigen Entwicklern der Community aufgenommen und es entstand ein eigenes Spring Unterprojekt. Acegi ist, wie Spring selbst, ein Open-Source Projekt unter der Apache Lizenz. Längst bewährt in Projekten, ist Acegi im Mai 2006 in der stabilen Version 1.0 erschienen. Die Vorteile einer Open-Source Sicherheitslösung liegen auf der Hand:

- breite Unterstützung durch die Community
- stetige Qualitätskontrolle und Prüfung auf Sicherheitslücken des Quellcodes
- Quellcode liefert Vorlage für eigene, unternehmensspezifische Implementierungen

KEY FEATURES

Zu den wichtigsten Features von Acegi zählen:

- Unterstützung für Webanwendungen (URL-Based Security) durch Verwendung von Servlet Filtern
- Unterstützung von Sicherheit auf Methodenebene durch den Einsatz von Interceptoren (Aspektorientierung)
- viele Technologien für die Authentifikation integriert
- Anwendung bleibt frei von Security-Code
- alle Teile des Frameworks sind austausch- und erweiterbar

DER GEIST VON ACEGI: DAS "SPRING-PRINZIP"

Ein Grundgedanke des Spring Framework ist es, durch Einsatz von Inversion of Control und mit mitteln der Aspektorientierung möglicherweise orthogonale Aspekte wie Business Logik und Technologie Code zu trennen. Man erhält so flexible Software, bei der Technologien leicht durch andere ausgetauscht werden können.

Acegi bleibt diesem Prinzip treu und ermöglicht es, bei der Entwicklung einer Anwendung Sicherheitsaspekte von der übrigen Anwendung zu trennen. Acegi erweitert eine bestehende Anwendung mit seinen fertigen Komponenten durch Konfiguration. Als Beispiel sei hier eine Anwendung genannt, die vorerst eine Authentifikation gegen eine Datenbank verwendet, später aber gegen ein LDAP System authentifiziert – lediglich durch Änderungen an der Konfigurationsdatei. Viele Komponenten kann man leicht durch eigene spezielle Implementierungen austauschen.

EINSATZ DER ASPEKTORIENTIERUNG

Spring bietet einen Ansatz für die aspektorientierte Programmierung (AOP). Beliebige Objekte können um Funktionalität erweitert werden, ohne den Code der zugehörigen Klasse zu verändern. Wie in Abb. 1 dargestellt, wird dies möglich, indem nicht mehr auf das eigentliche Objekt zugegriffen, sonder mit einem Stellvertreter (Proxy) kommuniziert wird. Dieser kann zur Laufzeit erzeugt werden und ist für den Aufrufer identisch aufgebaut wie das Zielobjekt. Dieser Proxy delegiert beim Aufruf einer Methode an das Originalobjekt weiter, hat jedoch die Möglichkeit, vor und/oder nach dem Ausführen der Methode noch andere Objekte zwischenzuschalten, die zusätzliche Funktionalität bieten. Man spricht hierbei von sog. Interceptoren.

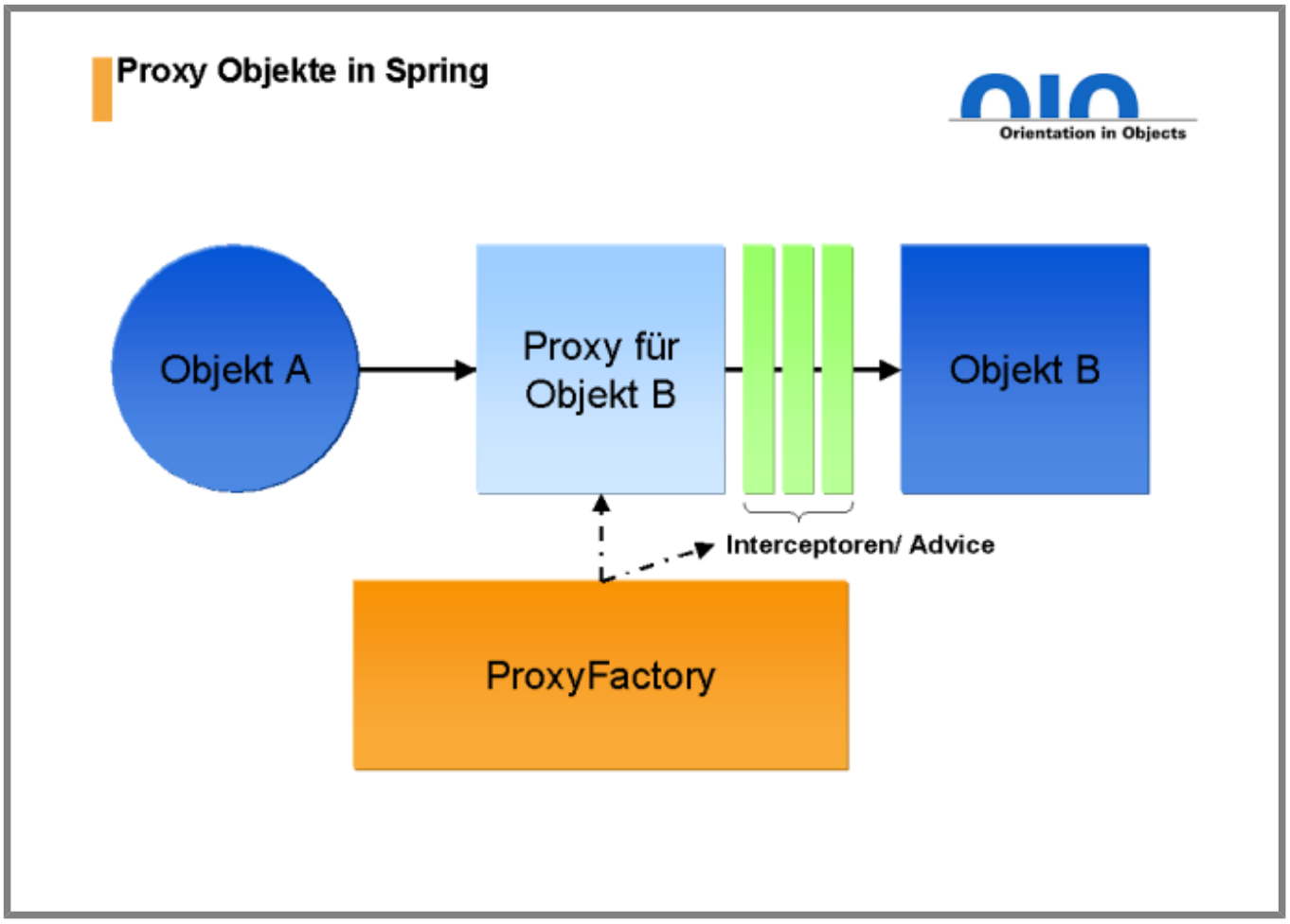

Abbildung 1: Proxy Objekte in Spring

Acegis Method Based Access Control setzt auf Spring AOP auf. Dabei wird den zu schützenden Objekten ein Proxy vorgeschaltet, der die Kontrolle an den sog. SecurityInterceptor übergibt. Dieser kann nun die Zugriffsberechtigung des Benutzers prüfen und ihm den Zugriff gestatten oder verweigern. Die geschützten Klassen und Methoden wissen dabei nichts von einem Sicherheitsmechanismus.

ACEGI OHNE SPRING?

Obwohl Acegi speziell für die Verwendung mit dem Spring Framework entwickelt wurde, kann Acegi auch in Anwendungen ohne Spring verwendet werden. Hierfür muss allerdings Infrastrukturcode und ein Konfigurationsmechanismus selbst entwickelt werden - in einem solchen Fall ist zu überlegen, ob Acegi hier wirklich die passende Lösung darstellt, oder nicht doch andere Produkte geeigneter sind.

