Karlsruher Institut für Technologie

Diplomarbeit

Roland Steinegger

Authentifizierungs- und Autorisierungsmuster in bestehenden Sicherheits-Frameworks

Januar 2012 – Juni 2012

Erstgutachter:	Prof. Dr. Sebastian Abeck
Zweitgutachter:	Prof. Dr. Bernhard Neumair
Betreuender Mitarbeiter:	Dipl.-Inform. Aleksander Dikanski

Cooperation & Management (C&M, Prof. Abeck)
Institut für Telematik, Fakultät für Informatik
www.cm.tm.kit.edu

KIT – Universität des Landes Baden-Württemberg und
nationales Forschungszentrum in der Helmholtz-Gemeinschaft

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 30. Juni 2012

Roland Heinz Steinegger

Inhaltsverzeichnis

1 Einleitung 1

1.1 Einführung in das Themengebiet 1

1.2 Fragestellungen 2

1.2.1 Welche Sicherheitsmuster werden von Sicherheits-Frameworks unterstützt und wie werden diese implementiert? 2

1.2.2 Wie könnten Beziehungen zwischen Sicherheitsmustern modelliert werden? 3

1.3 Beschreibung des Demonstrators 3

1.4 Gliederung der Arbeit 6

2 Grundlagen 9

2.1 Sicherheitsmuster 9

2.1.1 Muster in der Softwaretechnik 9

2.1.2 Sicherheitsmuster 10

2.1.3 Die Unified Modeling Language zur Musterbeschreibung 11

2.2 Musterbeziehungen und deren Darstellung 13

2.3 Software Produktlinien und Feature Model 15

2.4 Aspektorientierte Programmierung 16

2.5 Dependency Injection 17

2.6 Representational State Transfer 18

2.7 Java Enterprise Edition 19

2.7.1 Java Servlet Spezifikation 20

2.7.2 Programmierschnittstellen für REST-basierte Webservices 22

2.8 Spring Security 24

2.8.1 Spring 24

2.8.2 Spring Security 27

3 Stand der Technik 31

3.1 Ein Mustersystem für Zugriffskontrolle (2004) 31

3.1.1 Bewertung des Ansatzes 32

3.2 Eine Mustersprache für Identitätsmanagement (2007) 32

3.2.1 Bewertung des Ansatzes 34

3.3 Softwareproduktlinienreferenzarchitektur für Sicherheit (2006) 35

3.3.1 Bewertung des Ansatzes 37

3.4 Muster und Musterdiagramme für Zugriffskontrolle (2008) 38

3.4.1 Bewertung des Ansatzes 39

3.5 Bewertung der Modellierungsformen von Musterbeziehungen 40

4 Authentifizierung- und Autorisierungsmuster und deren Umsetzung in bestehendem Sicherheits-Framework 41

4.1 Vorgehen zur Überprüfung des Spring Security Frameworks auf Muster 41
4.1.1 Automatisierte Mustersuche 41
4.1.2 Manuelle Mustersuche 42
4.2 Überprüfung auf Muster für Richtlinien 43
4.2.1 Rollenbasierte Zugriffskontrolle 43
4.2.2 Varianten der Rollenbasierten Zugriffskontrolle 47
4.2.3 Attributbasierte Zugriffskontrolle 50
4.2.4 Identitätsbasierte Zugriffskontrolle 54
4.2.5 Autorisierung 55
4.3 Überprüfung auf Architekturmuster 56
4.3.1 Intercepting Web Agent 56
4.3.2 Authorization Enforcer 60
4.3.3 Reference Monitor oder Policy Enforcement Point 63
4.3.4 Authentication Enforcer 65
4.4 Umsetzung des Zugriffskontrollbeschreibung und -durchsetzung am Beispiel der Arbeitsplatzsuche 68
4.5 Zusammenfassung der Überprüfung 71

5 Abhängigkeiten zwischen Mustern in einer Sicherheitsarchitektur 73

5.1 Feature Model zur Darstellung der Mustersprache 73
5.1.1 Merkmale einer Produktlinie im Vergleich zu Mustern einer Mustersprache 74
5.1.2 Darstellung von hierarchischen Beziehungen 74
5.1.3 Darstellung von weiteren Beziehungen zwischen Mustern 76
5.1.4 Erweiterung der Merkmaldiagramme um Teilbäume 78
5.2 Abbildung der Mustersprache auf Merkmaldiagramme 78
5.2.1 Abbildung des Abstraktionsniveaus der Muster 79
5.2.2 Abbildung der weiteren Beziehungen zwischen den Mustern 81
5.3 Merkmaldiagramme zur Unterstützung der Entwurfsphase 82
5.3.1 Benutzung der Merkmaldiagramme 83
5.3.2 Mustersprache für Zugriffskontrolle als Merkmaldiagramm 84
5.3.3 Evolution der Mustersprache für Zugriffskontrolle 86
5.4 Benutzung der Merkmaldiagramme am Beispiel der Arbeitsplatzsuche 89
5.5 Zusammenfassung 92

6 Zusammenfassung und Ausblick 93

6.1 Zusammenfassung 93

6.2 Ausblick 94

Anhänge **I**

A Veröffentlichung von Aleksander Dikanski I

B Abkürzungen und Glossar VII

C Abbildungen XI

D Literaturanalysen XIII

E Literatur XXIII

1 Einleitung

In der Einleitung wird das Themengebiet vorgestellt und die Problematik, die in der folgenden Diplomarbeit behandelt wird, erläutert. Die Lösungsansätze werden an Hand mehrerer Beispiele demonstriert, deshalb wird eine Einführung in das Umfeld des Demonstrators gegeben. Abschließend wird ein Einblick in die Struktur der Diplomarbeit und das Vorgehen zur Lösung des Problems gegeben.

1.1 Einführung in das Themengebiet

Bedrohungen auf Daten in Informationssystemen entstehen meist durch Sicherheitslücken in der eingesetzten Software, was aus dem Lagebericht IT-Sicherheit des Bundesamts für Sicherheit in der Informationstechnik [BSI11] hervorgeht. Weiter zeigt der Bericht, dass die Zahl der Sicherheitslücken und auch anderer Bedrohungen steigt. Die Sicherheit der Software wird zunehmend wichtiger und verarbeitete Daten müssen vor unbefugtem Zugriff gesichert werden.

"Security is often an afterthought in system design and implementation" [SF+06:xv]

Derzeit werden Sicherheitsbelange einer Anwendung meist nicht von Anfang an betrachtet, sondern fließen erst nach dem funktionalen Systementwurf bzw. der Implementierung ein. Es wäre von Vorteil, wenn während der ganzen Entwicklung neben den fachlichen Anforderungen auch Sicherheitsaspekte berücksichtigt würden. Auf diese Weise kommt es im Nachhinein nicht zu Problemen bei der Implementierung, da in die Architektur bereits Sicherheitsanforderungen einfließen.

Beispielsweise haben Unternehmen oft bestehende Sicherheits-Software im Einsatz und Entwickler, die sich mit der Software bereits auskennen, oder es soll eine bestimmte Software eingesetzt werden. Wird dies im Entwurf nicht beachtet, kann dies dazu führen, dass der Entwurf mit der Sicherheits-Software nicht umgesetzt werden kann und entweder der Entwurf oder die bestehende Software angepasst werden muss. Die Änderung des Systemmodells kann sehr umfangreich werden, falls die bevorzugte Software eine Reihe von Funktionen nicht anbietet, auf denen der Entwurf aufgebaut hat. Die Wahl einer neuen Software zur Umsetzung des Entwurfs hat zur Folge, dass die Entwickler sich neu Einarbeiten müssen und Expertise verloren geht. Das Erweitern des eingesetzten Sicherheitswerkzeugs stellt ebenfalls ein Problem dar, da dies einige Zeit in Anspruch nehmen kann.

[SF+06] nennt als Grund, für die Nichtbeachtung von Sicherheitsbelangen im Entwurf, den Mangel an Entwurfsmustern im Sicherheitsbereich. In der Analysephase werden Sicherheitsanforderungen identifiziert, beispielsweise der Schutz der Integrität von bestimmten Ressourcen und damit einhergehend der Schutz des Zugriffs und die Feststellung der Identität von Benutzern. Aus diesen Anforderungen sollten sich wiederkehrende Probleme und Entscheidungen für den Entwurf ergeben. Dieses strukturierte Vorgehen ist, nach der Meinung der Autoren, auf Grund der fehlenden Entwurfsmuster nicht möglich. Deshalb stellen sie eine Reihe von Sicherheitsmustern vor, darunter Möglichkeiten zur Identitätsfeststellung in einer Anwendung, kurz auch Authentifizierung genannt. Zum Beispiel werden Muster vorgestellt, die eine Authentifizierung durch ein Passwort, einen Fingerabdruck oder eine Chipkarte vorsehen. Durch diese Sicherheitsmuster ist es möglich, Lösungen für Probleme zu finden, die aus den Sicherheitsanforderungen entstehen.

Die Autoren stellen einige Lösungsmöglichkeiten für Probleme vor, jedoch fehlt der Zusammenhang zur präferierten Software respektive dem präferierten Sicherheits-Framework. Das Problem, ob der Entwurf umgesetzt werden kann, bleibt offen. Des Weiteren bieten die Muster keine Unterstützung bei der Entwicklung, da Alternativen zwischen den Mustern nicht aufgezeigt werden. Neben den Alternativen zu einem Muster werden weitere wichtige Abhängigkeiten zwischen den Mustern nicht betrachtet, beispielsweise ein gegenseitiges Ausschließen.

Zur Beschreibung von Beziehungen zwischen Sicherheitsmustern können Musterdiagramme nach [FP+08] eingesetzt werden. Diese bieten allerdings keine formale Beschreibung der Notation und somit der Beziehungen. Dies führt dazu, dass die Diagramme interpretiert werden müssen und es zu Missverständnissen kommt. Eine automatische Verarbeitung der Beziehungen ist ebenfalls nicht möglich.

Eine Lösung für das Problem, dass Sicherheitsmuster nicht von Beginn der Entwicklung an betrachtet werden, sind Mustersprachen nach der Idee von Alexander et al. aus [AIS77]. Mustersprachen beschreiben Muster und Probleme für einen speziellen Kontext und bieten eine Anleitung für die Benutzung der Muster. Derzeitig gibt es allerdings keine geeigneten Diagramme, die die Benutzung von Mustersprachen unterstützen. Zudem bieten bereits entwickelte Mustersprachen, wie beispielsweise von [DF+07, FH06], keine Verbindung zu Frameworks. Dies erschwert die durchgängige Betrachtung von Sicherheitsaspekten bei der Entwicklung.

1.2 Fragestellungen

Aus den zuvor vorgestellten Problemen ergeben sich verschiedene Fragestellungen.

1.2.1 Welche Sicherheitsmuster werden von Sicherheits-Frameworks unterstützt und wie werden diese implementiert?

Diese Frage ergibt sich direkt aus dem Problem, dass unklar ist, welche Sicherheitsmuster von den verschiedenen Sicherheits-Frameworks umgesetzt werden. Es sollte herausgefunden werden, welche Muster von einem speziellen Framework unterstützt werden. Dieser Frage soll in dieser Arbeit Bottom-Up nachgegangen werden. Das heißt, an Hand des Wissens über ein Sicherheits-Framework werden Muster ausgewählt, auf die das Framework überprüft werden soll. Durch dieses Vorgehen soll sichergestellt werden, dass die modellierten Sicherheitsmuster auch umgesetzt werden können. Der umgekehrte Top-Down Ansatz wäre, für eine Auswahl von Sicherheitsmuster zu prüfen, ob diese mit dem Framework umgesetzt werden können. Dies könnte längere Zeit in Anspruch nehmen, da das bestehende System nicht beachtet wird und somit nicht jedes angedachte Muster mit dem Framework umgesetzt werden könnte. Zudem ist die Vollständigkeit nicht gewährleistet, das heißt womöglich werden nicht alle Funktionen des Frameworks erfasst. Exemplarisch wird als Sicherheits-Framework das Spring Security Framework [SprS11c] betrachtet und auf ein Vorgehen in anderen Frameworks hingewiesen.

Neben der Identifikation von umgesetzten Mustern des Frameworks, soll eine Form gefunden werden, um das Wissen benutzbar festzuhalten. Durch Diagramme könnte dargestellt werden, dass ein Sicherheitsmuster von einem Framework umgesetzt wird oder dass es verschiedene Implementierungsalternativen gibt, beispielsweise könnte ein Framework die Authentifizierung gegenüber einer Datenbank oder einer Software zur Speicherung von Authentifizierungsdaten anbieten. Diese Diagramme könnten beim Entwurf der Architektur helfen, indem direkt ersicht-

lich ist, ob eine Architektur basierend auf Mustern von einem Framework umgesetzt werden kann.

Daneben ist es interessant, wie ein Muster auf bewährte Weise mit dem Framework implementiert werden kann. So könnte selbst ein Entwickler, der wenig oder keine Erfahrung im Einsatz des Framework hat, Muster zeiteffizient und sicher implementieren.

1.2.2 Wie könnten Beziehungen zwischen Sicherheitsmustern modelliert werden?

Diese Frage betrifft das Problem, dass die Betrachtung der Sicherheitsaspekte einer Anwendung meist nicht von Beginn an erfolgt. Mustersprachen bieten einen guten Einstiegspunkt für dieses Problem, allerdings erfordern diese ein Wissen über die betrachteten Muster und deren Zusammenhänge. Eine geeignete Darstellungsform von Musterbeziehungen, die einen Teil des bisher erforderlichen Wissens beim Entwurf einer Sicherheitsarchitektur bereitstellt, soll deshalb gefunden werden.

Meist gibt es eine Reihe von alternativen Sicherheitsmustern, die zur Lösung eines ähnlichen Problems eingesetzt werden können. Zudem gibt es weitere Beziehungen zwischen Mustern, die für den Entwurf einer Architektur wichtig sind. Muster können sich beispielsweise ausschließen oder in der Umsetzung eines Musters wird ein anderes verwendet. Mit diesem Wissen, nutzbar in einem Diagramm, könnte ein Entwickler effizient Alternativen für ein Muster oder Problem finden. Des Weiteren sieht ein Entwickler direkt, welche neuen Probleme durch den Einsatz eines Musters auftreten und welche Sicherheitsmuster zur Lösung ausgewählt werden können. Das Wissen kann auch genutzt werden, um festzustellen, ob die Muster einer Architektur umgesetzt werden können.

Eine Anleitung zur Benutzung der Diagramme ist ein weiteres Ziel. Die Anleitung soll eine Unterstützung bei der Auswahl von Mustern für Probleme bieten. Dies soll die Betrachtung der Sicherheitsaspekte einer Anwendung von Beginn der Entwicklung an unterstützen und eine Benutzung der Mustersprache durch unerfahrene Entwickler erleichtern. Entwickler können durch die Musterbeziehungen navigiert werden und so die Sicherheitsarchitektur entstehen.

Diese Idee soll zudem mit dem zuvor gewonnenen Wissen über Muster in Frameworks kombiniert werden. Die Auswahl von Mustern könnte beispielsweise eingeschränkt werden auf Muster, die mit einem bestimmten Sicherheits-Framework umgesetzt werden können. Dies würde sicherstellen, dass die Umsetzung eines Entwurfs mit einem Framework möglich ist. Der Entwickler müsste sich zudem mit einer kleineren Menge an Mustern auseinandersetzen.

1.3 Beschreibung des Demonstrators

Durch den Demonstrator soll das Vorgehen zur Nutzung der vorgestellten Konzepte von den Sicherheitsanforderungen über den Entwurf bis hin zur Implementierung an Beispielen in den jeweiligen Kapiteln demonstriert werden.

Als Beispiel wurde eine Arbeitsplatzsuche gewählt, die eine Erweiterung des KITCampusGuide ist. Der KITCampusGuide ist eine Web-Anwendung, die eine Navigation zu verschiedenen wichtigen Punkten, so genannte Points of Interest (POI), wie zum Beispiel Büros von Professoren oder Mitarbeitern, Gebäude auf dem Campus der Universität, Arbeitsplätzen und sonstigen wichtigen Orten, ermöglicht. Die Arbeitsplatzsuche ist ebenfalls eine Web-Anwendung, mit welcher Studenten und Mitarbeitern des KIT nach Lernräumen, Gruppenarbeitsräumen und

Arbeitsplätzen suchen und diese reservieren können sollen. Dabei können Wünsche für Besonderheiten des Arbeitsplatzes, wie Steckdosen- oder Netzwerkanschluss, Beamer, etc. berücksichtigt werden.

Die Arbeitsplatzsuche umfasst eine Reihe sicherheitsrelevanter Aspekte. Mögliche Angriffsszenarien sind das Erstellen von Profilen zu Personen oder, falls der Wohnort eines Anwenders bekannt ist, das Planen von Einbrüchen in die Wohnung, sofern jeder Benutzer Zugriff auf alle Reservierungen anderer Anwender hat. Zudem könnten Benutzer sich gegenseitig Reservierungen löschen und die Anwendung würde ihren Zweck nicht erfüllen. Ebenfalls wäre es möglich den Datenverkehr mitzulesen oder zu manipulieren, woraus sich obige Probleme ergeben und des Weiteren Reservierungen verhindert oder Doppelreservierungen vorgetäuscht werden können. Es gibt eine Reihe weiterer Bedrohungen auf das System, dieser Auszug an Beispielen soll aber zeigen, dass eine Absicherung des Systems notwendig ist.