TECHNOLOGIE-UNTERSTÜTZUNG

Acegi verwendet zur Authentifikation einen oder mehrere sog. AuthenticationProvider. Mitgeliefert werden zahlreiche Provider, die die Authentifikation entweder selbst übernehmen oder auf bestehende Authentifikationssysteme zurückgreifen. Die wichtigsten Provider für Method based Security sind folgende:

- In-Memory Authentication

  Hierbei werden die Benutzerinformationen (Benutzername und Passwort) direkt im Speicher abgelegt. Dieser Provider wird oft während der Entwicklung einer Anwendung und in JUnit-Tests verwendet, um nicht auf eine externe Datenquelle wie eine Datenbank angewiesen zu sein.

- JDBC Authentication

  Die Benutzerinformationen werden aus einer beliebigen (SQL-)Datenbank bezogen. Die hierbei verwendeten SQL-Queries können frei konfiguriert werden.

- JAAS Authentication

  Der Java Authentication and Authorization Service ist seit Version 1.4 offizieller Bestandteil des Java 2 SDKs. Acegi liefert einen AuthenticationProvider für JAAS mit, der die Authentifikation erledigt. Die Autorisierung erfolgt jedoch nicht über JAAS, sondern über Acegi selbst.

- LDAP Authentication

  Über LDAP können personenbezogene Daten von einem Verzeichnisdienst abgefragt werden, einschließlich den für die Authentifikation relevanten Daten Benutzername, Passwort und Rollen. Acegi's LDAP Authentication Provider ist voll konfigurierbar und kann verschiedene Strategien verwenden, um an die relevanten Daten aus dem Verzeichnisdienst zu gelangen.

- CAS Authentication (Central Authentication Service)

  CAS ist eine OpenSource Single Sign-On Lösung der JA-SIG Organisation, die eine hohe Verbreitung hat. Der entsprechende Provider des Acegi Frameworks bietet Unterstützung für CAS 2.0 und 3.0.

Des weiteren gibt es einige Provider, die speziell für Webanwendungen von Bedeutung sind. Eine Übersicht dieser Provider finden Sie im offiziellen Acegi Reference Guide.

ACEGI ARCHITEKTUR

Acegi verrichtet seine Arbeit mit einer Hand voll Klassen und Interfaces. Die Bedeutung dieser wenigen Klassen bzw. Interfaces und ihr Zusammenspiel zu verstehen ist der Schlüssel zum Verständnis von Acegi. Will man unternehmensspezifische Anforderungen umsetzten, für die Acegi keine eigene Lösung mitbringt, ist es wichtig, die Bedeutung der Komponenten zu kennen, um so die richtigen Ansatzpunkte zu finden. Im folgenden wird die Architektur Acegis mit dem Blickwinkel auf Method based Security vorgestellt.

ÜBERSICHT DER ZENTRALEN KLASSEN UND INTERFACES

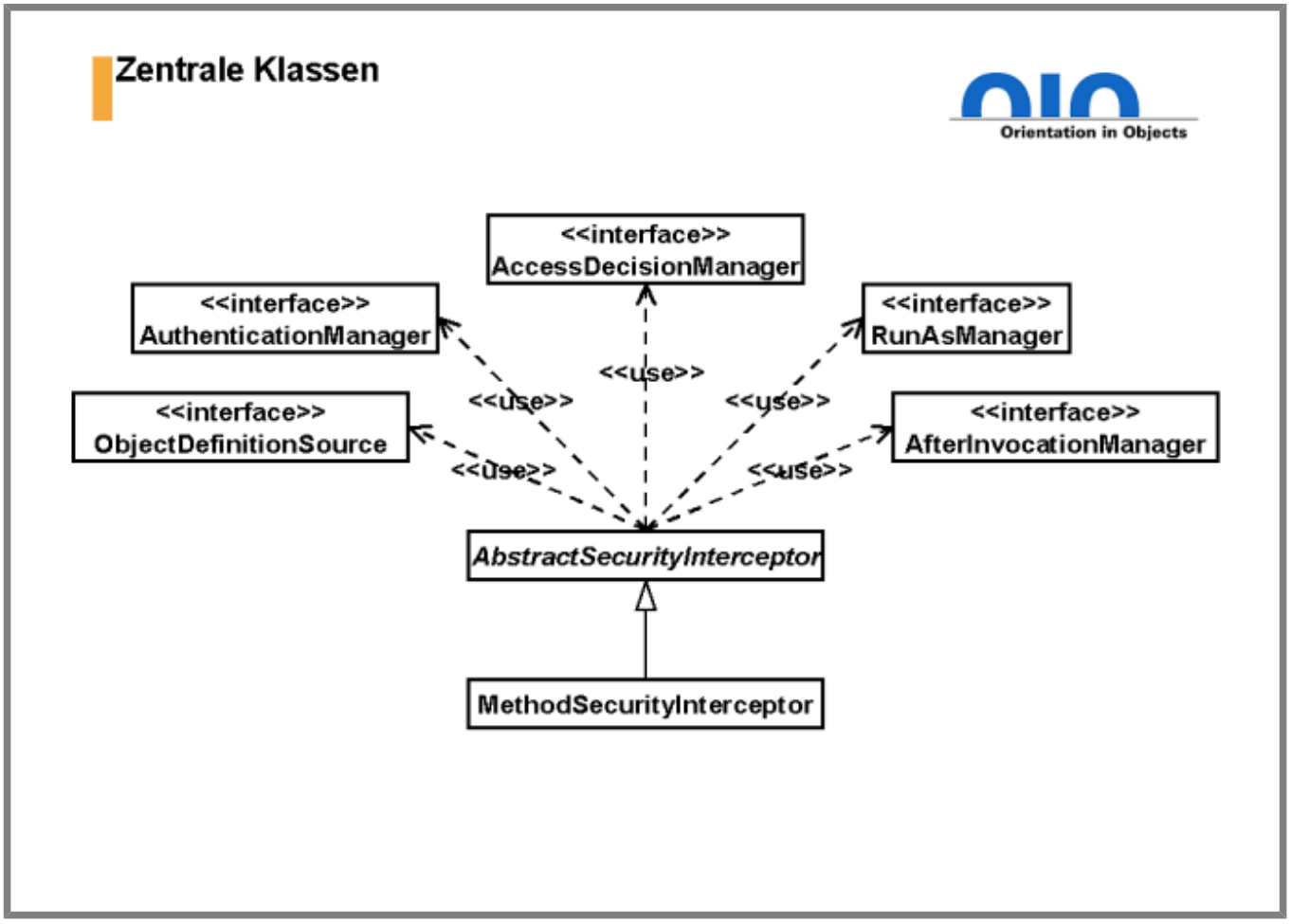

Abbildung 2: Übersicht der zentralen Klassen und Interfaces

ABSTRACTSECURITYINTERCEPTOR

Dies ist die zentrale Klasse des Acegi Frameworks. Der Interceptor wird immer dann aktiv, wenn auf eine geschützte Ressource zugegriffen wird. Die eigentliche Arbeit des Frameworks erledigen jedoch andere Klassen, mit denen dieser Interceptor nur über Interfaces kommuniziert. So wird es möglich, verschiedene Implementierungen zu verwenden und beliebig auszutauschen. Von diesem abstrakten Interceptor erben einige konkrete Implementierungen, darunter auch der MethodSecurityInterceptor – ein Interceptor für Method based Security mit Spring AOP.

OBJECTDEFINITIONSOURCE

Speichert, für welche geschützten Ressourcen (Methoden) man welche Berechtigungen (z.B. Rollen) besitzen muss.

AUTHENTICATIONMANAGER

Der AuthenticationManager ist dafür zuständig, Authentifikationsanforderungen anhand eines Authentication-Objektes zu prüfen. Wie in Abb. 3 dargestellt, verwendet er für diese Aufgabe einen oder mehrere sog. AuthenticationProvider, die beispielsweise auf eine Datenbank zugreifen. Außerdem setzt der AuthenticationManager die dem Benutzer gewährten Berechtigungen in das Authentication-Objekt – man spricht hierbei von sog. "Granted Authorities".

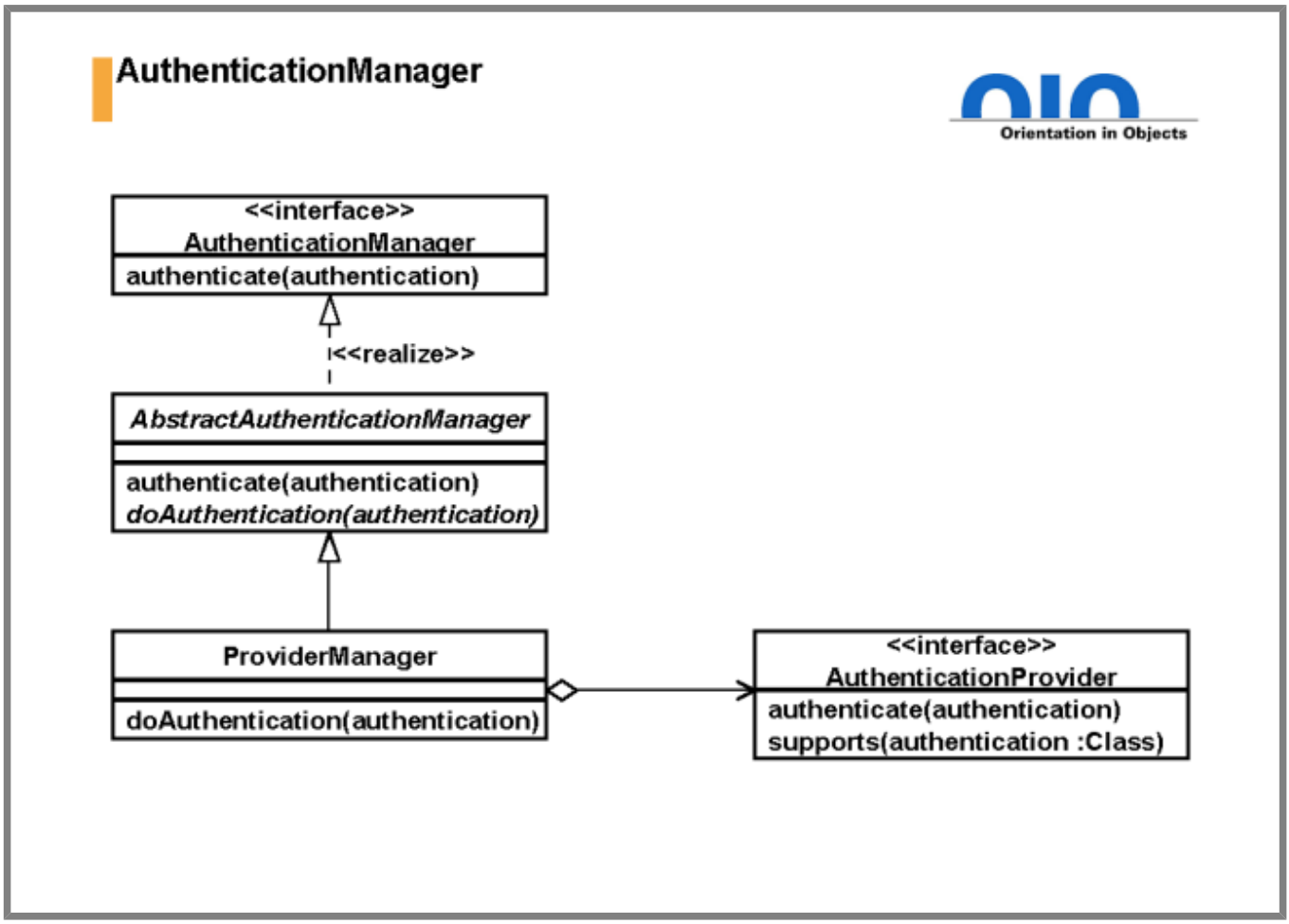

Abbildung 3: AuthenticationManager

ACCESSDECISIONMANAGER

Dieser Manager hat die Aufgabe zu entscheiden, ob ein Benutzer autorisiert ist, auf eine geschützte Ressource zuzugreifen. Um eine Entscheidung zu treffen, befragt er, wie in Abb. 4 dargestellt, einen oder mehrere Voter, die – anhand der angeforderten Ressource und den Berechtigungen des Benutzers – dafür oder dagegen stimmen. Die verwendete Strategie, ob eine Für-Stimme ausreicht oder alle Voter dafür stimmen müssen, hängt von der verwendeten Implementierung des AccessDecisionManagers ab.

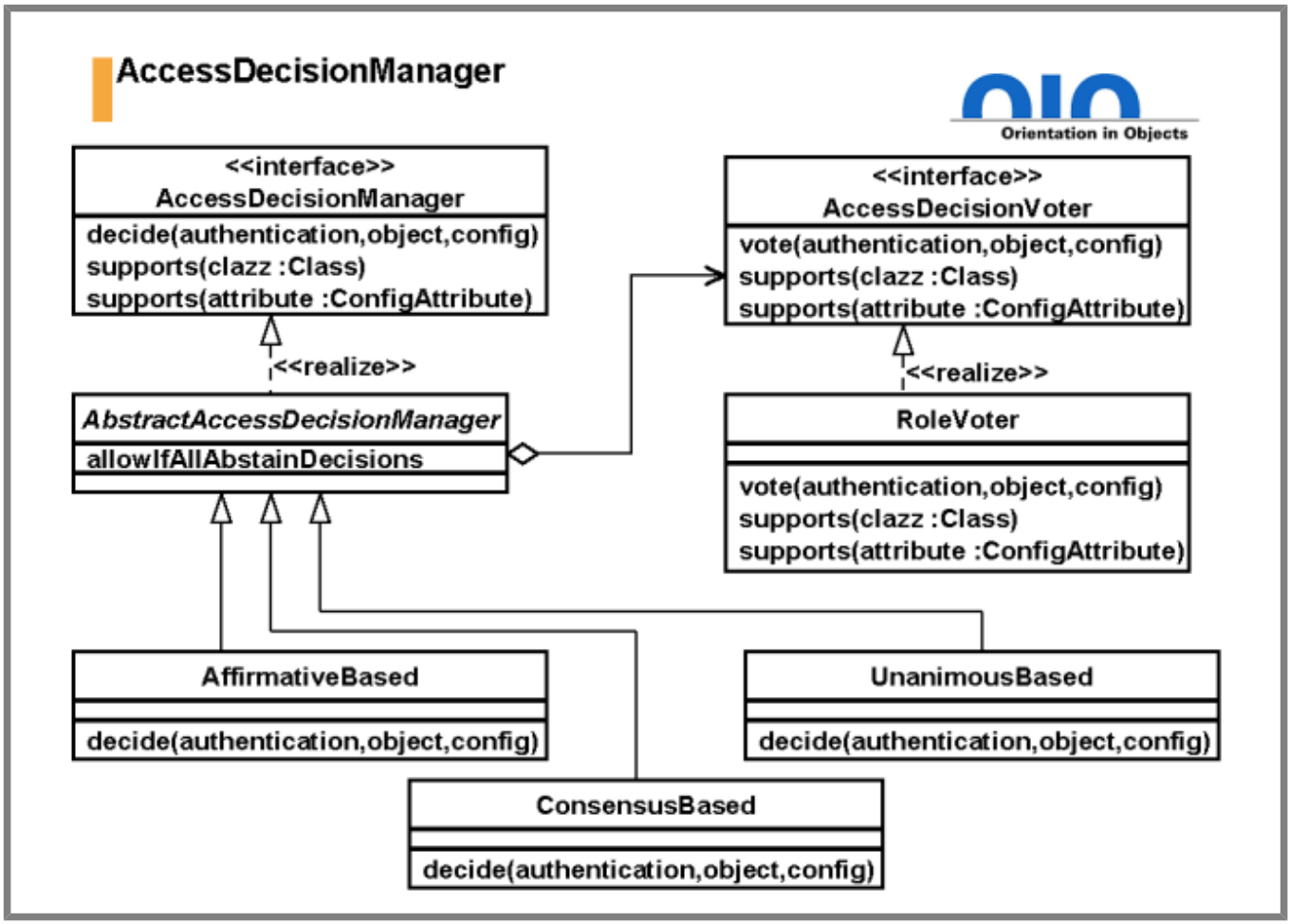

Abbildung 4: AccessDecisionManager

RUNASMANAGER

Durch den RunAsManager wird es ermöglicht, für die Dauer eines Aufrufs einer geschützten Methode die Berechtigungen eines Benutzers zu ändern. So wird es möglich, dass die geschützte Methode weitere geschützte Methoden aufruft, für die der Benutzer keine Berechtigung besitzt.