Aus den genannten Bedrohungen ergeben sich Sicherheitsmaßnahmen, die wiederum zu Sicherheitsanforderungen an das System führen. Beispielsweise könnte durch Verschlüsselung und Signieren der Nachrichten verhindert werden, dass der Datenverkehr mitgelesen oder verändert werden kann. Dies führt zur Anforderung, dass die Vertraulichkeit der ausgetauschten Nachrichten gewahrt werden muss. Um das gegenseitige Löschen oder Auslesen von Daten zu verhindern, könnte als Maßnahme zwischen den Benutzern unterschieden und der Zugriff auf die Reservierungen kontrolliert werden. Als konkrete Sicherheitsanforderung ergibt sich daraus die Bewahrung der Integrität der Nachrichten. Daraus leiten sich die Notwendigkeit der Authentifizierung der Benutzer und die Zugriffskontrolle ab.

Durch die erarbeiteten Konzepte dieser Arbeit soll es möglich sein, aus solchen Sicherheitsanforderungen die Erstellung des Systementwurfs zu unterstützen. Das heißt, an Hand der Sicherheitsanforderungen der Arbeitsplatzsuche, soll mittels der Modelle eine Architektur, unter Abwägung der verschiedenen Konsequenzen, aufgestellt werden.

Beispielhaft könnte sich die Sicherheitsanforderung Zugriffskontrolle wie in Abbildung 1 aus Anwendungsfällen und Bedrohungen auf die Arbeitsplatzsuche ergeben. Um das Löschen von fremden Reservierungen zu unterbinden, wird zuvor die Berechtigung des Benutzers geprüft. Für die Erfüllung der Anforderung durch den Entwurf stehen verschiedene Muster für Zugriffskontrollmodelle zur Verfügung, beispielsweise dargestellt die direkte Zuweisung von Rechten je Benutzer oder die rollenbasierte Zugriffskontrolle.

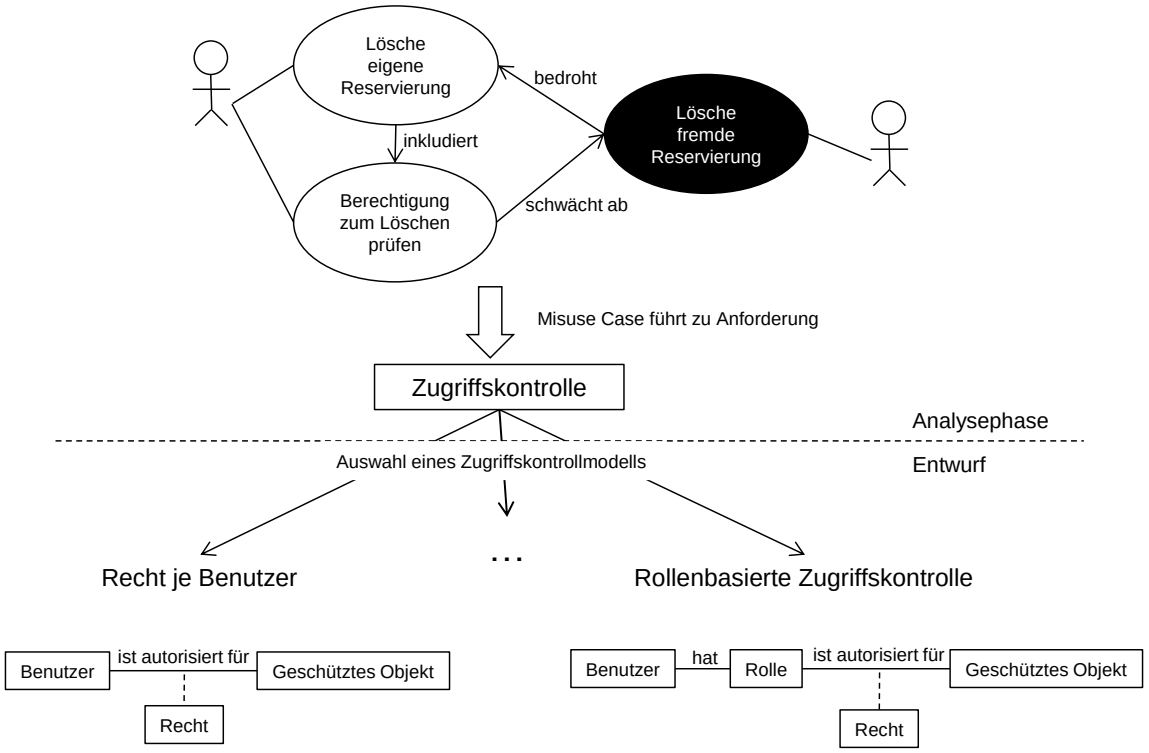

Abbildung 1: Zusammenhang zwischen Sicherheitsanforderung Zugriffskontrolle und Sicherheitsmustern.

Aus den Sicherheitsmustern ergeben sich Entscheidungsmöglichkeiten für die Implementierung, dies ist in Abbildung 2 dargestellt. Das Sicherheitsmuster rollenbasierte Zugriffskontrolle benötigt einen Speicherort für die verschiedenen Rollen. Dargestellt ist, dass die Rollen in einem LDAP-Server, einer Datenbank oder in einer XML-Konfiguration gespeichert werden könnten. Eine Entscheidung für eine oder mehrere der Varianten muss in der Implementierungsphase getroffen werden.

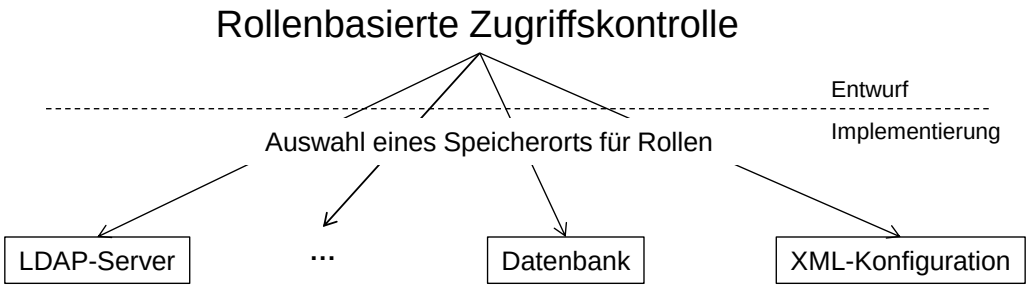

Abbildung 2: Zusammenhang zwischen Sicherheitsmuster Rollenbasierter Zugriffskontrolle und verschiedenen Implementierungen des Musters.

Bei der Umsetzung der Sicherheitsarchitektur soll für die jeweiligen Sicherheitsmuster eine Implementierung gefunden werden, die auch bestehende Systeme berücksichtigt, deshalb ist der Zusammenhang von Diensten und Implementierungsmöglichkeiten zu den verwendeten Frameworks wichtig. Abbildung 3 stellt die Verbindung des Spring Security Frameworks zu den zuvor gezeigten Varianten dar.

Spring Security bietet an, die Rollen in einem LDAP-Server, einer Datenbank, einer XML-Konfiguration oder einem CAS-Server zu speichern und von dort für die Zugriffskontrolle zu laden. Ist eines der Systeme bereits im Einsatz und soll weiter verwendet werden, so kann, auf

Grund der dargestellten Information, der Einsatz von Spring Security in Betracht gezogen werden.

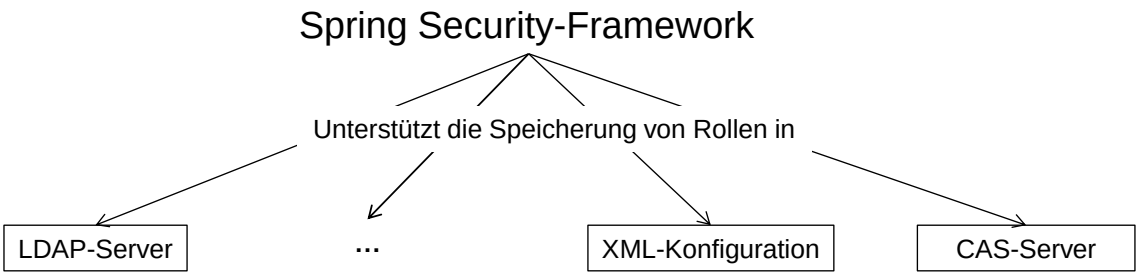

Abbildung 3: Der Zusammenhang zwischen Spring Security und verschiedenen Implementierungsmöglichkeiten zur Rollenspeicherung.

An Hand des Demonstrators sollen die verschiedenen Lösungen der Probleme in den jeweiligen Kapiteln demonstriert werden. Für die Überprüfung auf Muster wird mit der Arbeitsplatzsuche gezeigt, wie eine Auswahl an Sicherheitsmustern mit dem Spring Security Framework implementiert werden kann. Bei der Einführung einer Darstellungsform für Musterzusammenhänge und deren Anwendung in Form einer Mustersprache wird am Beispiel der Arbeitsplatzsuche demonstriert, wie die Diagramme beim Entwurf der Sicherheitsarchitektur helfen können.

1.4 Gliederung der Arbeit

Im Folgenden werden der Aufbau der Arbeit und der Inhalt der verschiedenen Kapitel vorgestellt.

Kapitel 2: Grundlagen

In diesem Kapitel werden die Konzepte, auf denen diese Arbeit aufbaut, eingeführt. Dies beinhaltet eine Einführung des Konzepts der Muster, verschiedene Formen der Beziehungsmodellierung und deren Darstellung. Software Produktlinien, deren Modellierungsform auf die Darstellung von Musterbeziehungen übertragen wird, werden ebenfalls eingeführt. Das betrachtete Framework benutzt verschiedene Paradigmen der Softwaretechnik, zu diesen gehören die aspektorientierte Programmierung, die Dependency Injection und das Representational State Transfer-Konzept. Diese Paradigmen werden ebenfalls vorgestellt. Abschließend wird auf die Java Enterprise Edition und das Spring Security Framework, mit denen die Arbeitsplatzsuche implementiert wird, eingegangen.

Kapitel 3: Stand der Technik

Im Stand der Technik wird auf verschiedene Ansätze zur Modellierung und Darstellung von Musterzusammenhängen eingegangen. Für die verschiedenen Vertreter dieser Ansätze wird eine Bewertung abgegeben. Die Kritik soll der Ansatzpunkt für die Beiträge dieser Arbeit sein und eine Einordnung der Beiträge in bisherigen Arbeiten auf dem Gebiet ermöglichen.

Kapitel 4: Authentifizierungs- und Autorisierungsmuster und deren Umsetzung in bestehendem Sicherheits-Framework

In diesem Kapitel wird gezeigt, welche Sicherheitsmuster von einem Sicherheits-Framework umgesetzt werden und wie diese in Zusammenhang mit alternativen Implementierungen im Sicherheits-Framework stehen. Konkret wird das Spring Security Framework betrachtet und gezeigt, welche Muster von diesem umgesetzt werden. Ein Abschnitt des Kapitel widmet sich

der Implementierung von Mustern am Beispiel der Arbeitsplatzsuche mit dem Spring Security Framework. Die eingesetzten Konfigurationsdateien und der Quellcode werden dabei vorgestellt.

Kapitel 5: Abhängigkeiten zwischen Mustern in einer Sicherheitsarchitektur

Aufbauend auf den vorherigen Kapiteln sollen die Sicherheitsmuster genutzt werden, um Alternativen bei der Wahl von Mustern für Sicherheitsanforderungen zu bieten. Um dies zu ermöglichen, werden Diagramme vorgestellt, mit denen die Darstellung möglich ist. Des Weiteren wird gezeigt, wie die Diagramme aus bekanntem Wissen über Musterzusammenhänge erzeugt werden können. Eine Anleitung zum Entwurf einer Sicherheitsarchitektur mit Hilfe der Diagramme wird vorgestellt. Eine Mustersprache für Zugriffskontrolle und die Erweiterung der Sprache, um neue Domänenmuster oder durch neues Wissen, soll zudem vorgestellt werden. Abschließend wird exemplarisch an der Arbeitsplatzsuche gezeigt, wie die Mustersprache in Verbindung mit den Diagrammen eingesetzt werden kann, um eine Sicherheitsarchitektur basierend auf Mustern zu entwerfen.

Kapitel 6: Zusammenfassung und Ausblick

Abschließen soll in diesem Kapitel eine Zusammenfassung der Konzepte und ein Ausblick auf weitere Arbeiten in diesem Bereich, aufbauend auf den Beiträgen der Arbeit, gegeben werden.

2 Grundlagen

Zur Bearbeitung und Beantwortung der Kernfragen in den Kapiteln 3.5 und 5 werden eine Reihe Technologien und Werkzeugen benötigt. Diese sollen in den folgenden Abschnitten eingeführt werden und ein Einblick in der Weise gegeben werden, dass das Vorgehen bei der Beantwortung nachvollzogen werden kann. Gemäß dem Aufbau der Arbeit wird zuerst auf die Techniken eingegangen, die bei der Überprüfung auf Muster von Bedeutung sind und in einem zweiten Block auf die Grundlagen für die Modellierung der Abhängigkeiten zwischen den Mustern eingegangen.

2.1 Sicherheitsmuster

Ein zentrales Thema dieser Arbeit sind Sicherheitsmuster, deren Umsetzung in einem Framework und Zusammenhänge der Muster zur Unterstützung im Entwicklungsprozess. Deshalb gibt dieser Abschnitt eine Einführung in das Konzept von Mustern, in den Bereich der Software- und Sicherheitsmuster und deren Beschreibung.

2.1.1 Muster in der Softwaretechnik

Das Konzept der Muster in der Softwaretechnik wurde aus dem Bereich der Architektur von Christopher Alexander übernommen. Dieser veröffentlichte 1977 einen Katalog für Muster in der Architektur und Städteplanung [AIS77], welcher von Kent Beck und Ward Cunningham 1987 in [BC87] aufgegriffen und in den Bereich der Softwaretechnik übertragen wurde.

Die von Alexander eingeführten Muster sollen die Effizienz beim Entwurf verschiedener architektonischer Objekte steigern, beispielweise den Entwurf einer Stadt oder eines Hauses, und boten deshalb Lösungsansätze an. Die Muster des Katalogs von Alexander sollten bewährte Lösungen für wiederkehrende Probleme in der Architektur bieten. Beck und Cunningham übernahmen dieses Prinzip und lieferten Muster zur Gestaltung von Fenstern für Anwendungsoberflächen. Unter anderem durch den umfangreichen Katalog an Entwurfsmustern von Gama et al. in [GHJV95] verbreitete sich die Benutzung von Mustern im Bereich der Softwaretechnik und es entstanden weitere Kataloge für unterschiedliche Domänen auf verschiedenen Abstraktionsebenen; in [SF+06:5] ist eine Auflistung solcher Kataloge zu finden. Der Sicherheitsbereich ist eine solche Domäne, für die eine Reihe von Sicherheitsmustern beschrieben wurde.

Ein passendes Muster für ein bestimmtes Problem zu finden ist, bei der Vielzahl der entstandenen nicht kategorisierten Kataloge, recht aufwendig. Deshalb wurde beispielsweise in [BMR+96] die Kategorisierung nach der Abstraktionsebene der Muster nach den folgenden Kategorien eingeführt: Architekturmuster, Entwurfsmuster und Idiome. Buschmann et al. liefern kurze Definitionen der verschiedenen Musterkategorien:

> *„Ein Architekturmuster spiegelt ein grundsätzliches Strukturierungsprinzip von Software-Systemen wider. Es beschreibt eine Menge vordefinierter Subsysteme, spezifiziert deren jeweiligen Zuständigkeitsbereich und enthält Regeln zur Organisation der Beziehungen zwischen den Subsystemen.“ [BMR+96:12]*
>
> *„Ein Entwurfsmuster beschreibt ein Schema zur Verfeinerung von Subsystemen oder Komponenten eines Software-Systems oder den Beziehungen zwischen ihnen. Es beschreibt eine häufig auftretende Struktur von miteinander kommunizierenden Kompo-*

> *nenten, die ein allgemeines Entwurfsproblem in einem speziellen Kontext löst." [BMR+96:13]*

In diesen beiden Definitionen tritt der Begriff *System* zentral mehrfach auf. Von diesem Begriff ist abhängig, ob ein Muster als Architekturmuster oder Entwurfsmuster anzusehen ist. Bei einem sehr großen Software-System würde der Begriff Subsystem auf größere Einheiten angewendet werden als bei einem sehr kleinen Software-System. Deshalb fügen die Autoren eine Randbedingung ein: „Jede Kategorie besteht aus Mustern, die Software-Systeme desselben Umfangs betreffen oder auf ein- und derselben Abstraktionseben liegen", [BMR+96:12]. Die Einteilung ist somit abhängig von der Größe des Software-Systems und der damit einhergehenden Unterscheidung der Begriffe Entwurf und Architektur, auf welche in [BHS07:215] eingegangen wird. In dieser Arbeit wird versucht Muster mit ähnlichem Abstraktionsniveau auf gleicher Ebene anzusiedeln und auf diese Weise konsistent zu bleiben.

> *„Ein Idiom ist ein für eine bestimmte Programmiersprache spezifisches Muster auf einer niedrigen Abstraktionsebene. Es beschreibt, wie man spezielle Aspekte von Komponenten oder den Beziehungen zwischen ihnen mit den Mitteln der Programmiersprache implementieren kann." [BMR+96:13]*

Als Ergänzung zu dieser Definition, wird in [BHS07:215] verallgemeinert, dass ein Idiom ein Muster ist, das in einem spezifischen Kontext Verwendung findet. Eine Programmiersprache ist ein solcher Kontext, der in der obigen Definition angesprochen wird, aber ebenso eine bestimmte Technologie. Das Spring Security Framework ist eine konkrete Technologie im Java Umfeld und bewährte Verwendungen des Frameworks können nach dieser Ergänzung als Idiom angesehen werden.