AFTERINVOCATIONMANAGER

Wird dem SecurityInterceptor ein AfterInvocationManager zugeteilt, so erhält dieser nach jedem erfolgten Aufruf einer geschützten Methode die Möglichkeit, das zurückgelieferte Objekt zu modifizieren oder die Rückgabe komplett zu unterbinden.

WEITERE ZENTRALE KLASSEN

SECURITYCONTEXT

Der SecurityContext enthält das Authentication-Objekt und stellt die aktuelle Sitzung eines Benutzers dar. Er ist anwendungsweit verfügbar über die statische Methode SecurityContextHolder.getContext().

AUTHENTICATION

Ein Authentication-Objekt enthält die Benutzerinformationen wie Benutzername und Passwort (je nach Implementierung) und die "Granted Authorities". Es kann die Zustände authentifiziert und nicht authentifiziert annehmen.

LOGIN ABLAUF

Vor dem Aufruf geschützter Methoden muss eine Authentifikation durchgeführt werden. Deshalb wird hier zunächst der Grundgedanke des Login-Vorgangs in Acegi vorgestellt. Das Login ist oft der einzige Vorgang, für den in der Anwendung Security Quellcode geschrieben werden muss. Dieser beschränkt sich jedoch oft auf lediglich 2 bis 3 Zeilen Code. Das Login läuft wie folgt ab:

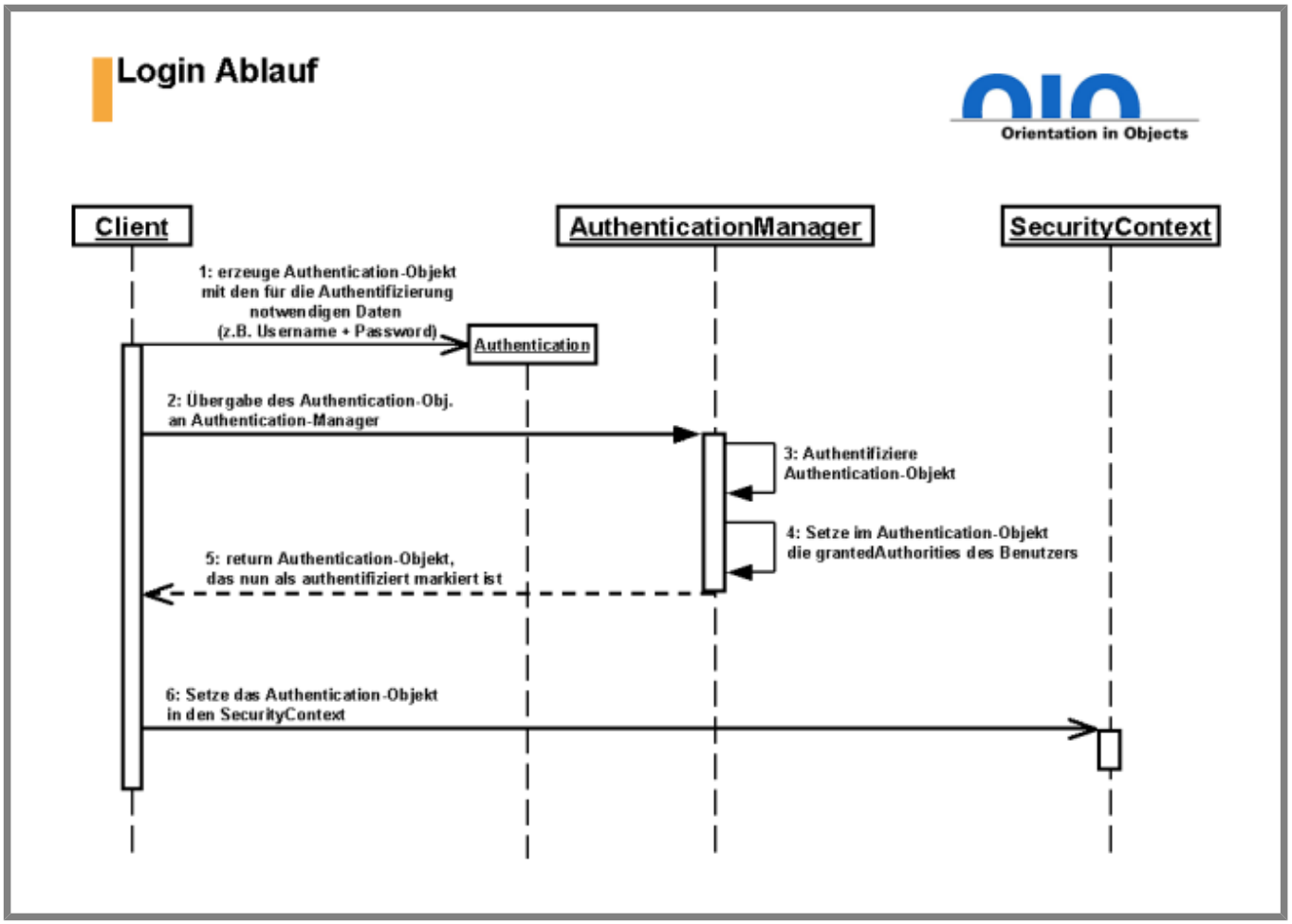

Abbildung 5: Login-Ablauf

Es wird ein Authentication-Objekt erzeugt, dem man die für die Authentifikation notwendigen Daten setzt (1). Im Falle der UsernamePasswordAuthenticationToken Implementierung sind dies Benutzername und Passwort. Dieses Authentication-Objekt übergibt man dem AuthenticationManager (2), der die Authentifikation mit Hilfe eines oder mehrerer AuthenticationProvider durchführt (3). Wurden die Anmeldedaten erfolgreich verifiziert, so werden die sog. „grantedAuthorities“ gesetzt (4) und der Manager liefert ein Authentication-Objekt zurück (5), das er als erfolgreich authentifiziert markiert hat. Dieses Objekt wird nun in den SecurityContext gesetzt (6) - der Benutzer gilt von nun an als authentifiziert.

AUFRUF EINER GESCHÜTZTEN METHODE

Im folgenden wird der eigentlichen Bereich der Method based Security erläutert. Wie Abb. 6 zeigt, wird beim Aufruf einer geschützten Methode zuerst geprüft, ob der Benutzer überhaupt angemeldet ist.

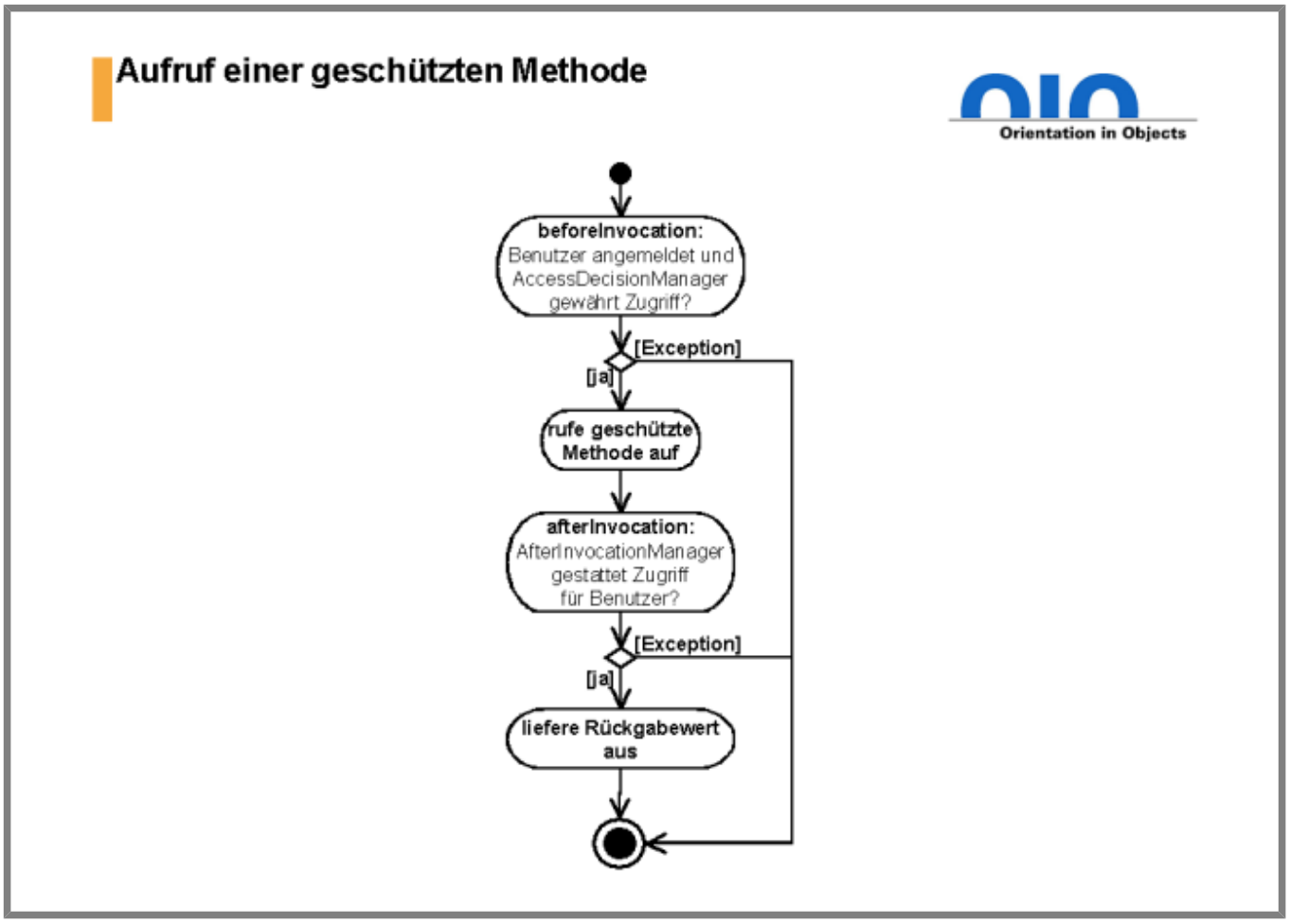

Abbildung 6: Aufruf einer geschützten Methode

Ist dies nicht der Fall, so wird geprüft, ob eine Authentifikationsanfrage vorliegt (es wird geprüft, ob im SecurityContext ein Authentication-Objekt vorhanden ist, das noch nicht authentifiziert wurde). In einem solchen Fall führt der AuthenticationManager die Authentifikation durch und setzt im Authentication-Objekt die Berechtigungen, die dem Benutzer gewährt werden. Liegt keine Authentifikationsanfrage vor oder ist die Authentifikation gescheitert, so wird mit einer Exception der Zugriff auf die Methode verweigert. Anschließend wird festgestellt, ob der Benutzer die nötigen Berechtigungen für den Zugriff auf diese Methode hat. Hierfür wird der AccessDecisionManager zu Rate gezogen, der die Entscheidung mit Hilfe von Votern treffen kann, die dafür oder dagegen stimmen können. Kommt der AccessDecisionManager zu dem Schluß, dass der Benutzer berechtigt ist, so erhält der RunAsManager (falls ein solcher konfiguriert wurde) die Möglichkeit, das Authentication-Objekt im Security-Context für die Dauer des Methodenaufrufs durch ein anderes zu ersetzten. Dadurch wird es möglich, dass eine geschützte Methode andere geschützte Methoden aufruft, für die der Benutzer selbst keine Rechte besitzt. Kommt der AccessDecisionManager jedoch zu dem Schluß, dass der Benutzer nicht über die nötigen Berechtigungen verfügt, so verweigert er den Zugriff durch das Werfen einer Exception. Schließlich wird die geschützte Methode ausgeführt. Der hier beschriebene Vorgang der "beforeInvocation" ist in Abb. 7 detailliert dargestellt.