2.1.2 Sicherheitsmuster

Die ersten Sicherheitsmuster wurden 1997 von Yoder und Barcalow in „Architectural Patterns for Enabling Application Security" beschrieben, auf welches eine Reihe weiterer Artikel über das Thema folgten. Detailliert führt [SF+06:30] in die Geschichte der Sicherheitsmuster ein. In Abgrenzung zu Mustern in der Softwaretechnik, löst ein Sicherheitsmuster ein konkretes Sicherheitsproblem. In [SF+06:31] wird folgende Definition gegeben:

> *„A security pattern describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged into multiple concrete design structures, as well as a process to create one particular such structure." [SF+06:31]*

Diese Definition schließt nicht nur Muster für softwaretechnische Probleme ein, sondern ebenfalls Probleme aus anderen Domänen in denen Sicherheit eine Rolle spielt, beispielsweise der Schutz von Gebäuden oder Personen. In dieser Arbeit wird der Begriff Sicherheitsmuster synonym zu dem Begriff softwaretechnische Sicherheitsmuster benutzt. Die Domäne der Sicherheitsmuster soll somit innerhalb dieser Arbeit implizit das Software Security Engineering sein.

Ein umfangreicher Katalog an Sicherheitsmustern wird von Schumacher et al. in [SF+06] gegeben. Die Spanne der Abstraktionsebenen, der dort vorgestellten Sicherheitsmuster, ist sehr groß. Zum Beispiel wird einerseits das abstrakte Sicherheitsmuster *Identification and Authentication Requirements* vorgestellt, welches beschreibt wie Anforderungen für den Bereich der Identifikation und Authentifizierung erhoben werden und wie deren Wichtigkeit gegenübergestellt wer-

den kann, und andererseits wird auf das File Authorization-Muster eingegangen, welches die Zugriffskontrolle für Dateien und Verzeichnisse in einem Betriebssystem beschreibt.

Für Sicherheitsmuster gilt dieselbe Kategorisierung nach dem der Abstraktionsebene der Muster nach Architekturmustern, Entwurfsmustern und Idiomen. Die Muster können zudem gemäß Domäneneigenschaften eingeteilt werden. Im Sicherheitsbereich gibt es sogenannte Schutzeigenschaften von Geschäftswerten einer Anwendung, wobei die vier Haupteigenschaften laut [FH06:288, SF+06:20] folgende sind: Vertraulichkeit, Integrität, Verantwortlichkeit, Verfügbarkeit. Um aus diesen Schutzeigenschaften eine Einteilung der Muster zu erreichen, können die Taktiken Vorbeugung, Entdeckung und Erholung von Fægri und Hallsteinsen aus [FH06:283] genutzt werden. Ein Muster kann gemäß seinem Lösungsansatz in folgende Kategorien eingeteilt werden:

- Vorbeugung

  Vorbeugende Muster reduzieren die Wahrscheinlichkeit von negativen Vorfällen auf Schutzeigenschaften.

- Entdeckung

  Diese Kategorie beinhaltet Muster, die bei der Entdeckung von negativen Ereignissen auf Schutzeigenschaften, helfen.

- Erholung

  Muster, die bei der Wiederherstellung des Zustandes vor einem Vorfall mit negativem Einfluss auf Schutzeigenschaften unterstützen, gehören in diese Kategorie.

2.1.3 Die Unified Modeling Language zur Musterbeschreibung

Die strukturelle Beschreibung der Muster und deren Abläufe werden in den jeweiligen Kapiteln mit Diagrammen der Unified Modeling Language (UML) vorgestellt. Rupp et al. bieten folgende Beschreibung der UML:

> *„Die [...] UML dient zur Modellierung, Dokumentation, Spezifizierung und Visualisierung komplexer Softwaresysteme, unabhängig von deren Fach- und Realisierungsgeiet. Sie liefert Notationselemente gleichermaßen für die statischen und dynamischen Modelle von Analyse, Design und Architektur und unterstützt insbesondere objektorientierte Vorgehensweisen.“ [RQ+07:12]*

Zudem merken die Autoren an, dass die UML sich stetig weiterentwickelt. Dies hat unter anderem damit zu tun, dass die Sprache weder perfekt noch vollständig ist. Als Grund hierfür werden „der schnelle Fortschritt und die hohe Komplexität der heutigen Softwareentwicklung“ genannt, [RQ+07:12]. Einige Probleme der UML liegen in ihrer Entstehungsgeschichte.

Die komplexer werdenden Systeme erforderten eine Beschreibung der Zusammenhänge und Abläufe. Deshalb entstanden Ende der 80er Jahre eine Vielzahl von Modellierungssprachen und Notation für diese Zwecke, die auf Grund unterschiedlicher Darstellungsformen, Vorgehensweisen und Zielen größtenteils nicht miteinander vereinbar waren. Diese Heterogenität beeinträchtigte die Kommunikation zwischen Entwicklern und führte zum Einsatz mehrerer Notationen zur Darstellung von Systemstrukturen und -abläufen. Dieses Problem wird beispielsweise an dem Buch von Gama et al „Design Patterns“ augenscheinlich, diese benutzten zwei unterschiedliche Notationen zur Beschreibung der Muster, [GHJV95:449]. Die UML wurde schließlich von Booch, Jacbosen und Rumbaugh aus der Vereinigung einer Vielzahl von Methoden

entwickelt und etablierte sich. Die Version 1.0 wurde 1997 veröffentlicht. Ab diesem Zeitpunkt wurde die UML von der Object Management Group (http://omg.org/) weiterentwickelt.

Aus der Zusammenführung verschiedener Modellierungssprachen ergibt sich, dass die UML eine Reihe von Diagrammen umfasst. Diese können zur Beschreibung von Softwaresystemen genutzt werden. Die Diagramme teilen sich in Strukturdiagramme und Verhaltensdiagramme auf. Strukturdiagramme beschreiben hauptsächlich den statischen Teil einer Anwendung, wohingegen bei Verhaltensdiagrammen die dynamischen Aspekte im Vordergrund stehen. Zur Darstellung eines Sachverhalts ist ein einzelnes Diagramm nicht ausreichend, da es nur einen Teilaspekt beschreibt. Dies gilt nicht nur für dynamische und statische Aspekte, sondern ebenfalls innerhalb der statischen oder dynamischen Diagramme. Beispielsweise kann mit einem Diagramm dargestellt werden, dass eine Kommunikation zwischen Komponenten stattfindet (Sequenzdiagramm), zur Modellierung des konkreten zeitlichen Ablaufs ist aber ein weiteres Diagramm notwendig (Timing-Diagramm).

Bei Mustern stehen Komponenten, die eine bestimmte Rolle respektive Aufgabe im Muster übernehmen, im Vordergrund. Bei der Überprüfung ist es zudem notwendig, dass modelliert werden kann, welche Komponenten des Frameworks eine bestimmte Rolle übernehmen. Das einzige Diagramm, das diese Aufgabe explizit darstellen kann, ist das Kompositionsstrukturdiagramm (KSD). Diese können einerseits zur Darstellung der Rollen und Rollenbeziehungen eines Musters und andererseits zur Kennzeichnung des Musters im Sicherheits-Framework genutzt werden.

Die für diese Arbeit benutzen Diagrammelemente werden auf Abbildung 4 vorgestellt. Die Beschreibung des Musters wird in einer gestrichelten Ellipse dargestellt. Die Rollen sind in Kasten innerhalb der Ellipse dargestellt und Beziehungen zwischen den Rollen können mit annotierten Linien zwischen diesen modelliert werden. Eine Instanz des Musters in einer Architektur kann durch eine gestrichelte Ellipse um den Namen der Instanz und das umgesetzte Muster dargestellt werden. Die Namen der Komponenten, die eine Rolle im Muster übernehmen, werden in einem Kasten dargestellt. Der Kasten wird mit einer gestrichelten Linie zur Instanz des Musters verbunden und die Rolle der Komponente steht an der Verbindungslinie. Ein konkretes Beispiel mit dem *Model View Controller*-Entwurfsmuster und eine detaillierte Beschreibung des KSD wird in [RQ+07:201] vorgestellt.

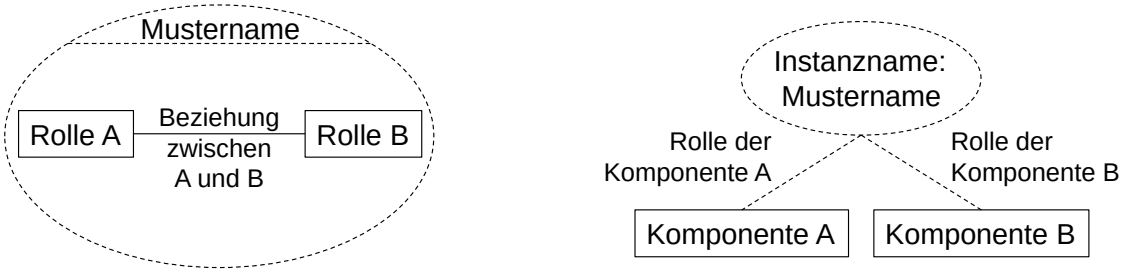

Abbildung 4: Diagrammelemente des Kompositionsstrukturdiagramms. Beschreibung der Rollen und Rollenbeziehungen eines Musters (links) und der Instanz eines Musters mit den Rollen der Komponenten (rechts)

Die statische Betrachtung der Muster ist nicht ausreichend, um deren Vorkommen im Sicherheits-Framework zu überprüfen. Die dynamischen Aspekte des Musters müssen ebenfalls betrachtet werden. Die Interaktion der Rollen des Musters soll modelliert werden, so dass überprüft werden kann, ob diese eingehalten wird. Aus der UML bieten sich Interaktionsdiagramme an, da mit diesen Kommunikation zwischen Komponenten dargestellt werden kann. Das Zeit-

verlaufsdiagramm dient hauptsächlich der Modellierung von zeitlichen Abläufen und das Interaktionsübersichtsdiagramm enthält lediglich Elemente, die Interaktionen darstellen, legt aber selbst nicht den Fokus auf die Interaktion, weswegen beide Diagramme für das vorliegende Szenario nicht geeignet sind.

Somit bleiben das Kommunikationsdiagramm und das Sequenzdiagramm als mögliche Interaktionsdiagramme. In den betrachteten Quellen über Sicherheitsmuster wird die Interaktion überwiegend mit Sequenzdiagrammen modelliert, weswegen diese im Folgenden eingesetzt werden. Mit Sequenzdiagrammen kann die Kommunikation zwischen Komponenten sowohl auf abstrakter als auch auf implementierungsnaher Ebene beschrieben werden. In dem Fall von Mustern wird die Kommunikation der Rollen des Musters dargestellt.

Die verwendeten Sequenzdiagrammelemente sind auf Abbildung 5 zu sehen. Die zeitliche Achse verläuft vertikal. Horizontal sind die verschiedenen Komponenten respektive Rollen in Kästen aufgetragen. Zwischen den Komponenten kann die Kommunikation mit Nachrichten modelliert werden. Die erste Nachricht führt zur Erzeugung einer Instanz der Komponente B durch die Komponente A. Die zweite Nachricht stellt einen synchronen und die dritte Nachricht einen asynchronen Aufruf der Komponente B dar. In Klammern hinter der Nachricht können Argumente modelliert werden. Nach der synchronen Bearbeitung antwortet Komponente B auf die Nachricht von Komponente A. Die *Aktionssequenzen* stellen dar, dass unterschiedliche Prozesse in der Komponente auf die Nachrichten reagieren. Durch *Lebenslinien* wird gezeigt, dass eine Komponente eine aktive Instanz besitzt.

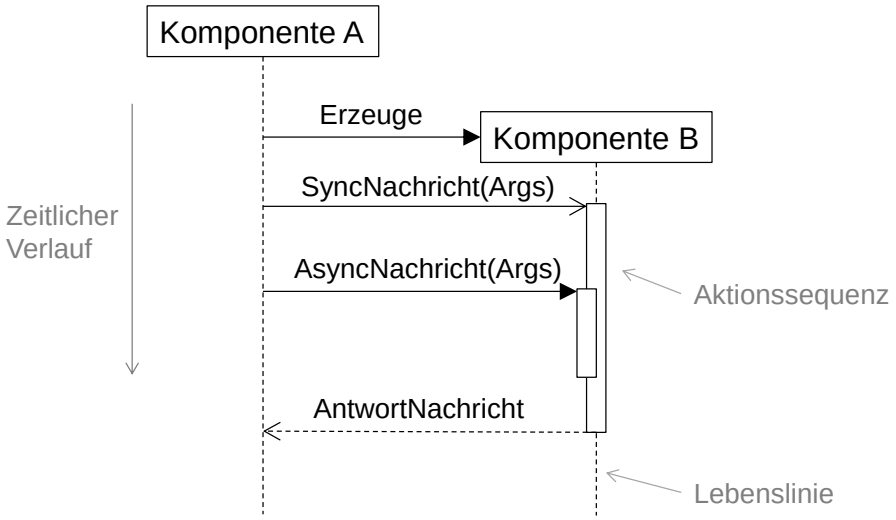

Abbildung 5: Grundlegende Elemente eines Sequenzdiagramms

2.2 Musterbeziehungen und deren Darstellung

Aus der Notwendigkeit der Modellierung von Musteralternativen stellt sich die Frage, welche bisherigen Ansätze es zur Modellierung von Beziehungen zwischen Sicherheitsmustern gibt. Zur Beantwortung dieser Frage wird auf Abhängigkeiten zwischen Entwurfsmustern und deren Darstellung in der Softwaretechnik eingegangen.

Bei der Beschreibung der Zusammenhänge gibt es zwei verschiedene Ansätze. Zum einen kann direkt die Beziehung zwischen den Mustern beschrieben werden, zum anderen können die Muster in verschiedene Kategorien eingeteilt und ein gemeinsames Merkmal herausgestellt werden.

Die sogenannte *Gang of Four* hat in [GHJV95] eine Einteilung der Muster in verschiedene Kategorien vorgenommen. Einerseits werden die Muster in klassenbasierte und objektbasierte eingeteilt, [GHJV95:11f, BHS07:223]. Objektbasierte Muster beschreiben die Lösungen für Probleme zwischen konkreten Instanzen, wohingegen klassenbasierte diese für Probleme zwischen ganzen Klassen beschreiben. Des Weiteren werden die Muster aus [GHJV95] in erzeugende, verhaltensorientierte und strukturelle eingeteilt [BHS07:223]. Alle erzeugenden Entwurfsmuster beziehen sich auf das Vorgehen zur Erzeugung von Objekten. Bei den verhaltensorientierten Entwurfsmustern steht die Interaktion von Klassen und Objekten im Vordergrund. Die strukturellen Muster beschreiben die Struktur und die Beziehungen zwischen den verschiedenen Objekten bzw. Klassen. Eine Visualisierung der Kategorisierung bietet die Gang of Four nicht. Jedoch bieten Gama et al. eine Visualisierung der Zusammenhänge der Muster auf der Buchkladde, jedoch ohne Semantik.

Auf den beschriebenen Mustern aus [GHJV95] aufbauend begann Zimmer die Betrachtungen von Zusammenhängen zwischen Entwurfsmustern [Zim95]. Die Gang of Four beschrieb die Zusammenhänge der Muster in natürlicher Sprache und modellierte sie nicht weiter. Diese textuellen Beschreibungen hat Zimmer genutzt, um eine grafische Darstellung zu erzeugen. Zimmer analysierte die Beziehungen und erstellte eine Klassifikation in drei Beziehungskategorien. Die erste Beziehung ist die „X Benutzt Y-Beziehung“ für Muster, die andere Muster zur Lösung ihres Problems benutzen können. Die „X ist ähnlich zu Y-Beziehung“ ist die zweite Form, welche ausdrückt, dass zwei Muster ein Problem auf ähnliche Weise lösen. Als letztes können einige Muster miteinander kombiniert werden, was die „X ist kombinierbar mit Y-Beziehung“ beschreibt. Dargestellt werden die Zusammenhänge in [Zim95] durch ein Diagramm ähnlich den UML Klassendiagrammen. Jeder Musternamen wird mit einem Kasten umrahmt und je nach Beziehung gibt es verschieden farbige Verbindungslinien zwischen den Kasten.

Im Vergleich zu den zuvor genannten Ansätzen wird in [BMR+96] eine Struktur vorgestellt, die bei der Auswahl von Mustern und somit beim Softwareentwicklungsprozess unterstützen soll. Ähnlich wie die Gang of Four werden die Muster in Kategorien eingeteilt. Die erste Klassifizierung geht nach der Abstraktionsebene bzw. der Phase im Entwicklungsprozess und sieht die Einteilung in Architekturmuster, Entwurfsmuster und Idiome vor. Des Weiteren wird die Klassifizierung nach verschiedenen Problemgebieten als zweite Kategorie eingeführt, beispielsweise „Vom Chaos zur Struktur“, vorwiegend Architekturmuster zur Strukturierung des Problems; „Zugriffskontrolle“, Muster zur Lösung von Zugriffskontrollproblemen und „Organisation von Arbeit, für Muster die Arbeitsabläufe besser organisieren. Ein Muster kann ebenfalls zwei Gebieten zugeordnet werden und es ist angedacht für neue nicht zuzuordnende Probleme, neue Problemgebiete einzuführen. Durch diese Einteilung, die in [BMR+96] ein Mustersystem (*Pattern System*) genannt wird, soll an Hand eines Problems effizient eine mögliche Lösung in Form eines Musters gefunden werden. Visualisiert wird die Einteilung nach den beiden Kategorien in [BMR+96] durch eine Tabelle, deren Reihen nach Architekturmustern, Entwurfsmustern und Idiomen und die Spalten nach den verschiedenen Problemgebieten sortiert sind.