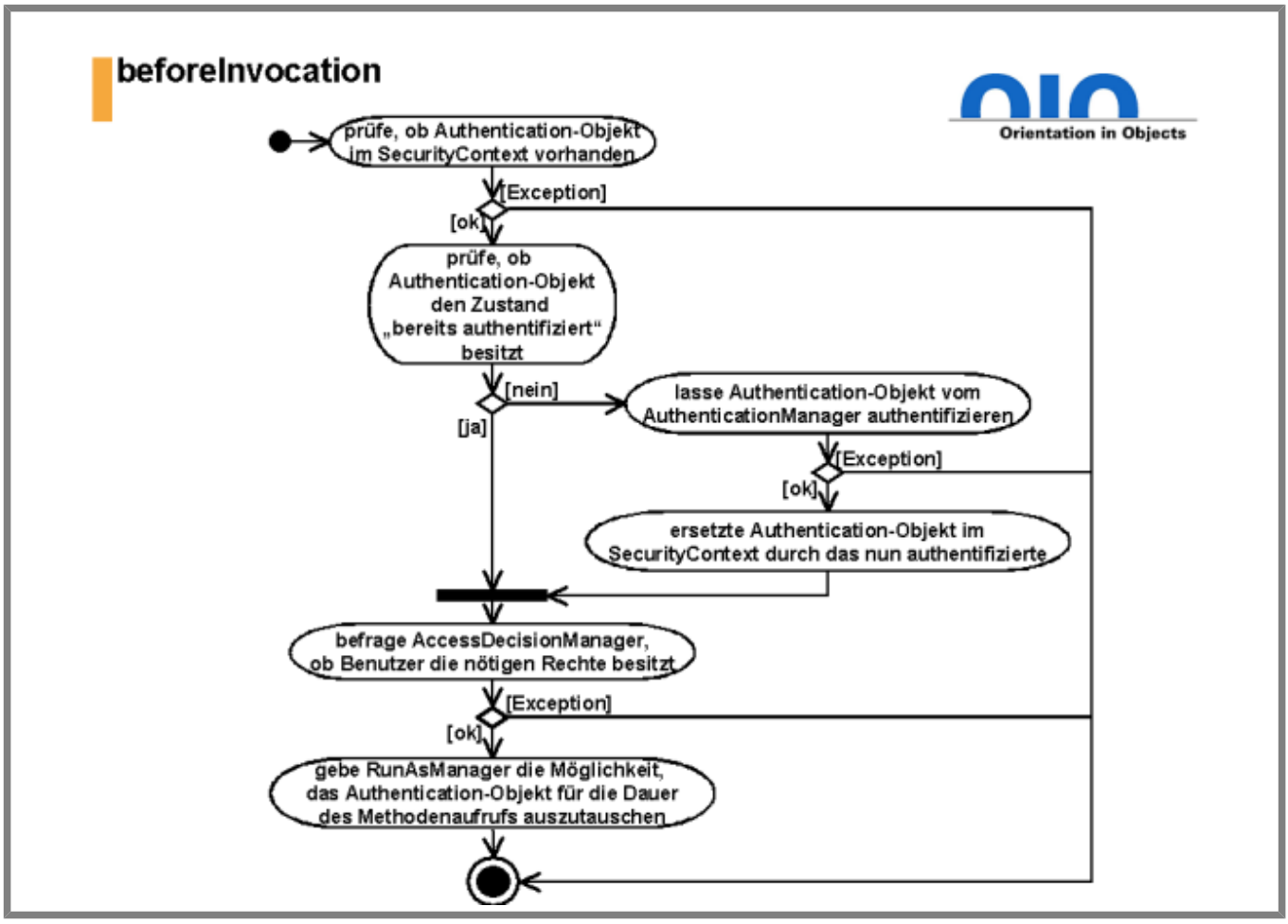

Abbildung 7: beforeInvocation

Nachdem die geschützte Methode ausgeführt wurde, beginnt die sog. „afterInvocation“, die in Abb. 8 dargestellt ist. Zu Beginn dieser Phase wird ggf. das ursprüngliche Authentication-Objekt wiederhergestellt, falls der RunAsManager aktiv war. Somit wird sichergestellt, dass die nun anschließende Behandlung durch den AfterInvocationManager im Rahmen der Sicherheit der ursprünglichen Autorisierung stattfindet. Dieser AfterInvocationManager wird nun aktiv, falls ein solcher konfiguriert wurde. Er hat die Möglichkeit, das Objekt, das von der geschützten Methode zurückgeliefert wird, zu inspizieren und noch einmal eine Entscheidung zu treffen, ob der Benutzer tatsächlich das Objekt erhalten darf. Er hat auch die Möglichkeit, Änderungen an diesem Objekt vorzunehmen, wie z.B. kritische Felder zu löschen, die nur bestimmten Benutzern zugänglich gemacht werden sollen. Zu guter letzt wird dem Benutzer das Objekt zurückgeliefert.

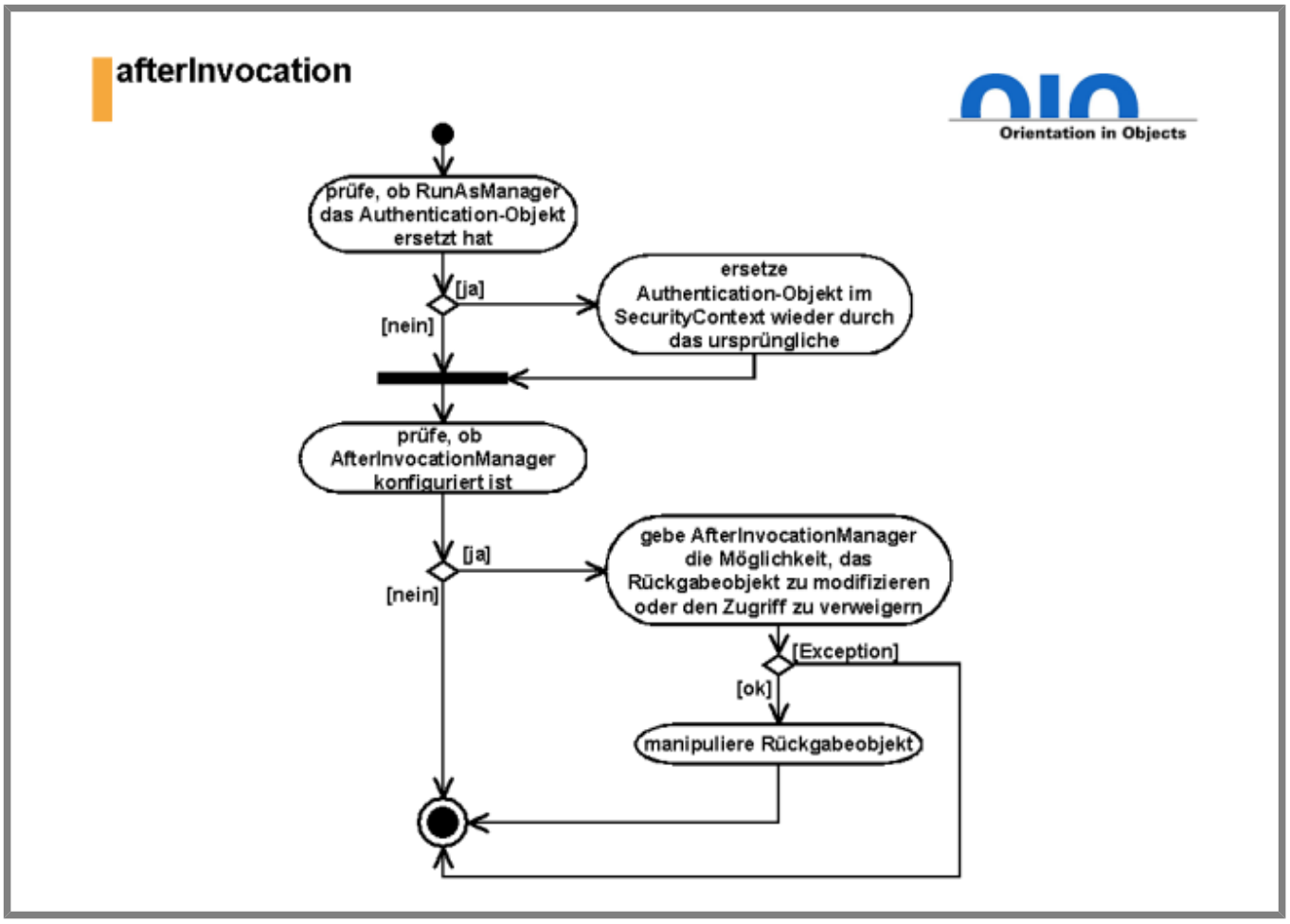

Abbildung 8: afterInvocation

BEISPIEL RECHNUNGSVERWALTUNG

Als Beispielanwendung für den Einsatz von Acegi dient folgendes Szenario:

SZENARIO

Eine Anwendung soll Rechnungen verwalten. Ein sog. InvoiceManager ermöglicht es, bestehende Rechnungen abzurufen oder neue Rechnungen zu erzeugen. Diese Komponente soll nun so abgesichert werden, dass nur Benutzer, die eine Buchhalterrolle inne haben (ROLE_ACCOUNTANT) beliebige Rechnungen erzeugen und abrufen können. Benutzer mit einer Kundenrolle (ROLE_CUSTOMER) sollen Rechnungen abrufen können - allerdings nur ihre eigenen.