Ein weiterer Ansatz ist die Beschreibung einer Mustersprache. In [BHS07] werden verschiedene Eigenschaften von Mustersprachen eingeführt. Eine Mustersprache soll demnach Probleme, die in einer bestimmten Domäne auftauchen können beschreiben und Lösungsansätze in Form von Mustern bieten [BHS07:250f]. Als mögliche Bestandteile einer Mustersprache führen Buschmann et al. Mustersequenzen ein [BHS07:183ff]. Mustersequenzen lösen eine umfangreiche Problemstellung in einem bestimmten Kontext einer Domäne. Eine solche Sequenz beschreibt, wie nacheinander Muster zur Lösung eines Problems kombiniert werden können. Musterse-

quenzen sollen aus Mustergeschichten entstehen, diese beschreiben eine solche Sequenz an Hand einer konkreten Anwendung in spezieller Form. Als Beispiel einer Mustergeschichte nennen Buschmann et al. die Muster für den Entwurf einer Architektur einer Textbearbeitungssoftware. Eine Menge von Mustersequenzen kann zur Bildung einer Mustersprache genutzt werden, so die Autoren.

Zur Eigenschaft einer Mustersprache gehört es, dass diese sich weiterentwickelt, da die Technologie sich stetig verändert. Neue Muster entstehen, bestehende Muster verändern sich oder werden nicht mehr benötigt, [BHS07:57]. Dies gilt ebenfalls für den Sicherheitsbereich und Sicherheitsmuster.

2.3 Software Produktlinien und Feature Model

Software Produktlinien sind eine Methode aus dem Software Engineering, um hohe Funktionalität und Flexibilität zu geringen Kosten zu erreichen [BK+04:3]. Sie sollen durch Variabilität einzelner Funktionen, die Anpassungen an verschiedene Bedürfnisse erleichtern. Dieses Vorgehen benötigt aus Implementierungssicht entsprechende Konzepte, damit einzelne Funktionen lose gekoppelt und leicht austauschbar sind. Daneben müssen aber auch bei der Anforderungsanalyse und beim Entwurf entsprechende Werkzeuge vorhanden bereitgestellt werden, um die Variabilität modellieren zu können. Für diese Arbeit sind diese Modellierungsmöglichkeiten aller Produkte einer Software Produktlinie von Interesse.

Zur Modellierung werden Feature Model eingesetzt. Ein Feature ist in diesem Zusammenhang eine Funktion oder Eigenschaft, die in einer Produktlinie variabel ist, das heißt, die Eigenschaft muss nicht in jedem Produkt vorkommen. Am Beispiel einer Software zur Betrachtung von Dokumenten könnte ein variables Feature das Schreiben von Notizen für Dokumente sein. Die Software zur Dokumentenbetrachtung könnte somit in der Variante mit und ohne Notizenfunktion angeboten werden.

Mit Merkmaldiagrammen (engl. Feature Diagram) kann ein Feature Model einer Produktlinie dargestellt werden. Im Allgemeinen ist die Anordnung der Features mit dem Diagramm eine Baumstruktur [Bat05], allerdings kann ein Feature in unterschiedlichen Teilbäumen auftreten und so einen Kreis bilden. Die Wurzel des Baumes ist die zu entwickelnde Software selbst. Von dieser gehen alle Features der Software aus.

Es gibt verschiedene Erweiterungen und Varianten der Merkmaldiagramme. Die in dieser Arbeit verwendeten Diagrammelemente sind auf Abbildung 6 dargestellt und werden in einer Zusammenfassung verschiedener Varianten in [SH+06] beschrieben. Die Diagrammelemente sind in der Legende dargestellt. Features können entweder Komponenten anderer Features sein oder diese Spezialisieren. Die Komponentenbeziehung entspricht dem vorigen Beispiel, der Dokumentenbetrachter hat eine Komponente Notizen. Diese Beziehungen können optional oder verpflichtend sein, das heißt diese kann oder muss angewendet werden. Die Notizfunktion wäre eine optionale Komponente, da sie im Dokumentenbetrachter auch weggelassen werden kann. Bietet eine Beziehung mehrere Elemente zur Auswahl an, so können diese als Alternativen zusammengefasst werden. Es kann ausgedrückt werden, dass alle (and) Komponenten, mindestens eine (or) oder genau eine (eXclusive-or) Komponente ausgewählt werden muss.

Das abgebildete Beispiel stellt somit dar, dass Feature A aus Feature B und C zusammengesetzt werden kann. Feature B kann dabei und Feature C muss ausgewählt werden. Des Weiteren muss genau eine Spezialisierung von Feature C in einem Produkt der modellierten Produktlinie im-

plementiert werden, sobald Feature C Teil des Produkts ist. Zudem können Feature D und Feature B nicht in demselben Produkt eingesetzt werden.

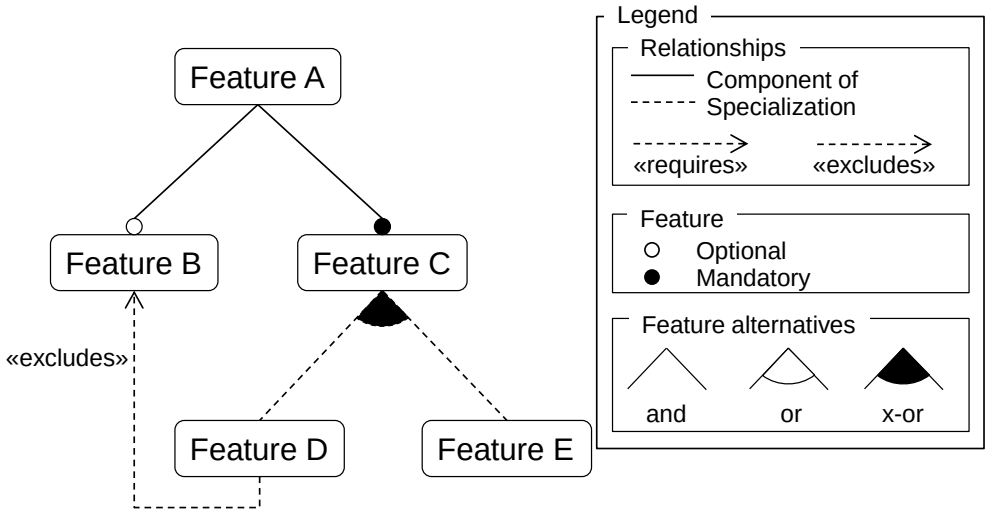

Abbildung 6: Elemente und Beispiel eines Merkmaldiagramms

Mit Merkmaldiagrammen können alle Produkte einer Produktlinie beschrieben werden. Mit dem Baum kann eine Instanz gefunden werden, indem jede modellierte Auswahl getroffen wird. Diese Instanz eines Merkmaldiagramms stellt ein konkretes Produkt der Linie dar. Je nach Anforderungen kann ein anderes Produkt kombiniert werden.

2.4 Aspektorientierte Programmierung

Die aspektorientierte Programmierung (AOP) ist ein programmiernahes Software-Paradigma, um generische Funktionalität an mehreren Klassen oder Methoden integrieren zu können. Dies trifft beispielsweise für Sicherheitsfunktionalität zu, die an mehreren Stellen in der Software eingesetzt werden soll. Die AOP wurde erstmals 1997 von Kiczales et al. in [KL+97] beschrieben. Diese identifizierten Probleme lassen sich mit den Paradigmen der objektorientierten- und der prozeduralen Programmierung nur entweder ineffizient oder unübersichtlich lösen, so die Autoren. Die Probleme entstehen durch sogenannte *Cross-Cutting Concerns*, das heißt Programmierbelange, die sich gegenseitig beeinflussen. Ein einfaches Beispiel hierfür ist die Basisfunktionalität der Anwendung und das Protokollieren der Abläufe, das während des Programmflusses erfolgen sollte. Diese Funktionalitäten beeinflussen sich somit gegenseitig.

Als Lösung wurden Aspekte vorgestellt, durch die der Programmablauf an jeder Stelle, um weitere Funktionalität erweitert werden kann. Durch einen *Aspect Weaver* werden der Programmquellcode und die Aspekte miteinander verbunden. An bestimmten Stellen, den *Joint Points*, fügt der Aspect Weaver weitere definierte Funktionalität ein. Join Points sind wohldefinierte Punkte, beispielsweise ein Methodenaufruf, ein Variablenzugriff, ein Schleifenanfang oder ähnliche Programmstellen.

Eine konkrete Implementierung zur Nutzung von AOP mit der Java Plattform ist AspectJ. Der Aspect Weaver ist ein eigener Compiler, der aus den Java-Klassen den ausführbaren Bytecode erzeugt. Aspekte bestehen bei AspectJ aus Join Points, *Pointcuts* und *Advices*. Ein Pointcut schränkt durch ein Muster, die Menge aller Join Points ein. Ein Advice beschreibt den auszuführenden Quellcode und die Stelle an der er ausgeführt werden soll. Die Stelle wird durch einen Pointcut definiert. Zudem gibt es drei verschiedene Arten von Advices. Durch den Before

bzw. After Advice, wird der Quellcode vor bzw. nach dem Join Point ausgeführt und durch den Around Advice wird der Quellcode des Advice anstelle des Join Points verarbeitet.

2.5 Dependency Injection

Die Dependency Injection basiert auf dem Paradigma der Inversion of Control (IoC). Der Name des IoC Prinzips ist bereits seine Hauptaussage, die Programmsteuerung soll umgekehrt werden. Meist wird der Kontrollfluss an einer zentralen Komponente definiert und andere Komponenten können den Kontrollfluss durch Registrierung an der zentralen Komponente beeinflussen. Das Prinzip findet beispielsweise bei Frameworks Einsatz. Das Framework übernimmt die Aufgabe der Steuerung des Kontrollflusses und mit dem eigenen Programm, wird nur ein Funktionsteil zum Framework hinzugefügt. Die Java Servlets verfolgen das gleiche Prinzip, diese registrieren sich beim Servlet Container und werden von diesem ausgeführt.

Fowler beschreibt die Dependency Injection als Muster, um Komponenten aus unterschiedlichen Projekten zu einer Anwendung mit hoher Kohäsion zusammen zu setzen [Fow04:1]. Das Hauptmerkmal der Dependency Injection ist es, die Benutzung einer Komponente von ihrer Konfiguration zu trennen [Fow04:17]. Dies führt ebenfalls zur Trennung von der Schnittstelle und ihrer Implementierung. Die Dependency Injection kann deshalb eingesetzt werden, um eine lose Kopplung der Komponenten zu erreichen.

Auf Abbildung 7 ist die Dependency Injection beispielhaft dargestellt. Die sortierbare Listenansicht benötigt einen Sortieralgorithmus, um die Einträge sortiert darstellen zu können. Dabei verweist die Komponenten nur auf die Schnittstelle eines Sortieralgorithmus. Die Verbindung zur Implementierung wird per Konfiguration zugewiesen. Vor der Benutzung der Schnittstelle erzeugt eine Komponente, in diesem Fall der *Assembler*, die Instanz der konkreten Implementierung des Sortieralgorithmus und übergibt diese an die Listenansicht. Der Sortieralgorithmus kann auf diese Weise per Konfiguration an die jeweilige Umgebung der Anwendung angepasst werden, ohne dass die sortierbare Listenansicht davon beeinflusst wird.

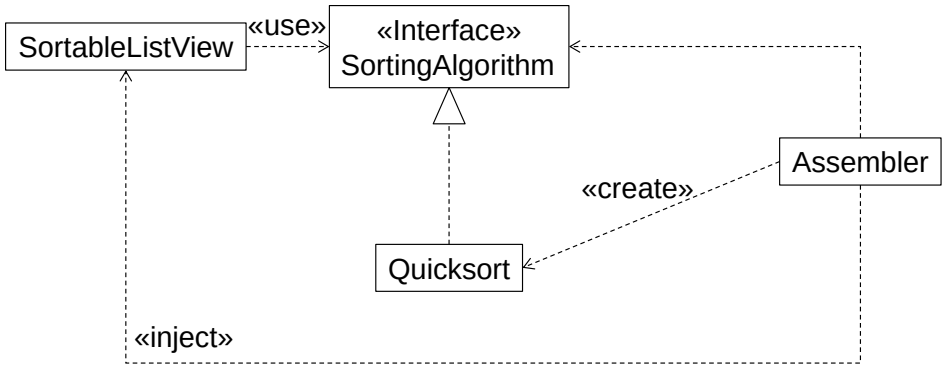

Abbildung 7: Beispiel für Dependency Injection: Der sortierbaren Listenansicht wird die konkrete Implementierung des Sortieralgorithmus injiziert

Depndency Injection im Java Umfeld kann auf verschiedene Weisen vorgenommen werden. Im Allgemeinen ist die Idee, dass für jede Klasse, die benutzt werden kann, eine Schnittstelle geschrieben wird. Diese wird von benutzenden Klassen als Typ des Feldes eingesetzt, und nicht die konkrete Implementierung. Das Injizieren der konkreten Klasse geschieht zur Laufzeit entweder per Konstruktor, der ein Objekt der Schnittstelle entgegennimmt und dem Feld zuweist, per Setter-Methode für das Feld oder per Aspekt, der bei der Erstellung des benutzenden Objekts oder beim Zugriff auf das Feld eingreift und die konkrete Implementierung einsetzt.

Quellcodebeispiel 1 zeigt die Klassen und Schnittstellen des Sortierbeispiels, bei dem über den Konstruktor injiziert werden kann. Um das injizierte Feld zu markieren, bietet ein Java Standard zudem die Annotation *@Inject*.

```
public class SortableListView {
  @Inject
  private SortingAlgorithm sortingAlgorithm;
  public SortableListView(SortingAlgorithm sortingAlgorithm) {
      this.sortingAlgorithm = sortingAlgorithm;
  }
}
public interface SortingAlgorithm { … }
public class Quicksort implements SortingAlgorithm { … }
```

Quellcodebeispiel 1: Beispiel von Klassen zur Nutzung von Dependency Injection

2.6 Representational State Transfer

Der Representational State Transfer (REST) Architekturstil wurde von Thomas Fielding in seiner Dissertation [Field00:76] eingeführt. Angedacht ist der Einsatz von REST für verteilte Hypermedia Anwendungen. REST kombiniert die folgenden Architekturstile für netzwerkbasierte Hypermedia Anwendungen [Field00:77ff]:

- Client-Server

  Die Darstellungslogik soll von der Datenverarbeitungslogik gemäß der *Separation of Concern* getrennt werden

- Stateless

  Die Kommunikation zwischen den Komponenten soll zustandslos sein. Für jede Anfrage muss somit die notwendige Information zur Verarbeitung übermittelt werden

- Cache

  Antworten sollen als (nicht) zwischenspeicherbar markiert werden, so dass der Client ggf. die vorherige Antwort benutzen kann, statt eine erneute Anfrage zu senden

- Uniform Interface

  Die Schnittstelle der verschiedenen Komponenten soll einheitlich sein und nicht auf die Art des Dienstes zugeschnitten sein.

- Layered System

  Es gibt nicht nur Client und Server, sondern auch Hybride und somit eine Schichtung der Architektur.

- Code on Demand

  Der Client soll Zugriff auf Ressourcen haben, die er nicht verarbeiten kann. Die Logik zur Verarbeitung der Ressourcen empfängt er von einem Server

Aus diesen Anforderungen ergibt sich eine Architektur für Anwendungen, die auf verschiedene Weise implementiert werden kann. Fielding stellt eine Implementierungsmöglichkeit basierend

auf den Web-Technologien Hypertext Transfer Protocol (HTTP) und Uniform Resource Identifier (URI) vor, [Field00:107ff]. Die Technologien müssen für den Einsatz von REST allerdings angepasst werden, weshalb Fielding bei der Entwicklung der heutigen HTTP Version maßgeblich beteiligt war.

In der frühen Zeit des Internets wurden Ressourcen, die sich hinter einer URI verbergen, als Dokumente interpretiert, beispielsweise ein HTML-Dokument. Diese Interpretation muss für die Anwendung von REST umformuliert werden, da sich hinter der URI bei REST-basierten Anwendungen Dienste und andere Ressourcen verbergen können. Das Hauptziel bei der Anpassung an die Bedürfnisse von REST ist, dass sich der URI so selten wie möglich ändert. Dies soll erreicht werden, indem die Semantik eines URI an seine Funktion gekoppelt ist. Auf diese Weise ändert sich der URI nicht, auch wenn der Inhalt der Ressource sich ändert.

Das HTTP soll die fehlenden Eigenschaften der Architekturstile erfüllen. Durch den URI ist bereits eine generelle Adressierung gegeben. Durch das HTTP wird das Ziel der universellen Schnittstelle vervollständigt, beispielsweise kann der Typ der Nachricht mittels der Multipurpose Internet Mail Extension (MIME) definiert werden. Zusätzlich ist das HTTP unabhängig von der Semantik der Schnittstelle, wodurch der Kopf der Nachricht effizient verarbeitet werden kann, was die Effizienzeinbußen durch das Prinzip der Schichtung vermindert. Das HTTP ist ein zustandsloses Protokoll, was eine weitere Forderung ist. Des Weiteren bietet das HTTP verschiedene Methoden zur Unterstützung des Zwischenspeicherns an, wie ein Ablaufdatum oder Versionsnummern für Ressourcen. Weitere Übereinstimmungen und vor allem die Anpassungen des HTTP an den REST Architekturstil können in [Field00:116ff] nachvollzogen werden.

2.7 Java Enterprise Edition

Die Java Platform, Enterprise Edition (Java EE) ist ein Standard zur Entwicklung von Unternehmensanwendungen mit der Programmiersprache Java. Entwickelt wurde Java EE von verschiedenen Unternehmen im Rahmen des Java Community Process (JCP). Der JCP beschreibt das Vorgehen zur Standardisierung von Technologien im Java Umfeld mit Beteiligung der Gemeinschaft der Java Programmiersprache. Innerhalb des JCP können Mitglieder Anträge zur Standardisierung stellen, einen sogenannten Java Specification Request (JSR). Die Version sechs der Java EE entstand als JSR 316, welcher die grundlegende Software-Architektur einer Java EE Applikationen mit folgenden Komponenten beschreibt:

- Java EE Platform

  Eine Standardplattform zur Ausführung einer Java EE Anwendung.

- Java EE Compatibility Test Suite

  Eine Sammlung von Kompatibilitätstests zur Verifizierung, ob eine Anwendung konform zur Java EE Plattform ist.

- Java EE Reference Implementation

  Eine Referenzimplementierung der Java EE Plattform als funktionsfähige Spezifizierung des Verhaltens der Java EE Plattform und zum Testen von Java EE Anwendungen.

- Java EE BluePrints

  Verschiedene in der Praxis bewährte Methoden zur Entwicklung von Multitier Diensten mit Thin-Client.

Die Java EE BluePrints stellen einen wichtigen Teil der Java EE Spezifikation dar, da durch diese die Anwendungen auf einem ähnlichem Fundament aufbauen und bei Einhaltung der Standards Interoperabilität gewährleistet ist und der Einsatz von Frameworks, wie dem Spring Security Framework ermöglicht wird. Die Java EE Blueprints entstehen durch Java Specification Requests (JSR) und werden in folgende Kategorien eingeteilt, [Ora11c]:

- Web Services Technologies

  Spezifikationen für den Austausch von Nachrichten zwischen Diensten und Anwendungen über Internet Protokolle. Beispiele sind der XML-Nachrichtenaustausch oder die Definition der Dienstschnittstelle im Quellcode.

- Web Application Technologies

  Technologien, die bei der Entwicklung einer Web-Anwendung genutzt werden können, beispielsweise Java Servlets zur Verarbeitung von Anfragen oder Standards zur Generierung von dynamischen Inhalten in HTML.

- Enterprise Application Technologies

  In dieser Kategorie sind allgemeine Technologien zur Entwicklung von Unternehmensanwendungen enthalten. Zu diesen gehören beispielsweise Standards für Transaktionen über Anwendungsgrenzen hinweg, die Validierung von Benutzereingaben und das Persistieren von Daten.

- Management and Security Technologies

  Authentifizierungs- und Autorisierungsspezifikationen sind in dieser Kategorie enthalten. Beispielsweise Spezifikationen, die Autorisierung und Authentifizierung bereits in die Ausführungsumgebung der Anwendung einbetten, sodass diese nicht mehr von der Anwendung selbstimplementiert werden müssen. Des Weiteren sind in dieser Kategorie Technologien zur Verwaltung der Anwendungen enthalten.

- Java EE-related Specifications in Java SE

  Diese Kategorie umfasst alle Technologien, die von der Java Platform, Standard Edition (Java SE) verwendet werden. Spezifikationen zur Verarbeitung von XML und zur Datenbankanbindung sind Beispiele für diese Kategorie.

2.7.1 Java Servlet Spezifikation

Ein zentraler Standard der Java EE ist die Java Servlet Spezifikation [JCP02], welche die Verarbeitung von Anfragen eines Clients an eine Web-Anwendung, im konkreten Fall an ein Servlet-Komponente, spezifiziert. Bei der Betrachtung des Sicherheits-Frameworks werden die meisten der vorgestellten Objekte und Abläufe wieder aufgegriffen bzw. vorausgesetzt. Die Spezifikation umfasst die Definition der Objekte, die bei einer Anfrage, den Lebenszyklus einer Anfrage von der Adressierung bis zur Antwort des Servlets, die Speicherung von Daten zur Laufzeit des Servlets, das Verarbeiten von Ereignissen und Sicherheitsaufgaben.

Zentral für die Entwicklung ist die *Servlet*-Klasse. Ein Servlet in einer Java EE Anwendung kann URL-Pfaden zugewiesen werden und dient als Einstiegspunkt bei der Verarbeitung von Anfragen an diese Pfade. Da der Standard für die Web-Schicht spezifiziert wurde, werden die Methoden des Hyper Text Transfer Protocol (HTTP) [IETF99], beispielsweise *post* oder *get*, aufgegriffen und ein Servlet kann auf diese reagieren. Im Servlet können verschiedene Objekte

benutzt werden. Zu diesen Objekten gehören der *Request*, aus dem Anfrageparameter und weitere Anfrageinformationen ausgelesen werden können, der *Response*, mit dem der Typ der Antwort und deren Inhaltgesetzt werden kann die *Session*, in der über mehrere Anfragen hinweg Daten für einen Client persistiert werden können.

Zur Absicherung der Anwendung kann ein Servlet vor unbefugtem Zugriff geschützt werden. Um dies zu gewährleisten, definiert der Standard Methoden zur Authentifizierung und Möglichkeiten Autorisierung zu konfigurieren. Zur Authentifizierung werden die Methoden des HTTP Standards und die Anmeldung per HTML-Formular angeboten. Die Autorisierung kann entweder getrennt von der Anwendungslogik des Servlets in einer XML-Konfigurationsdatei oder per Annotationen im Servlet definiert werden. Ein Benutzer erhält seine Autorisierung für bestimmte URLs respektive ein Servlet über eine Rolle. Der Rolle wird anschließend der jeweilige Zugriff gewährt oder entzogen. Wie Benutzerdaten und deren Rollen gespeichert werden sollen, ist nicht Teil der Spezifikation.

Neben diesen eingebauten Sicherheitsfunktionen bietet die Java Servlet Spezifikation einen Filtermechanismus an. Die Spezifikation sieht vor, dass ein Filter eine Anfragen an den Web Server Vor- respektive Nachverarbeiten und ebenfalls die Verarbeitung durch die Anwendung verhindern kann. Eine Menge von Filtern, die hintereinander ausgeführt werden, wird Filterkette genannt. Die Filterkette wird bei jeder Anfrage abgearbeitet, sofern kein Filter die Kette unterbricht oder eine Ausnahmebehandlung veranlasst. Ein Filter könnte beispielsweise die Authentifizierung des Clients durch Parameter in der Anfrage anbieten oder die Autorisierung des Clients prüfen und die Anfrage ggf. zurückweisen. Weitere Beispiele für Filter sind Caching Filter, Ver- und Entschlüsselungsfilter und Datenkomprimierungsfilter.

Die Konfiguration der Anwendung wird über den sogenannte Deployment Descriptor, eine XML-Datei, vorgenommen. Jeder Servlet Container muss für diesen eine Unterstützung anbieten. In dem Deployment Descriptor einer Anwendung können u.a. die Stammdaten, verschiedenen Servlets und deren Adresse und die Filter und deren Adresse angegeben werden.

Ein Beispiel eines Deployment Descriptors für die Arbeitsplatzsuche zeigt Quellcodebeispiel 2. Im Vorspann der Datei werden u.a. der Namensraum und die Schemadatei des Standards eingeführt. Darauf folgen die Stammdaten, der Name der Anwendung und eine Beschreibung. Darauf folgt die Spezifikation eines Servlets und seiner Adresse, das sogenannte Mapping. Durch das Mapping kann der Zusammenhang von Pfaden zu Servlets hergestellt werden. Das Servlet wird aufgerufen, sobald das Muster des Mappings relativ zum Anwendungspfad übereinstimmt. Im Beispiel ist nur der Stern als Wildcard Symbol angegeben, das heißt jede Anfrage an die Anwendung wird an das Servlet weitergeleitet. Die Liste der Willkommensseiten gibt an, an welche Seite weitergeleitet werden soll, wenn der Anwendungspfad ohne Pfadzusatz aufgerufen wurde.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http:/java.sun.com/xml/ns/javaee"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
     id="Arbeitsplatzsuche" version="3.0">
     <display-name>Arbeitsplatzsuche</display-name>
     <description>Anwend. zur Suche von Arbeitsplätzen</description>
     <servlet>
           <servlet-name>Arbeitsplatzsuche</servlet-name>
           <servlet-class>edu.kit.cm.Arbeitsplatzsuche</servlet-class>
           <load-on-startup>1</load-on-startup>
     </servlet>
     <servlet-mapping>
           <servlet-name>Arbeitsplatzsuche</servlet-name>
           <url-pattern>*</url-pattern>
     </servlet-mapping>
     <welcome-file-list>
           <welcome-file>suchseite.html</welcome-file>
     </welcome-file-list>
</web-app>
```

Quellcodebeispiel 2: Beispiel eines Deployment Descriptors einer Java EE Anwendung

2.7.2 Programmierschnittstellen für REST-basierte Webservices

Für den REST Architekturstil gibt es eine eigene Spezifikation, die Teil der Java EE BluePrints ist. Die Spezifikation heißt „Java API for RESTful Web Services“ (JAX-RS) [JCP09]. Diese wird bei der Implementierung der Arbeitsplatzsuche eingesetzt und deshalb ebenfalls genauer betrachtet. JAX-RS definiert eine Reihe von APIs, die bei der Entwicklung von Webservices nach dem REST Architekturstil unterstützen sollen. Die Ziele der Spezifikation sind [JCP09:2]:

- POJO-based

  Durch verschiedene Annotationen sollen Plain Old Java Objects (POJO) als Web-Ressource veröffentlicht werden können.

- HTTP-centric

  Als Grundlage für die API soll das Implementierungsbeispiel von REST basierend auf dem HTTP und den URI genutzt werden. Für die Methoden des HTTP sollen deshalb Annotationen angeboten werden, um Java-Methoden als Einstiegspunkt für Anfragen zu definieren.

- Format independence

  Die API soll unabhängig vom Nachrichtenformat sein und von diesem abstrahieren.

- Container independence

  Für die Java EE werden verschiedene Laufzeitumgebungen, sogenannte Container, angeboten, die Spezifikation soll unabhängig von diesen funktionieren.

- Inclusion in Java EE

  Eine Anwendung, die die Spezifikation benutzt, soll auf einem Java EE Container ausgeführt werden können. Zudem soll die API definieren, wie Java EE Funktionen und Komponenten genutzt werden können.

Die einzelnen Punkte werden der Reihe nach betrachtet. Mittels der *@Path*-Annotation an eine Klasse oder Methode wird ein POJO zur Web-Ressource. Bei dem Pfad kommt die Fokussierung auf das HTTP zum Tragen, da als Argument ein URI relativ zum Pfad der Anwendung übergeben werden kann. Für die URI werden verschiedene sogenannte URI Templates [JCP09:13] angeboten. Diese Templates lassen beispielsweise Wildcards oder die Definition von Pfadvariablen, die aus dem URI extrahiert werden sollen, zu. Des Weiteren kann per Annotation festgelegt werden, bei dem Aufruf welcher HTTP-Methode eine Java-Methode zur Verarbeitung aufgerufen werden soll. Durch die Verwendung verschiedener Annotationen an den Java Methodenparameter, kann beispielsweise auf Pfadvariablen oder Anfrageparameter und weitere Parameter von JEE Anwendungen zugegriffen werden. Ein Beispiel der Annotationen ist in Quellcodebeispiel 3 zu sehen.

Die Unabhängigkeit von dem Nachrichtenformat wird durch eine erweiterbare Architektur erreicht. Sogenannte Provider verarbeiten einen bestimmten Medientypen, wie im Quellcodebeispiel 3 der Typ XML (application/xml), und wandeln die Anfrage in das jeweilige Objekt um. Solche Provider können selbst implementiert werden, wodurch eine Unterstützung für jedes Nachrichtenformat möglich ist. Eine Ressource, die gemäß JAX-RS geschrieben ist, muss nach Spezifikation auf JEE Container ausführbar sein, beispielsweise als Java Servlet. Der Container kann dabei so konfiguriert werden, dass er automatisch nach JAX-RS Annotationen sucht und die gefunden Ressourcen verfügbar macht. Die Einbettung in Java EE geschieht durch die Verfügbarkeit der gleichen Objekte, die von dem JEE Container zur Verfügung gestellt werden. Im Falle eines Servlet Containers sind dies beispielsweise die aus der Java Servlet Spezifikation bekannten Objekte *Request* und *Response*.

Quellcodebeispiel 3 zeigt die Definition eines POIs als Ressource mit JEE-Annotationen. Die Ressource kann über den Pfad */poi* relativ zur URL des Anwendungskontexts abgerufen werden. Als Methode ist das Speichern von POIs exemplarisch aufgeführt. Die Methode kann, wegen der *@PUT* Annotation, per PUT-Befehl des HTTP unter der URL */poi/{id}/save* abgerufen werden, was die zusätzliche *@Path* Annotation angibt. In der URL ist *{id}* ein Platzhalter, der extrahiert werden kann. Mit der *@Consumes* bzw. *@Produces* werden die unterstützten Datentypen der Methode definiert. Die dargestellte Methode kann XML entgegennehmen und antwortet in XML Form. Durch die *@PathParam* Annotation kann auf Platzhalter in der URL zugegriffen werden, in diesem Fall wird die Kennung aus der URL in die Variable *id* gespeichert. Der zweite Parameter, *poi*, wird aus dem XML der Anfrage erzeugt.

```
@Path("/poi") //path to the poi resource. methods may extend the path
public class PointOfInterestResource {
  @PUT //method handles http put requests
  @Path("{id}/save") //{id} is a path parameter that can be extracted
  @Consumes("application/xml") //method consumes XML
  @Produces("application/xml") //and produces XML as well
  /* path parameter is extracted to variable id and
     xml request content is serialized into an poi object.
     the response is serialized from SaveResponse object to xml */
  public SaveResponse save(@PathParam("id") String id,
                           PointOfInterest poi)
  { … }
}
```

Quellcodebeispiel 3: Beispiel einer Ressource mit Annotationen nach der JAX-RS Spezifikation

2.8 Spring Security

Für die Idee, Informationen über Muster in Frameworks zu sammeln, um diese im Entwicklungsprozess nutzen zu können, wird exemplarisch ein Framework aufgegriffen. Allerdings reicht es nicht aus nur ein Framework zu betrachten. Eine Teilaufgabe dieser Arbeit soll es sein, ein Vorgehen zur Überprüfung auf Muster aufzuzeigen und als Grundlage für weitere Überprüfungen anderer Frameworks zu dienen. Als Beispiel eines Sicherheits-Frameworks wird Spring Security betrachtet, da vor dieser Arbeit bereits Erfahrungen mit dem Framework gesammelt und beispielsweise Best Practices implementiert werden konnten.

2.8.1 Spring

Das Spring Framework [SprS11a] ist eine Open-Source-Software, die derzeit von SpringSource, einer Abteilung von VMware, entwickelt wird. Spring bietet vielfältige Unterstützung bei der Entwicklung von Unternehmensanwendungen. Das Ziel des Spring Frameworks ist es, Lösungen für wiederkehrende Probleme bei der Entwicklung von Java Anwendungen gekapselt in einem Produkt anzubieten und stellt damit einhergehend auf verschiedenen Ebenen bewährte Architekturen bereit. Das Framework ist modular aufgebaut und bietet viele Möglichkeiten für Erweiterungen.

Das Konzept der Beans ist ein wichtiger Bestandteil des Spring Frameworks. Beans repräsentieren Objekte und deren Beziehungen (Objektgraph). Durch Dependency Injection können Beans aufgebaut werden. Gemäß dem JavaBeans-Standard [Ora11a] können Objekte per *Getter-*/*Setter*-Methode oder Konstruktorparamter einem anderen Objekt zugewiesen werden. Beans werden oft in XML-Konfigurationsdateien beschrieben.

Ein Beispiel einer XML-Konfigurationsdatei angelehnt an das Beispiel der sortierten Liste mit dem Sortieralgorithmus aus Kapitel 2.5 ist im Quellcodebeispiel 4 zu sehen. In dem Beispiel werden zwei Beans definiert, die *quicksort*- und die *sortedListView*-Bean. Die jeweilige Klasse der Bean ist bei der Definition angegeben. Als Wert für den Sortieralgorithmus (*sortingAlgorithm*) der sortierten Liste, wird der Quicksort-Algorithmus injiziert. Im konkreten Beispiel wird nach der Erzeugung der *sortedListView*-Bean der Algorithmus per *Setter*-Methode übergeben,

da das XML-Element *property* benutzt wird. Mit dem XML-Element *constructor-arg* ist es möglich Parameter per Konstruktor zu übergeben. Im Vorspann der Datei werden die Namensräume und die Schema-Dateien des Spring Frameworks definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
   <!-- Intro with XML namespaces and schema file, above -->
<!--Definition of quicksort algorithm and sortable list view beans-->
  <bean id="quicksort" class="edu.kit.cm.Quicksort" />
  <bean id="sortedListView" class="edu.kit.cm.SortableListView">
      <!-- The concrete sorting algorithm is injected -->
      <property name="sortingAlgorithm" ref="quicksort" />
  </bean>
</beans>
```

Quellcodebeispiel 4: Spring Beispielkonfigurationsdatei für Dependency Injection

Auf Abbildung 8 sind die für diese Arbeit wichtigsten Module des Spring Frameworks dargestellt. Die Folgende Liste bietet eine kurze Beschreibung je Modul:

Bereich „Data Access“:

- Java Database Connectivity (JDBC)

  Das JDBC Modul bietet eine Abstraktionsschicht, die einen einheitlichen Zugriff auf Datenbanken ohne Anpassungen an verschiedene Anbieter erlaubt.

- Object Relational Mapper (ORM)

  Dieses Modul vereint Funktionen verschiedener ORM Werkzeuge und ermöglicht einen Austausch des konkreten Werkzeugs.

- Transactions

  Transaktionsmanagement ist Hauptbestandteil dieses Moduls. Die Konfiguration der Transaktionen kann programmatisch oder deklarativ vorgenommen werden. Dieses Modul ist in das ORM Modul integriert und bietet dort Transaktionsunterstützung für den Zugriff auf beispielsweise die Datenbank.