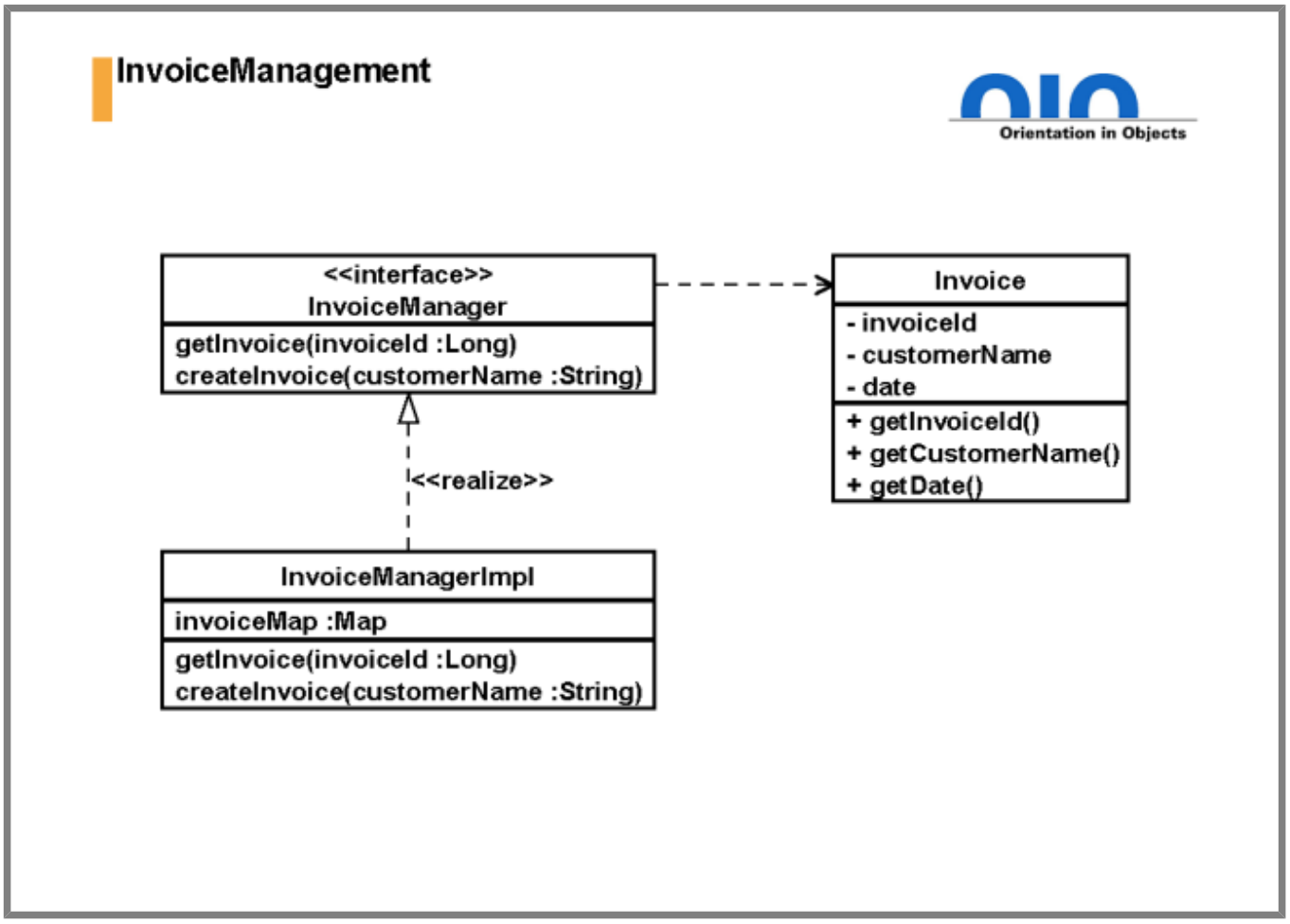

Abbildung 9: Beispielanwendung "InvoiceManagement"

IDEEN FÜR DIE UMSETZUNG

Die InvoiceManager-Klasse muss durch einen SecurityInterceptor geschützt werden. Die Methode zum Erstellen von Rechnungen (createInvoice) soll derart eingeschränkt werden, dass ein Zugriff nur mit der Buchhalter-Rolle ROLE_ACCOUNTANT möglich ist. Der Zugriff auf die Methode zum Abrufen von Rechnungen (getInvoice) muss sowohl mit der ROLE_ACCOUNTANT als auch mit der ROLE_CUSTOMER möglich sein. Es ist nun aber Kunden möglich, fremde Rechnungen abzurufen – das würde kein Datenschutzbeauftragter durchgehen lassen. Tatsächlich kann erst festgestellt werden, ob der Kunde eine Rechnung abrufen darf, nachdem die Rechnung aus einer Datenquelle wie einer Datenbank ausgelesen wurde – sprich nachdem die getInvoice() Methode des InvoiceManagers ausgeführt wurde und eine Rechnungsinstanz im Speicher existiert. Es muss ein AfterInvocationProvider implementiert werden, der bei einem zurückgeliefert Rechnungsobjekt und der ROLE_CUSTOMER prüft, ob die Rechnung auch tatsächlich zu dem Benutzer gehört, der die Anfrage stellt.

IMPLEMENTIERUNG

Die Implementierung wird in zwei Teile aufgespaltet: zuerst muss dafür gesorgt werden, dass die createInvoice-Methode nur von Buchhaltern und die getInvoice-Methode sowohl von Buchhaltern als auch von Kunden aufgerufen werden kann. Im zweiten Teil wird realisiert, dass Kunden nur ihre eigenen Rechnungen abrufen dürfen. Aber ein Schritt nach dem anderen ...

SCHRITT 1

Die Anwendung verfügt über eine Rechnung, die wie folgt aufgebaut ist:

```
public class Invoice {
  private Long invoiceId;
  private String customerName;
  private Date date;
  // getter und setter ...
}
```

Beispiel 1: Klasse Invoice

Desweiteren existiert ein InvoiceManager, der Methoden zum Laden und Erstellen von Rechnungen zur Verfügung stellt. Das Interface sieht wie folgt aus:

```
public interface InvoiceManager {
  public abstract Invoice getInvoice(Long invoiceId);
  public abstract Invoice createInvoice(String customerName);
}
```

Beispiel 2: InvoiceManager Interface

In der Spring XML-Konfigurationsdatei wird eine Implementierung des InvoiceManager-Interfaces als Bean aufgenommen. Da diese Bean durch Acegi geschützt werden soll, wird in der Konfigurationsdatei ein SecurityInterceptor wie folgt definiert:

```
<bean id="invoiceManagementSecurityInterceptor"
  class="org.acegisecurity.intercept.method.
                    aopalliance.MethodSecurityInterceptor">
  <property name="authenticationManager"
         ref="authenticationManager" />
  <property name="accessDecisionManager"
         ref="accessDecisionManager" />
  <property name="objectDefinitionSource">
    <value>
      de.oio.acegi.invoicemanagement.InvoiceManager.getInvoice*=
       ROLE_CUSTOMER,ROLE_ACCOUNTANT
      de.oio.acegi.invoicemanagement.InvoiceManager.createInvoice*=
       ROLE_ACCOUNTANT
    </value>
  </property>
</bean>
```

Beispiel 3: Konfiguration SecurityInterceptor

Die ObjectDefinitionSource wurde so konfiguriert, dass die getInvoice und die createInvoice Methoden nur noch von Benutzern mit den entsprechenden Rollen aufgerufen werden können. Der AuthenticationManager samt Provider und Datenquelle wird wie folgt definiert:

```
<bean id="authenticationManager"
  class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="authenticationProvider" />
    </list>
  </property>
</bean>
<bean id="authenticationProvider"
  class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="authenticationDao"/>
</bean>
<bean id="authenticationDao"
  class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      sothmann=mypass,ROLE_ACCOUNTANT
      schaefer=secret,ROLE_CUSTOMER
    </value>
  </property>
</bean>
```

Beispiel 4: Konfiguration AuthenticationManager

Der AuthenticationManager wird seine Arbeit mit Hilfe eines DaoAuthenticationProviders verrichten, der wiederum als Datenquelle einen InMemoryDao verwendet, bei dem die Benutzernamen, Passwörter und zugehörige Rollen direkt in der Konfigurationsdatei definiert werden.

Als nächstes muss ein AccessDecisionManager wie folgt konfiguriert werden:

```
<bean id="accessDecisionManager"
  class="org.acegisecurity.vote.AffirmativeBased">
  <property name="decisionVoters">
    <list>
      <bean class="org.acegisecurity.vote.RoleVoter">
        <property name="rolePrefix" value="ROLE_" />
      </bean>
    </list>
  </property>
</bean>
```

Beispiel 5: Konfiguration AccessDecisionManager

Dieser AccessDecisionManager verfolgt die Strategie, dass der Zugriff auf eine geschützte Methode gestattet wird, wenn mindestens ein Voter sein OK gibt. Hier wurde ein Voter konfiguriert, der mit Rollen umzugehen versteht und sich für alle Rollen zuständig fühlt, die mit ROLE_ beginnen.

Das letzte, was nun noch fehlt, ist den SecurityInterceptor der geschützten Klasse vorzuschalten. Dies wird bequem durch einen BeanNameAutoProxyCreator von Spring erledigt. Die Konfiguration hierfür lautet wie folgt:

```
<bean id="autoProxyCreator"
  class="org.springframework.aop.framework.
                        autoproxy.BeanNameAutoProxyCreator">
  <property name="interceptorNames">
    <list>
      <value>invoiceManagementSecurityInterceptor</value>
    </list>
  </property>
  <property name="beanNames">
    <list>
      <value>invoiceManager</value>
    </list>
  </property>
  <property name="proxyTargetClass" value="true" />
</bean>
```

Beispiel 6: Konfiguration AutoProxyCreator

Damit ist der erste Teil der Implementierung abgeschlossen. Wie man sehen kann, ist keine einzige Zeile Java-Quellcode nötig, Acegi wird einfach zur bestehenden Anwendung hinzukonfiguriert.