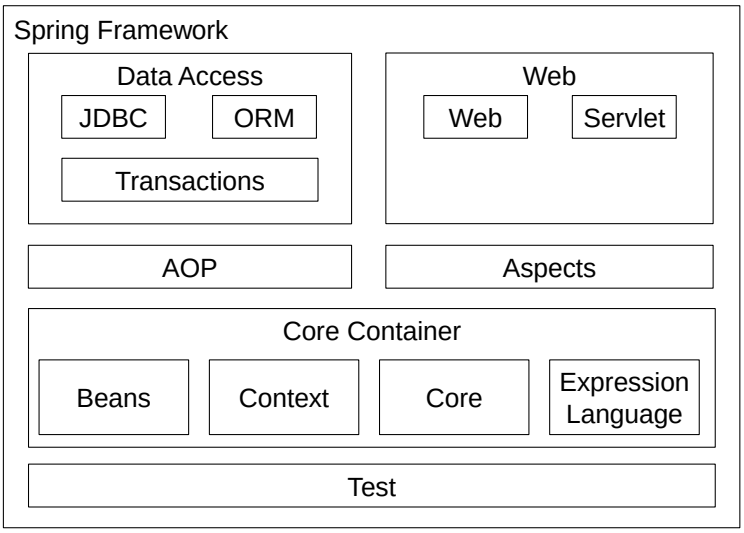

Abbildung 8: Übersicht über die Module des Spring Frameworks

Bereich „Web“:

- Web

  Eine Reihe von Basisfunktionen für die Entwicklung von Web-Anwendungen wird durch dieses Modul bereitgestellt. Beispielsweise ein Modul um Spring Hauptfunktionen beim Anwendungsstart zu laden oder Annotationen um Web-Ressourcen als solche zu markieren.

- Servlet

  Dieses Modul implementiert das Model-View-Controller-Muster, um eine Web-Anwendung mit klarer Trennung von Modellobjekt, Darstellung und Verarbeitungskomponente zu erhalten.

Bereich „Core Container“:

- Beans und Core

  Die Grundkonzepte von Spring werden durch diese Module angeboten. Zu diesen gehören die Dependency Injection, die eine lose Kopplung der Komponenten anbietet, und das umfangreiche Konzept der Beans.

- Context

  Dieses Modul bietet einen universellen Kontext und das Laden des Kontext für verschiedene Anwendungen an, beispielsweise eine laufende Java EE Anwendung oder eine Testumgebung. Dabei wird unter anderem der einheitliche Zugriff auf lokale Ressourcen und die Internationalisierung durch den Kontext angeboten.

- Expression Language

  Eine Sprache zum Stellen von Anfragen an Objektgraphen stellt das Expression Language Modul bereit. Mit dieser Sprache kann beispielsweise auf Beans gearbeitet werden.

Bereichsübergreifende Module:

- AOP

  In diesem Modul wird eine abstrakte Implementierung zur AOP Unterstützung gegeben. Enthalten sind beispielsweise Klassen zur Definition von Pointcuts oder Advices.

- Aspects

  Die konkrete Implementierung von AOP wird mit diesem Modul gegeben. Es integriert das bereits angesprochene AspectJ Framework.

- Test

  Mit dem Test Modul ist es möglich, die Anwendung mit gewohnten Spring Funktionen zu testen. Dabei integriert das Modul verschiedene Test-Frameworks.

Neben diesen Modulen sind verschiedene Projekte verfügbar, die den Einsatz von Spring in verschiedenen Anwendungsbereichen mit spezifischen Funktionen anbietet. Die Projekte basieren auf den Grundkonzepten des Frameworks und die Dependency Injection ist zentrales Konzept zur Erreichung einer losen Kopplung und Erweiterbarkeit. Zu diesen Projekten gehört beispielsweise *Spring Mobile* und *Spring for Android*, die bei der Entwicklung von mobilen Anwendungen eingesetzt werden können, oder *Spring Web Services*, das Unterstützung bei der Dienstentwicklung für die Kommunikation im Internet bietet. Genauer betrachtet wird im nächsten Abschnitt das Spring Security Framework.

2.8.2 Spring Security

Das Spring Security Framework bietet Funktionalität für die Integration von Authentifizierung und Autorisierung in eine Anwendung. Entstanden ist Spring Security Ende 2003 als eigenständiges Projekt unter dem Namen *The Acegi Security System for Spring* (Acegi Security). Auf Grund der stetigen Weiterentwicklung und Verbesserung von Acegi Security und dem produktiven Einsatz in verschiedenen Projekten wurde das Framework unter dem derzeitigen Namen Spring Security in das Spring Portfolio aufgenommen.

Das Spring Security Framework ist durch Dependency Injection modular aufgebaut und die Komponenten sind lose gekoppelt. Dies ermöglicht den Austausch und die Erweiterung des Frameworks durch eigene Funktionalität. Die Hauptkomponenten des Frameworks sind auf Abbildung 9 dargestellt. Die Benutzerinformationen werden im *Authentication*-Objekt gespeichert, welcher im *SecurityContext*, nach der Authentifizierung mit dem *AuthenticationManager*, gespeichert wird. Der *AbstractSecurityInterceptor* nimmt die zentrale Rolle bei der Durchsetzung der Zugriffskontrolle ein. Die Benutzerdaten ruft er aus dem *SecurityContext* ab, lädt die Metadaten des geschützten Objekts von der *SecurityMetadataSource* und erfragt die konkrete Zugriffsentscheidung beim *AccessDecisionManager* ab, dem er die gesammelten Informationen bereitstellt.

Die Implementierungen der einzelnen Komponenten können ausgetauscht werden, beispielsweise per XML-Konfiguration. Das heißt zum Beispiel die Mechanismen zur Authentifizierung, zur Bereitstellung von Metadaten über das geschützte Objekt oder zur Entscheidung über den Zugriff können durch eigene Implementierungen ersetzt werden. Auf diese Weise kann das Framework an neue Technologien adaptiert werden.

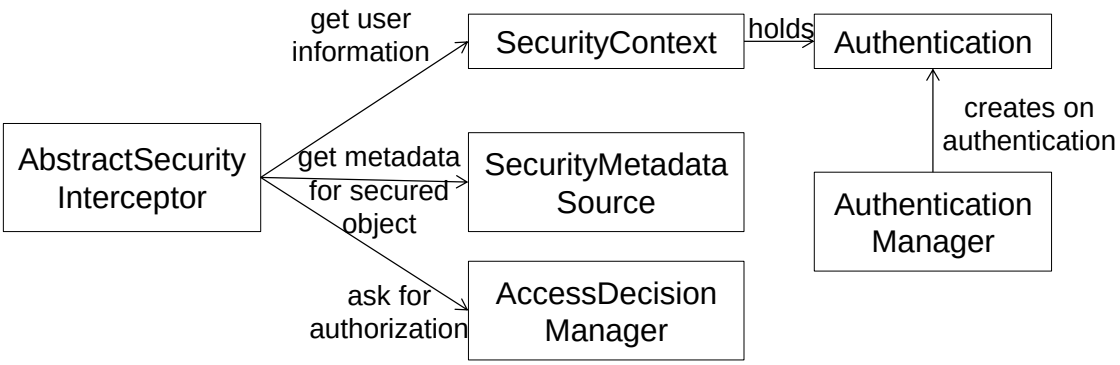

Abbildung 9: Die Hauptkomponenten des Spring Security Frameworks

Zum Einbau der grundlegenden Funktionalität des Spring Security Frameworks in eine Java EE Anwendungen sind drei Schritte notwendig [Mul10:26]:

1. Die Spring Filterkette der Anwendung hinzufügen
2. Sicherheitskonfiguration für die Anwendung aufstellen
3. Einbinden der Konfiguration

Zur Erweiterung der Anwendung benutzt das Framework das Konzept der Filter aus der Java Servlet Spezifikation. Um dem Entwickler Arbeit abzunehmen, implementiert das Framework eine eigene Filterkette, die wiederum ein Filter ist und der Anwendung hinzugefügt werden kann. Der Ausschnitt aus dem Deployment Descriptor ist in Quellcodebeispiel 5 zu sehen. Der Filter muss definiert werden und auf die abzusichernden Pfade angewendet werden.

```
<filter>
      <filter-name>springSecurityFilterChain</filter-name>
      <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
      </filter-class>
</filter>
<filter-mapping>
      <filter-name>springSecurityFilterChain</filter-name>
      <url-pattern>*</url-pattern>
</filter-mapping>
```

Quellcodebeispiel 5: Deployment Descriptor Erweiterung zur Nutzung von Spring Security

Der nächste Schritt ist die Erstellung einer Konfigurationsdatei für die Anwendung. Spring Security bietet bereits einige Voreinstellung, die mit wenig Konfigurationsaufwand angewendet werden können. Quellcodebeispiel 6 zeigt eine minimale XML-Konfiguration. Die Namensräume und Schemadateien werden im Vorspann eingeführt. Das *http*-Element stammt aus dem Spring Security Namensraum. Mit diesem Element können verschiedene voreingestellte Funktionen aktiviert werden. Mit dem Attribut *auto-config* wird die Standardkonfiguration aktiviert. Die Kindelemente *intercept-url* können zur Konfiguration der Zugriffskontrolle genutzt werden. Das *pattern* gibt mit einem Muster an, welche URLs der Anwendung geschützt werden sollen, und *access* definiert, wer Zugriff auf die betreffenden URLs haben soll. Falls mehrere *intercept-url*-Elemente eingesetzt werden, werden diese der Reihe nach bearbeitet und das erste passende Element für die Anfrage-URL benutzt. Die Benutzer der Anwendung werden in dem Beispiel

mit den Elementen *user* unter *authentication-manager*, *authentication-provider* und *user-service* angegeben. Dem Benutzer kann auf diesem Weg ein Name, ein Passwort und eine sogenannte Rolle gegeben werden. Die Rolle entspricht dabei der Rolle des Attributs *access* im *intercept-url*-Element.

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
     xmlns:beans="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security-
3.0.3.xsd">
     <http auto-config='true'>
          <intercept-url pattern="/secured/**" access="ROLE_ADMIN" />
          <intercept-url pattern="/**" access="ROLE_USER" />
     </http>
     <authentication-manager>
          <authentication-provider>
               <user-service>
                    <user name="jimi" password="jimispassword"
                         authorities="ROLE_USER, ROLE_ADMIN" />
                    <user name="bob" password="bobspassword"
                         authorities="ROLE_USER" />
               </user-service>
          </authentication-provider>
     </authentication-manager>
</beans:beans>
```

Quellcodebeispiel 6: Beispiel einer Anwendungskonfiguration für Spring Security

Der dritte und letzte Schritt zur Einbindung von Spring Security ist die Anwendungskonfiguration der Anwendung bereit zu stellen. Dies ist beispielsweise über den Deployment Descriptor möglich. In dieser muss das Spring *DispatcherServlet*, falls es noch nicht eingefügt ist, und ein globaler Anwendungsparameter mit einem Verweis auf die XML-Datei hinzugefügt werden. Der XML-Ausschnitt ist in Quellcodebeispiel 7 festgehalten.

Mit diesen drei Schritten ist die Grundfunktionalität des Spring Security Frameworks in die Anwendung eingebunden. Einem Benutzer, der eine geschützte URL anfragt, wird eine Anmeldeseite angezeigt. Auf dieser kann er seinen Benutzernamen und sein Passwort eingeben, welches mit den konfigurierten Daten verglichen wird. Falls der Benutzer authentifiziert ist und für eine Anfrage nicht autorisiert ist, so wird die Anfrage abgelehnt und es wird eine Fehlerseite angezeigt. Des Weiteren wird eine URL zum Abmelden angeboten, diese besteht aus der Anwendungs-URL und dem Suffix *j_spring_security_logout*. Die Sicherheitskonfiguration kann weiter an die Bedürfnisse der Anwendung angepasst werden, indem beispielsweise ein externer Benutzerspeicher angebunden wird oder komplexere Autorisierungsregeln definiert werden.

```
<servlet>
     <servlet-name>arbeitsplatzsuche</servlet-name>
     <servlet-class>
           org.springframework.web.servlet.DispatcherServlet
     </servlet-class>
     <load-on-startup>1</load-on-startup>
</servlet>
<context-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>
           /WEB-INF/arbeitsplatzsuche-security.xml
     </param-value>
</context-param>
```

Quellcodebeispiel 7: Erweiterung des Deployment Descriptor zum Laden der Sicherheitskonfiguration

3 Stand der Technik

Mit diesem Kapitel wird ein Einblick in bisherige Forschungsarbeiten im Zusammenhang mit den betrachteten Fragestellungen gegeben. In den Grundlagen wurde bereits auf verschiedene Ansätze zur Modellierung von Musterbeziehungen eingegangen. Diese werden aufgegriffen und konkrete Mustersysteme und Mustersprachen für den Sicherheitsbereich vorgestellt und bewertet.

3.1 Ein Mustersystem für Zugriffskontrolle (2004)

In [PF+04] wird von Priebe et al. ein Mustersystem für Zugriffskontrolle vorgestellt. Die Autoren motivieren den Artikel mit der Notwendigkeit von Mustersammlungen als Hilfe beim Systementwurf für unerfahrene Entwickler oder zum Lehren und Verstehen komplexer Probleme. In dem Artikel wird ein grundlegendes Muster der Zugriffskontrolle, die Autorisierung, beschrieben und in Beziehung zu weiteren Zugriffskontrollmustern gebracht.

Zuerst eine kurze Beschreibung der verschiedenen Muster. Das Autorisierungsmuster (engl. Authorization Pattern) gibt eine Beschreibungsform an, mit der definiert werden kann, wer auf Ressourcen in einem System zugegriffen werden kann. Die rollenbasierte Zugriffskontrolle (RBAC) beschreibt, wie der Zugriff auf Grund der Rolle der Subjekte beschrieben werden kann. Mit Hilfe der metadatenbasierten Zugriffskontrolle (MBAC) kann die Autorisierung durch Eigenschaften der Objekte oder Subjekte definiert werden. Das Sitzungsmuster (engl. Session Pattern) beschreibt, wie eine Umgebung aussehen kann, in der die Zugriffsrechte eines Subjekts eingeschränkt bzw. kontrolliert werden können. Als letztes beschreibt das Discretionary Access Control-Muster (DAC), dass über den Zugriff auf eine Ressource nur auf Basis der Identität eines Benutzers entschieden werden soll.

Die Beziehungen der Muster des Mustersystems werden auf Abbildung 10 in UML-Notation dargestellt. Die Muster RBAC, MBAC und DAC sind spezielle Ausprägungen der Autorisierungsmuster. In der Musterbeschreibung von RBAC wird zudem auf die Verwendung des Sitzungsmusters eingegangen, um die Menge der der möglichen Rollen für eine konkrete Sitzung einschränken zu können, deswegen die Benutzungsbeziehung zwischen den Mustern. Das gleiche gilt für das MBAC-Muster mit Sitzung, bei dem die Attribute des Benutzers in einer Sitzung reduziert werden können. Mit der Beziehung zwischen MBAC-Muster mit Prädikaten und DAC-Muster ist gemeint, dass das MBAC-Muster mit Aussagen über die Eigenschaften (Prädikate) eine Anpassung des DAC-Musters ist, bei der der Zugriff durch ein Prädikat für mehrere Identitäten definiert werden kann. Ein Beispiel für ein Prädikat wäre “Alter > 20“, alle Identitäten älter als 20.

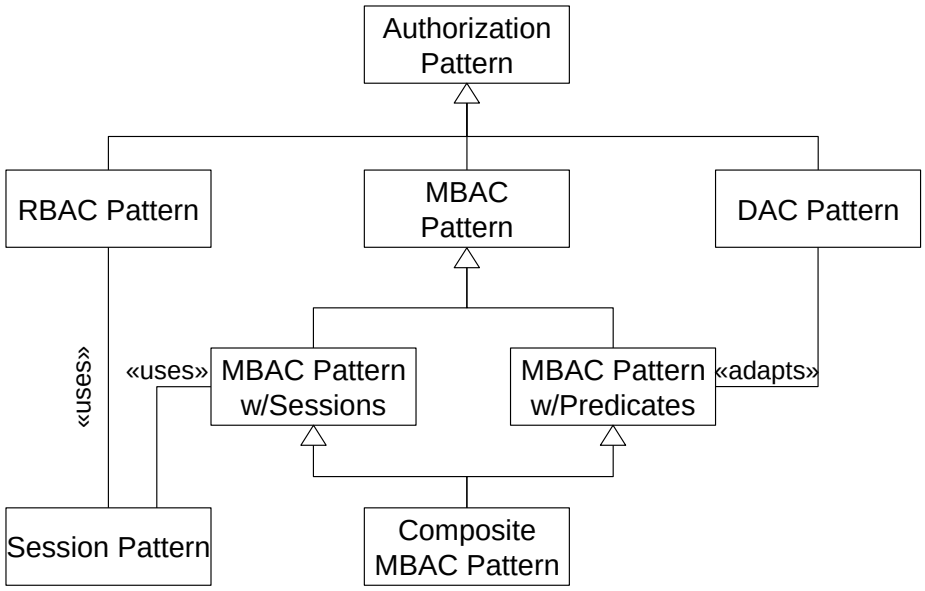

Abbildung 10: Mustersystem für Zugriffskontrolle nach [PF+04:5]

3.1.1 Bewertung des Ansatzes

Das Mustersystem von Priebe et al. weist einige Probleme hinsichtlich der Benutzung im Softwareentwicklungsprozess auf. Als kleine Kritik wäre zu nennen, dass die Benennung nicht konsistent ist, das RBAC-Muster benutzt ebenso eine Sitzung, wie das MBAC-Muster mit Sitzung, trägt dies aber nicht im Namen. Dies ist allerdings schnell zu beheben. Ein weit problematischerer Aspekt ist, dass die Semantik hinter dem Diagramm nicht ausreichend klar ist. Was bedeutet es, wenn ein Muster als Klasse modelliert wird? Und darauf aufbauend stellt sich die Frage, was die Beziehungen zwischen den Klassen in diesem Kontext bedeuten? Abgesehen davon wird in dem Mustersystem nicht auf Implementierungen der Muster eingegangen. Der Nutzen des Mustersystems oder der Darstellung als UML Diagramm bei der Softwareentwicklung wird ebenfalls nicht erklärt.

Durch eine genauere Erklärung der Darstellungsform von Mustern mit UML, könnten einige der Probleme gelöst werden. Für den Einsatz des Mustersystems wäre zudem eine Beschreibung der Verwendung bei der Softwareentwicklung notwendig. Dies könnte beispielsweise durch eine Kategorisierung der Muster, ähnlich dem Mustersystem von Buschmann et al. in [BMR+96:363], gelöst werden, wodurch die Auswahl eines Musters vereinfacht wird.

3.2 Eine Mustersprache für Identitätsmanagement (2007)

Einen anderen Ansatz als ein Mustersystem verfolgen Delessy et al. in [DF+07]. Die Autoren stellen eine Mustersprache für Identitätsmanagement vor. Als Mustersprache hat die Sammlung von Mustern somit auch den Anspruch vollständig zu sein [BMR+96:358]. Um dieses Ziel zu erreichen, werden drei Muster in dem Artikel vorgestellt und mit bekannten Mustern für das Identitätsmanagement in Beziehung gebracht. Die neuen Muster betreffen Probleme, die bei der Zusammenarbeit von Unternehmen auftauchen, die sogenannte Federation.

Die Mustersprache umfasst zwei Muster, die Modelle zur Beschreibung der Zugriffskontrolle liefern. Dies sind die attributbasierte Zugriffskontrolle (engl. Attribute-Based Access Control, ABAC), bei der der Zugriff durch Aussagen über verschiedene Attribute beschrieben wird, und

die Zugriffsmatrix (engl. Access matrix), bei der für jedes Paar von Subjekt und geschütztem Objekt die Zugriffsmethoden definiert werden. Für das Zugriffsmatrix-Muster werden zwei Implementierungsvariante angegeben, diese heißen Zugriffskontrollliste (engl Access Control List, ACL) und Befähigung (engl. Capability). Im Artikel wird zudem angegeben, dass das Zugriffsmatrix-Muster durch jedes andere Zugriffskontrollmodell, wie beispielsweise der bereits erwähnten rollenbasierten Zugriffskontrolle, ersetzt werden kann. Das Reference Monitor-Muster ist ebenfalls Bestandteil der Sprache und beschreibt, wie die Zugriffskontrollmodelle durchgesetzt werden können. Das Authenticator-Muster beschreibt, wie sich ein Subjekt authentifizieren kann. Bei der Authentifizierung wird ein Berechtigungsnachweis benötigt, diesen beschreibt das Credential-Muster.

Die drei Muster für Federation sind Zirkel des Vertrauens (engl. Circle of Trust), Identitätsföderation (engl. Identity Federation) und Identitätsanbieter (engl. Identity Provider). Der Zirkel des Vertrauens beschreibt, wie eine Vertrauensbeziehung zwischen Dienstanbietern aufgestellt werden kann, damit die Subjekte der Anbieter gegenseitig auf definierte sicherheitskritische Dienste zugreifen können. Das Identitätsföderations-Muster beschreibt, wie Identitätsdaten transparent zwischen verschiedenen Dienstanbietern ausgetauscht werden können. Durch die Anwendung des Identitätsanbieter-Musters wird gewährleistet, dass es eine zentrale Instanz zur Verwaltung von Identitäten innerhalb einer Sicherheitsdomäne. Das letzte Muster der Mustersprache ist SAML-Assertion, das Ausgangspunkt für die Identitätsföderation ist. SAML steht für Security Assertion Markup Language und ist ein auf XML basierender Standard zum Austausch von Authentifizierungs- und Autorisierungsinformationen. Das SAML-Assertion-Muster beschreibt ein Format zum Austausch von Identitätsinformationen zwischen Sicherheitsdomänen.

Des Weiteren werden von Delessy et al. die Beziehungen der Muster durch Abbildung 11 dargestellt. Neben den bereits beschriebenen Beziehungen stellt das Diagramm zusätzlich dar, dass das SAML-Assertion-Muster das attributbasierte Zugriffskontrollmuster und das Credential-Muster implementiert und eine Variante des Befähigungsmusters ist. Hinzu kommt, dass die Identitätsföderation die Identitätsanbieter beinhaltet und das Credential-, Authenticator- und Kreis des Vertrauens-Muster für seine Umsetzung benutzt. Die letzte noch nicht betrachtete Beziehung ist die Benutzung des Musters Zirkel des Vertrauens durch den Identitätsanbieter.

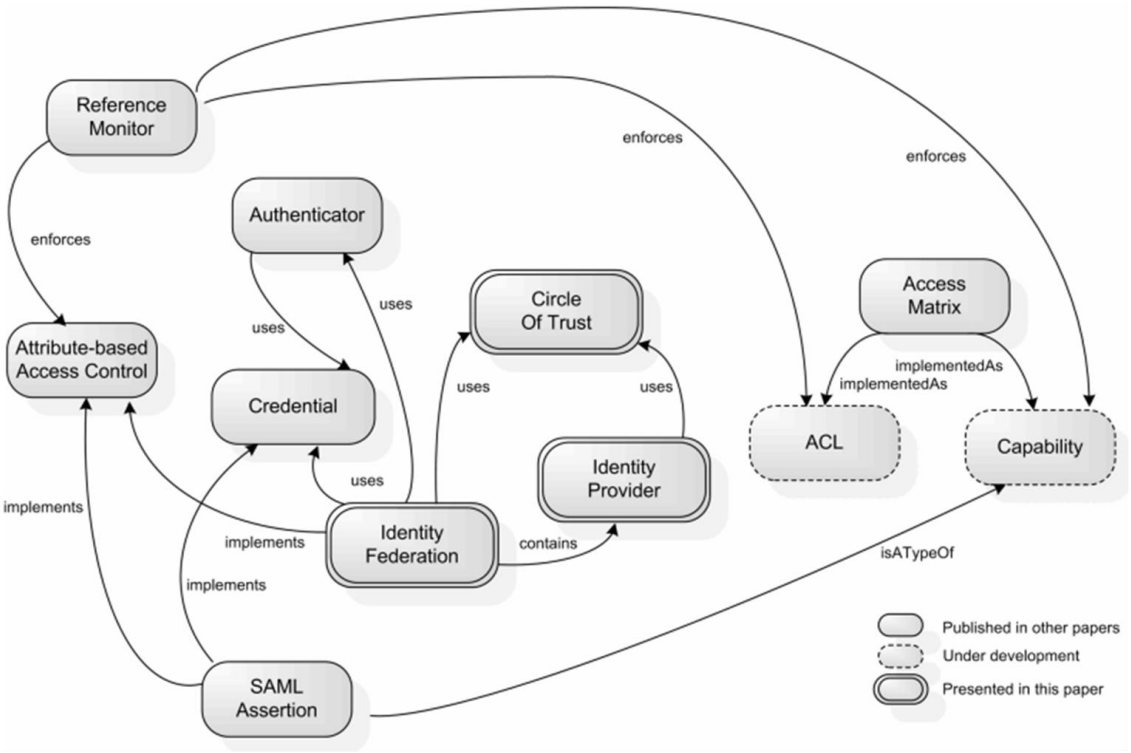

Abbildung 11: Beziehungen der Mustersprache für Identitätsmanagement. Quelle: [DF+07]

3.2.1 Bewertung des Ansatzes

In dem Artikel wird die Benutzung der Mustersprache nicht beschrieben. Anzunehmen ist, dass beim Entwurf in jedem Fall ein Reference Monitor und ein Authenticator umgesetzt werden sollte. Eine der Zugriffskontrollmodelle sollte zudem umgesetzt werden, wobei nicht klar ist, warum die attributbasierte Zugriffskontrolle herausgestellt wird und nur auf einen Austausch der Zugriffsmatrix hingewiesen wird. Der Artikel klärt zudem nicht darüber auf, warum das vorgestellte Identitätsföderationsmuster das einzige Muster zur Lösung des Föderationsproblems ist oder welche Alternativen es ggf. gibt. Damit einhergeht, dass ein zentraler Identitätsanbieter durch das gleichnamige Muster in der Sprache vorgegeben wird. Bei dieser Architekturentscheidung wird jedoch nicht erklärt, warum dies so sein muss und warum nicht ein verteilter Ansatz möglich ist.

Zur Darstellung der Beziehungen wird eine eigene Notation gewählt, wobei die Bedeutung einer Kante zwischen Mustern an diese annotiert ist. Hierdurch ist die Semantik der Annotationen nur durch einen Menschen lesbar und fehleranfällig, da diese interpretiert werden muss. Zudem führt dies dazu, dass die Annotationen nicht einheitlich sind. Beispielsweise wird zum einen *implements* und zum anderen *implementedAs* mit umgekehrtem Pfeil verwendet. Eine Hierarchie zwischen den Mustern, die bei der Benutzung der Sprache helfen würde, ist so schwerer zu erkennen, als wenn die Pfeilrichtung durchgehend zum abstrakteren oder zum konkreteren Muster gerichtet ist.

Ein Problem dieser Darstellung der Musterbeziehungen ist die Unübersichtlichkeit. In dem konkreten Diagramm bildet sich ein Zentrum, in dem sich Muster mit hoher Kohärenz sammeln. Zwischen diesen Mustern bestehen einige Beziehungen, die das Diagramm füllen. Kämen weitere Muster hinzu, beispielsweise Implementierungsvarianten des Authenticator-Musters, so müsste das Diagramm umstrukturiert werden, um es leserlich zu halten. Eventuell wäre es sinnvoll, Teile in einem eigenen Diagramm darzustellen, falls dies möglich ist.

Das Hauptproblem der Mustersprachendarstellung stellen die Abstraktionsebenen der Muster dar. Das Diagramm enthält sowohl recht abstrakte Muster, wie Identitätsföderation, als auch implementierungsnahe Muster, wie die Zugriffskontrolllisten. Durch die Darstellung ist dies, auf Grund der fehlenden Hierarchiestufen, nicht ersichtlich. Dies führt zu Problemen, beispielsweise wenn versucht wird, eine schrittweise Verfeinerung des Entwurfs zu erreichen. Es ist nicht klar, welche Muster der Sprache für einen Grobentwurf und welche für einen Feinentwurf geeignet sind. Hinzu kommt, dass durch die fehlende Darstellung der Abstraktionsebenen Musteralternativen nicht ersichtlich sind.

Insgesamt enthält der Ansatz einige ungeklärte Probleme. Jedoch erkennen Delessy et al., dass zur Darstellung der Musterbeziehungen die derzeitigen Diagramme der UML ungeeignet sind und bieten eine eigene Darstellungsform an. Als Ausgangspunkt für eine umfassende Sprache sind die beschriebenen Muster geeignet, da verschiedene Aspekte, wie Föderation, Authentifizierung, Autorisierung und deren Durchsetzung betrachtet werden. Diese könnten um weitere Muster und Musteralternativen ergänzt werden.

3.3 Softwareproduktlinienreferenzarchitektur für Sicherheit (2006)

Fægri und Hallsteinsen stellen in ihrem Journalbeitrag [FH06] ein Konzept zur Erstellung einer Sicherheitsarchitektur für eine Softwareproduktlinie vor. Fægri und Hallsteinsen beschreiben ein dreiteiliges Modell, das bei der Erstellung der Architektur unterstützen soll. Mit dem Modell versuchen die Autoren eine einheitliche Sprache für die Sicherheitsentwicklung anzubieten und eine Brücke zwischen Anforderungserhebung und Architektur zu bauen. Das konzeptionelle Modell ist auf Abbildung 12 dargestellt. Es besteht aus einer Decision Support-Modell, einem Security-Modell und einem Architecture-Model.

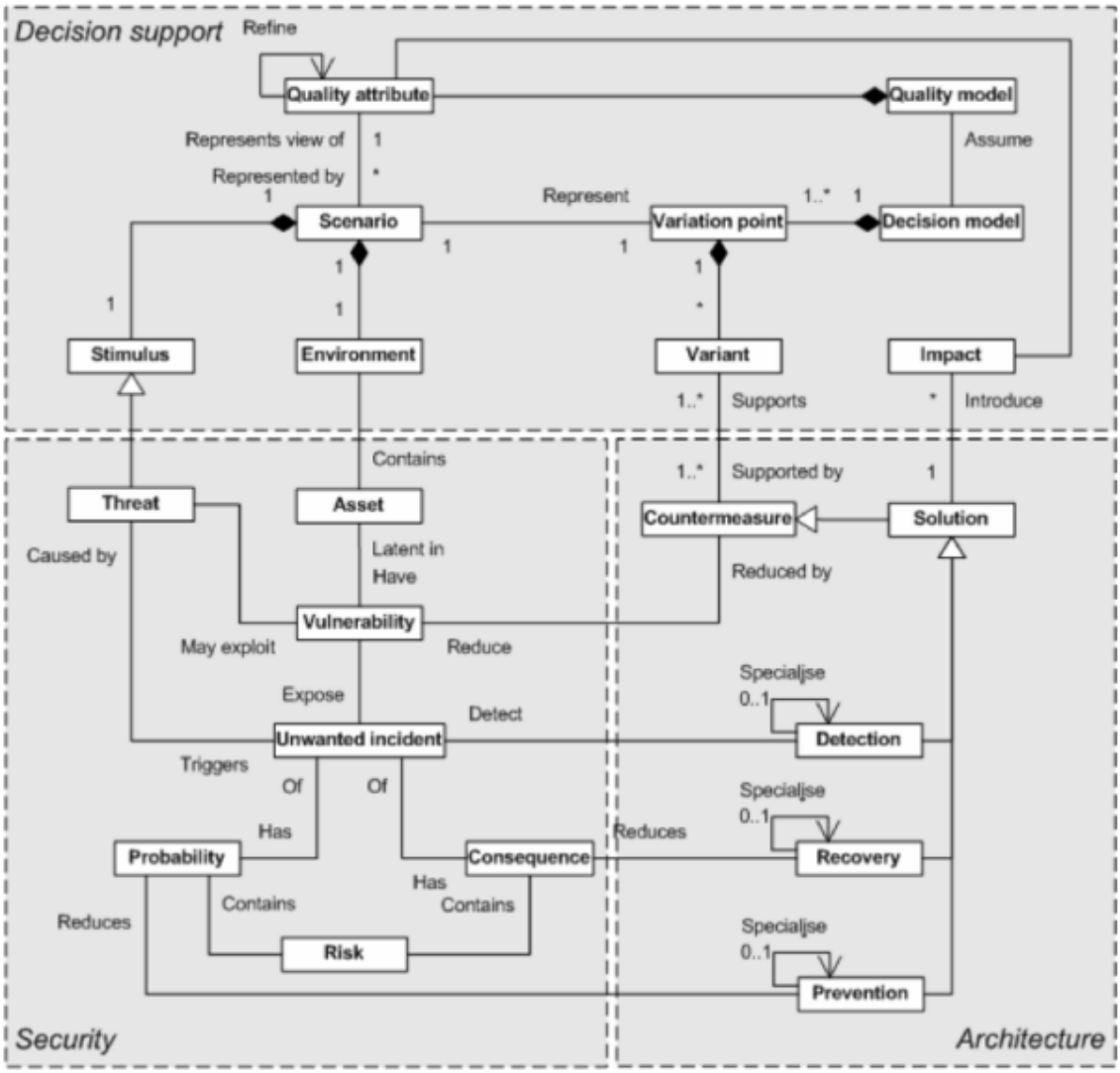

Abbildung 12: Konzeptionelles Modell der Referenzarchitektur. Quelle: [FH06:281]

Das Gesamtmodell setzt bereits bei der Erhebung der Sicherheitsanforderungen an. Diese können mit Qualitätsattributen des Decision Support-Modells strukturiert und beschrieben werden. Die Qualitätsattribute sind beispielsweise Verfügbarkeit oder Vertraulichkeit. Aus ihnen ergeben sich Szenarien für die Anwendung, die eine bestimmte Umgebung und negative Einflüsse (engl Stimulus) enthalten. Die Szenarien bieten zudem verschiedene Varianten.

Der Zusammenhang zum Sicherheitsmodell besteht durch die negativen Einflüsse, die eine Bedrohung (engl Threat) darstellen und der Umgebung, die bestimmte Unternehmenswerte (engl Asset) beinhaltet. Die Unternehmenswerte haben bestimmte Schwachstellen (engl Vulnerability), die durch einem ungewollten Zwischenfall (engl Unwanted Incident) offengelegt werden können. Der Zwischenfall wurde dabei durch eine Bedrohung verursacht. Ein Zwischenfall hat eine gewisse Eintrittswahrscheinlichkeit und Konsequenzen. Diese führen gemeinsam zu einem Risiko.