SCHRITT 2

Nun wird dem eingeschränkten Zugriff für Kunden Beachtung geschenkt. Es ist notwendig, einen AfterInvocationProvider zu schreiben, der die Objekte untersucht, die von der getInvoice-Methode zurückgeliefert werden. Stimmt das Feld customerName nicht mit dem Username des aufrufenden Benutzers überein, so wird der Zugriff mit einer Exception verweigert. Die Implementierung der dafür zuständigen decide-Methode lautet wie folgt:

```
public Object decide(Authentication authentication,
                Object invocation,
                ConfigAttributeDefinition config,
                Object returnedObject)
                throws AccessDeniedException {
  MethodInvocation methodInvocation = (MethodInvocation) invocation;
  if(methodInvocation.getMethod().getName().equals("getInvoice")
    && isNotAccountant(authentication)){
    Invoice invoice = (Invoice) returnedObject;
    String username =
              ((User)authentication.getPrincipal()).getUsername();
    if(!invoice.getCustomerName().equals( username )){
      throw new AccessDeniedException();
    }
  }
  return returnedObject;
}
```

Beispiel 7: Implementierung AfterInvocationProvider

Nun muss dieser AfterInvocationProvider samt einem passenden Manager noch der Konfigurationsdatei hinzufügt werden:

```
<bean id="afterInvocationManager"
  class="org.acegisecurity.afterinvocation.
                          AfterInvocationProviderManager">
  <property name="providers">
    <list>
      <ref local="invoiceAfterInvocationProvider" />
    </list>
  </property>
</bean>
<bean id="invoiceAfterInvocationProvider"
  class="de.oio.acegi.invoicemanagement.
                          InvoiceAfterInvocationProvider"/>
```

Beispiel 8: Konfiguration AfterInvocationManager

Im letzten Schritt wird der AfterInvocationManager in den SecurityInterceptor eingeklinkt. Dafür wird die Konfiguration folgendermaßen angepasst:

```
<bean id="invoiceManagementSecurityInterceptor"
  class="org.acegisecurity.intercept.method.
                      aopalliance.MethodSecurityInterceptor">
  <property name="authenticationManager"
         ref="authenticationManager" />
  <property name="accessDecisionManager"
         ref="accessDecisionManager" />
  <property name="afterInvocationManager"
         ref="afterInvocationManager" />
  <property name="objectDefinitionSource">
    <value>
      de.oio.acegi.invoicemanagement.InvoiceManager.getInvoice*=
        ROLE_CUSTOMER,ROLE_ACCOUNTANT
      de.oio.acegi.invoicemanagement.InvoiceManager.createInvoice*=
        ROLE_ACCOUNTANT
    </value>
  </property>
</bean>
```

Beispiel 9: AfterInvocationManager in SecurityInterceptor einklinken

Damit ist die Implementierung abgeschlossen und die Anwendung mit einem Access Control Mechanismus geschützt. Um die Funktionsweise zu gewährleisten, sollten JUnit-Tests geschrieben werden.

Das komplette Projekt zum Download

SICHERHEITSTESTS MIT JUNIT

Der Sicherheitsmechanismus einer Anwendung sollte unbedingt mit JUnit-Tests auf richtige Funktionsweise getestet werden, da hier Fehler mitunter fatale Folgen haben können. Der Einsatz des InMemoryDaos aus dem obigen Beispiel als Datenquelle für die Authentifikation ist ideal für Tests geeignet, da man kein externes System ansprechen muss und direkt Testaccounts definieren kann.

Ein Typischer JUnit-Test, der die Sicherheit einer Anwendung gewährleisten soll, würde für jede zu schützende Methode mindestens einen Testfall enthalten, in dem der Zugriff verschiedener Benutzer mit unterschiedlichen Rollen auf die geschützte Methode getestet wird. Man prüft, ob bei nicht autorisierten Benutzern eine Exception geworfen wird, andernfalls darf eine solche nicht auftreten.

Exemplarisch zwei Testfälle für die im oberen Beispiel genannte Methode createInvoice:

```
public void testAccountantCreateInvoice() {
  login("sothmann", "mypass");
  assertNotNull(invoiceManager.createInvoice("testcustomer"));
}
public void testCustomerCreateInvoice() {
  login("schaefer", "secret");
  try{
    invoiceManager.createInvoice("testcustomer");
    fail("customer shouldn't get access to the createInvoice method");
  } catch (Exception e) {
    // test successful
  }
}
```

Beispiel 10: Testfälle für die Methode `createInvoice`

Den kompletten Quelltext des JUnit-Tests finden Sie im Download-Paket der Beispielanwendung.

ANSATZPUNKTE FÜR UNTERNEHMENSSPEZIFISCHE ANFORDERUNGEN

Obwohl Acegi von Haus aus viele verschiedene Möglichkeiten der Authentifikation anbietet, kann es vorkommen, dass für ein Projekt keine der gegebenen Möglichkeiten die Anforderungen an die Software erfüllt. Darüber hinaus existieren möglicherweise bereits Komponenten, die sich um die Authentifikation kümmern. In einem solchen Fall muss man eine solche Komponente integrieren, indem man einen eigenen AuthenticationProvider implementiert. Dieser verwendet dann die bestehende Komponente und dient als Adapter für das Acegi Framework.

Es kann auch vorkommen, dass man von den gegebenen Möglichkeiten der Autorisierung abweichen möchte. Soll die Entscheidung über die Zugriffsberechtigung nicht anhand einer Rolle, sondern Beispielsweise anhand der IP-Adresse des Benutzters, anhand der Uhrzeit oder ähnlichem erfolgen, so ist ein eigener AccessDecisionVoter zu implementieren. Die vom Interface vorgeschriebene vote-Methode enthält hierbei die Entscheidungslogik.

Grundsätzlich kann jedes Interface des Acegi Frameworks leicht durch eigene Implementierungen ergänzt werden. Hierbei ist es hilfreich, dass Acegi ein OpenSource-Projekt ist, da man so den Quellcode bestehender Klassen als Vorlage und Hilfestellung für eigene Implementierungen verwenden kann.

Existieren Altsysteme, die in einem Projekt eingebunden und später ersetzt werden sollen, so kann man diese Systeme in Acegi nutzen und später austauschen. Die Änderungen hierfür beschränken sich dabei auf die Konfiguration, der Quellcode kann unangetastet bleiben.

ZUSAMMENFASSUNG

Acegi stellt die moderne Form eines Sicherheitsframeworks dar, bei dem die Komponenten lose gekoppelt und austauschbar sind. Durch den Einsatz von Acegi wird die Anwendung unabhängig von einer Sicherheitstechnologie. Spätere Technologiewechsel sind problemlos möglich, und der Anwendungscode bleibt frei von Sicherheitsaspekten. Die Integration von Acegi in bestehende Anwendungen gelingt in der Regel einfach, eine Hürde stellt jedoch der Einstieg in die Materie dar, da ein Verständnis für die Architektur und die Idee hinter Acegi vorhanden sein muss, um es effektiv einsetzten zu können. Ich hoffe, ich konnte Ihnen bei der Bewältigung dieser Hürde mit diesem Artikel ein wenig helfen.

REFERENZEN

- Acegi Security System for Spring
  (http://www.acegisecurity.org)
- Spring Framework
  (http://www.springframework.org)
- Acegi Security Reference Documentation
  (http://www.acegisecurity.org/reference.html)
- Securing Your Java Applications - Acegi Security Style
  (http://www.javalobby.org/articles/acegisecurity/part1.jsp)