Auf Seiten des Architekturmodells gibt es verschiedene Lösungsmöglichkeiten, die durch Gegenmaßnahmen Schwachstellen beheben. Die Gegenmaßnahmen unterstützen dabei verschiedene Varianten der Szenarien. Die eingesetzten Lösungen der Sicherheitsarchitektur und deren Gegenmaßnahmen werden durch die Qualitätsattribute beeinflusst. Ein Beispiel hierfür ist, dass zur Erhöhung der Verfügbarkeit andere Lösungen benötigt werden, als wenn der Zugriff auf

Ressourcen geschützt werden soll. Zwischen diesen Attributen muss abgewägt und andere Lösungen implementiert werden. Die architekturellen Lösungen teilen sich auf in präventive, detektierende und Maßnahmen zur Wiederherstellung.

Für das Entscheidungsmodell werden verschiedene Szenarien und deren Varianten vorgestellt. Für eine Anwendung werden geeignete Szenarien gewählt. Für ein Szenario werden wiederum verschiedene Lösungsmöglichkeiten in Form von Varianten angeboten. Die Lösungsmöglichkeiten gehören in das Modell der Architektur. Die Lösungsmöglichkeiten sollen sich nach der Idee der Autoren Verfeinern hinsichtlich der Abstraktionsebene. Die Hierarchie der Lösungen stellen die Autoren wie auf Abbildung 13 dar. Die abstraktere Ebene wird von den Autoren Taktik genannt, bis diese ein Niveau erreicht hat, dass die Lösungsansätze als Muster bezeichnet werden können.

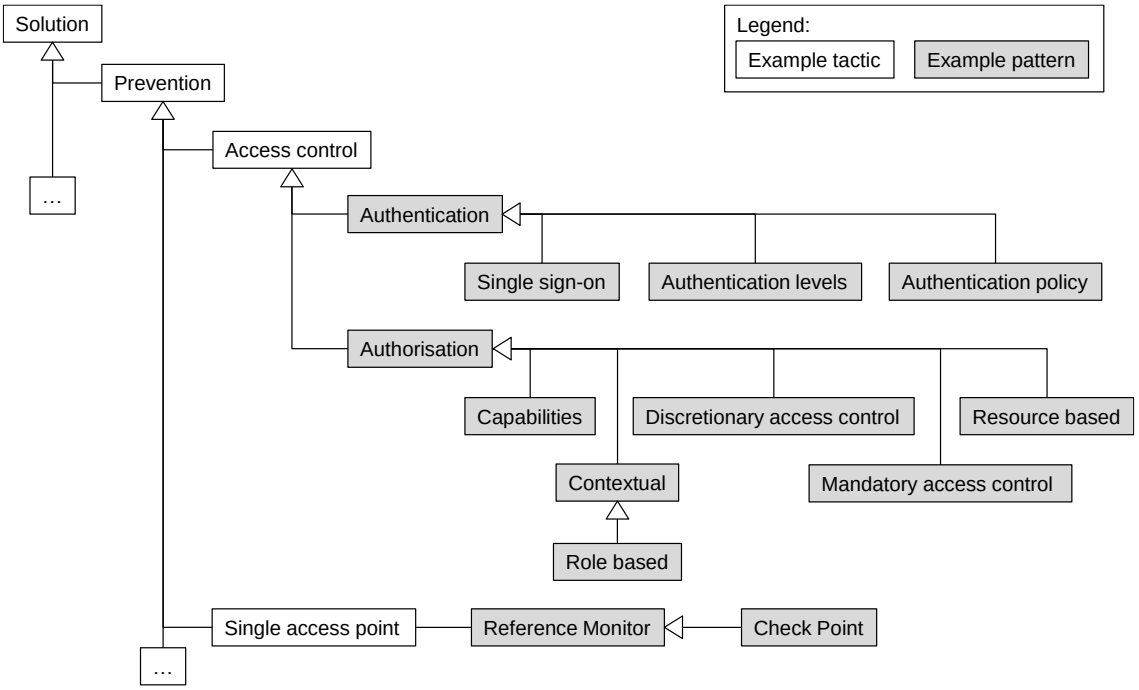

Abbildung 13: Ausschnitt der Sicherheitsarchitektursprache aus [FH06:303]

3.3.1 Bewertung des Ansatzes

Das Qualitätsmodell bietet eine gute Grundlage für die gemeinsame Kommunikation und so zur Auswahl geeigneter Szenarien, auf die Risikoanalyse und Bewertung von Unternehmenswerten wird in dem Artikel nicht genauer betrachtet. Der Zusammenhang ist für ein tieferes Verständnis der, durch die Autoren angedachten, Auswahlprozesse mit dem Sicherheitsmodell notwendig.

Der Ansatz, einen Übergang von Sicherheitsanforderungen zur Architektur zu gewährleisten, ist auf jeden Fall eine gute Idee und ein erster Schritt ist mit dem Artikel getan. Allerdings stellt sich die Frage, ob textuell beschriebene Szenarien der geeignete Weg sind. Warum werden beispielsweise keine Misuse Cases [SO05] verwendet, um die Bedrohungen auf das System zu identifizieren? Aus den Anwendungen des Decision Support-Modells, die in der Validierung der Ansätze des Artikels beschrieben werden, wird zudem nur auf einzelne Szenarien, die ausgewählt wurden, eingegangen. Hier stellt sich die Frage, ob dies ausreichend ist, um die Software zu schützen oder ob dies nur zur groben Architektur der Anwendung verhalf.

Inwieweit das Vorgehen zur Erstellung der feineren Architektur eingesetzt werden kann, müsste mit einer feineren Sicherheitsarchitektursprache getestet werden. Diese ist derzeit sehr grob und bietet wenige Möglichkeiten für Verfeinerungen. Des Weiteren sind Zusammenhänge zwischen den Mustern und Taktiken in der Darstellungsform nicht modelliert. Fægri und Hallsteinsen bieten allerdings eine Möglichkeit, zur Visualisierung von Auswahlmöglichkeiten zwischen verschiedenen Lösungsansätzen. Dies bietet beim Entwurf Vorteile, weil dem Entwickler eine konkrete Auswahlhilfe gegeben wird, im Vergleich zu den zuvor vorgestellten Mustersystem- und Mustersprachendarstellungen.

Die Autoren bieten eine Beschreibung, wie ihre Modelle beim Entwurf der Sicherheitsarchitektur eingesetzt werden können und bieten Beispiele an. Dies erleichtert den Einstieg bei der Benutzung und zeigt zudem, dass das Konzept funktioniert. Weiter gehen die Autoren auf die Unvollständigkeit der Szenarien und Lösungsansätze ein und beschreiben, wie die Modelle erweitert werden können. Allerdings stellt sich dabei die Frage, ob die vorgestellten Modelle für eine umfangreiche Sicherheitsarchitektur, deren Lösungsansätze von sehr abstrakten Taktiken bis zu konkreten Implementierungsbeschreibungen reichen, geeignet sind. Die Szenarien müssten für eine solche Sicherheitsarchitektur ebenfalls angepasst werden. Die Szenarien müssten konkreter werden und auf konkretere Lösungsansätze verweisen. Ob die Modelle hierfür geeignet sind, müsste an Hand von Studien getestet werden.

Das Ziel des Ansatzes müsste es sein, ebenfalls eine Brücke zwischen Architektur und Implementierung zu bauen, so dass eine durchgängige Entwicklung von den Anforderungen bis zur Implementierung möglich ist. Die Sicherheitsarchitektursprache müsste bis auf implementierungsnahe Muster ergänzt und auf ihre Tragfähigkeit getestet werden. Die Masse an Mustern und Szenarien für die Muster könnte die Anwendung des Modells stark beeinträchtigen, da die Suche nach passenden Szenarien lange dauern würde und vermutlich eine Vielzahl an Szenarien für die Anwendung passend wären.

3.4 Muster und Musterdiagramme für Zugriffskontrolle (2008)

Die Motivation von Fernandez et al. ist, dass Entwickler beim Entwurf auf grundlegende Zugriffskontrollmodelle zurückgreifen und kein passendes Modell verwenden, weil die bisherige Vielzahl an Zugriffskontrollmodellen die Entwickler verwirren. Die Autoren stellen deshalb in [FP+08] Musterdiagramm an Hand von Zugriffskontrollmodellen vor. Daneben wird gezeigt, wie die Diagramme durch einen Entwickler benutzt werden können, um ein Muster auszuwählen, und wie neue Zugriffskontrollmuster entwickelt werden können. Der Artikel verweist auf keine bestehende Literatur, auf welcher er aufbaut, weswegen angenommen werden kann, dass dies neue Ansätze sind.

Ein Beispiel eines Musterdiagramms zeigt Abbildung 14. Die Beschreibung der Beziehung zwischen zwei Mustern ist an die Verbindungslinie annotiert. Beispielsweise spezialisiert das sitzungsbasierte Reference Monitor-Muster das Reference Monitor-Muster. Das Reference Monitor-Muster setzt zudem die richtlinienbasierte Zugriffskontrolle durch. Die gestrichelte Linie trennt Muster verschiedener Abstraktionsebenen. Aus dem Artikel geht nicht klar hervor, ob es eine konkrete Semantik hinter den Diagrammen gibt und ob es beispielsweise Regeln für die Annotation an Verbindungslinien zwischen Mustern gibt.

Weiter wird von Fernandez et al. beschrieben, wie neue Zugriffskontrollmodelle abgeleitet werden können. Hierfür werden zwei Methoden beschrieben. Einerseits das Erweitern bestehender Modelle um Entwurfsmuster, wie das Kompositum, welches die Erzeugung von Hierarchien

zwischen Objekten beschreibt oder des Sitzungsmusters. Als Beispiel wird die Erweiterung der Rollen eines Subjekts oder des geschützten Objekts um das Kompositum-Muster genannt. Rollen bzw. Objekte können durch die Erweiterung hierarchisch aufgebaut werden. Als weiteres Vorgehen zur Musterfindung wird die Analogiemethode beschrieben, das heißt, ein Zugriffsmodell wird für einen Bereich neu modelliert und in einem anderen Kontext wiederverwendet.

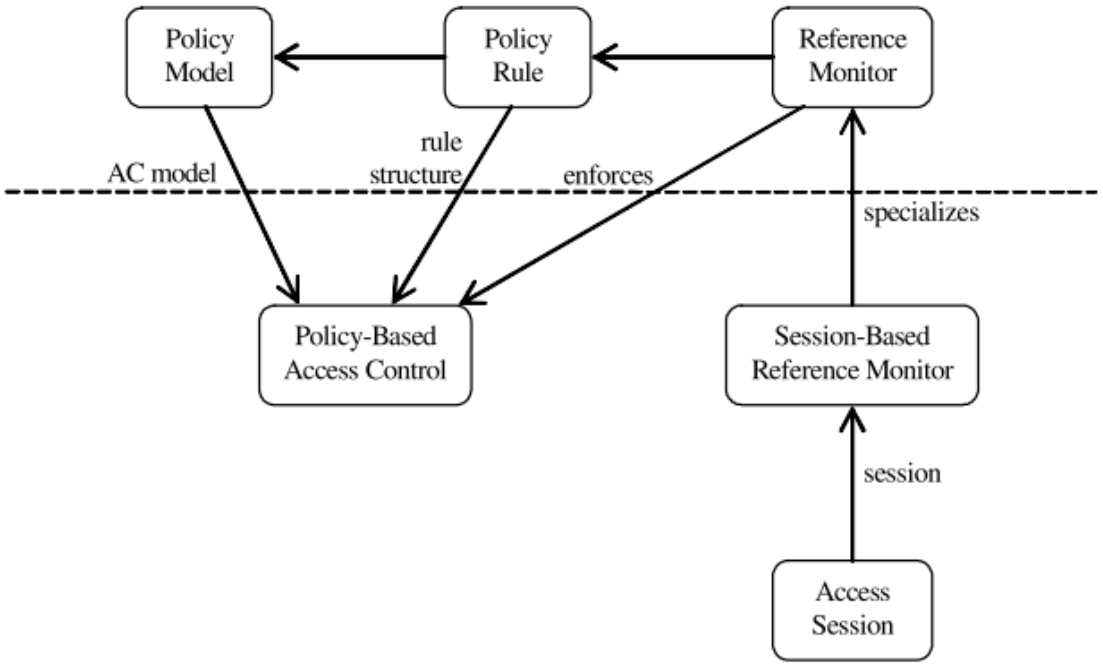

Abbildung 14: Zusammenhang von Zugriffskontrollmodellen als Musterdiagramm. Quelle: [FP+08:41]

3.4.1 Bewertung des Ansatzes

Die Idee des Musterdiagramms löst das Problem, das bisherige Ansätze nicht für Muster geeignet sind. Das größte Problem ist, dass die Semantik der Diagramme nicht beschrieben wird. Durch die Annotation an die Verbindungslinien ähnelt die Darstellung der aus [DF+07]. An diesem Artikel war Fernandez ebenfalls beteiligt, weshalb die Vermutung naheliegt. Dies führt zu der gleichen Schlussfolgerung, dass es an Semantik fehlt und die Diagramme unübersichtlich werden.

Eine sehr gute Neuerung bietet der Artikel mit der Darstellung von Hierarchien durch Abgrenzung der Muster mit gestrichelten Linien. Auf diese Weise können die Muster eingeteilt werden und gegebenenfalls an verschiedene Entwicklerteams delegiert werden. Beispielsweise übernimmt ein Team die Modellierung des Grobentwurfs bis zu einer bestimmten Grenze und ein weiteres Team konkretisiert den Entwurf weiter.

Die beiden Ansätze zur Mustergewinnung sind gut gelungen. Eine ähnliche Anleitung zur Gewinnung von Mustern beschreiben Buschmann et al. in [BMR+96:375], allerdings nicht in dieser konkreten Form. Es könnte hierbei untersucht werden, welche Muster sich als Erweiterung bestehender Zugriffskontrollmodelle, neben den beiden bisher genannten, eignen.

Insgesamt hat der Artikel gute Ansätze, aber diese werden nicht detailliert genug ausgeführt. Der Ansatz von Musterdiagrammen könnte mit semantischer Beschreibung der Beziehungen

geeignet sein, um die gewünschten Alternativen und andere Beziehungen zwischen Mustern für den Softwareentwicklungsprozess zu beschreiben.

3.5 Bewertung der Modellierungsformen von Musterbeziehungen

In den Grundlagen wurden verschiedene Möglichkeiten zur Modellierung von Musterbeziehungen eingegangen und in den vorigen Abschnitten Beispiele aufgezeigt und bewertet. Dieser Abschnitt soll deren Eignung für diese Arbeit bewertet werden.

Das Ziel der Modellierung von Alternativen zwischen Sicherheitsmustern ist es, den Entwurf der Sicherheitsarchitektur zu unterstützten. Die Sammlung von Mustern und eine grobe Kategorisierung, wie in [GHJV95], scheint nicht geeignet zu sein, da hierdurch die Wahl von Mustern nicht unterstützt wird. Die Beziehungen der Muster untereinander zu beschreiben, wie dies in [Zim95] gezeigt wird, stellt nicht dar, welche Alternativen für ein Muster vorhanden sind und ist deshalb für das vorliegende Problem auch nicht ausreichend. Die Idee des Mustersystems aus [BMR+96] hilft bei der Auswahl von Mustern. Für ein Problem auf einer bestimmten Abstraktionsebene werden verschiedene Alternativen angeboten. Allerdings wird der Zusammenhang der Muster untereinander nicht betrachtet und müsste aus den Musterbeschreibungen entnommen werden. Durch Mustersprachen werden nicht nur Lösungen geboten, sondern ebenfalls die auftretenden Probleme aufgezeigt. Für die Probleme wird ein konkreter Ablauf zur Lösung geboten, was bei Mustersystem nicht der Fall ist. Das Zusammenspiel der Muster zur Lösung umfangreicher Probleme ist nicht Teil von Mustersystemen.

Buschmann et al. stellen Mustersequenzen zur Aufstellung von Mustersprachen vor. Eine Menge von Mustersequenzen als Mustersprache enthält potentiell Redundanz, da Mustersequenzen für ähnliche Problemstellungen vermutlich ähnliche Muster enthalten. Dieser Aufwand jedoch reduziert werden, indem die Gemeinsamkeiten der Sequenzen zusammengefasst werden und als Alternative auswählbar sind. In einem abgeschlossenen Gebiet, wie dem für Authentifizierung und Autorisierung, werden Muster zur Feststellung der Authentizität eines Benutzers, Muster, die die Beschreibung von Autorisierungsregeln umfassen, und Muster, die deren Durchsetzung beschreiben, benötigt. Diese unterschiedlichen Muster können als alternativen für eine Software modelliert werden und durch diese Alternativendarstellung die Mustersprache bilden.

Bei der vorgestellten Form einer Mustersprache fehlt allerdings die Beachtung von bestehenden Systemen und somit die Verbindungen zur Implementierungsphase. Dies ist gerade für die betrachtete Fragestellung ein wichtiger Punkt, der ergänzt werden muss. Eine grafische Darstellung der Beziehungen bieten Musterdiagramme. Diese Diagramme sind derzeit noch wenig verbreitet. Zudem bieten die Diagramme keine Formalisierung, die Beziehungen können beliebig annotiert werden. Diagramme, die durch ihre Darstellung direkt Möglichkeiten zur Wahl von Mustern für bestimmte Probleme bieten, wären sinnvoller. Dies soll in Kapitel 5 weiter betrachtet werden und eine mögliche Darstellungsform wird eingeführt.

4 Authentifizierung- und Autorisierungsmuster und deren Umsetzung in bestehendem Sicherheits-Framework

Die erste Frage, die mit dieser Arbeit beantwortet werden soll, betrifft Sicherheitsmuster in Sicherheits-Frameworks und deren Umsetzung. Welche Muster werden von Frameworks umgesetzt und wie werden diese implementiert? Dieses Kapitel soll die aufgestellte Frage beantworten und als Grundlage zur Beantwortung der zweiten Frage dienen. Beispielhaft wird das Spring Security Framework auf ausgewählte Sicherheitsmuster überprüft und die Grundlage für die Modellierung von Abhängigkeiten zwischen den Mustern im nächsten Kapitel gebildet.

Zuerst wird erklärt, wie die Überprüfung des Frameworks auf Muster aussieht und ob eine automatisierte Prüfung notwendig ist oder diese manuell durchgeführt wird. Daraufhin werden ausgewählte Sicherheitsmuster vorgestellt und für jedes dieser Muster wird geprüft, ob es mit dem Spring Security Framework umgesetzt werden kann. Falls das Muster nicht direkt aber durch Implementierung weniger Klassen mit dem Framework umgesetzt werden kann, werden die hierfür notwendigen Erweiterungen aufgezeigt.

4.1 Vorgehen zur Überprüfung des Spring Security Frameworks auf Muster

Das Ziel des Kapitels ist es, das Spring Security Framework auf Sicherheitsentwurfsmuster zu überprüfen. Zur Überprüfung auf Muster bietet sich sowohl eine automatisierte als auch eine manuelle Suche in der Architektur an. Welche dieser Varianten gewählt wird und wie das Vorgehen grob aussieht, wird in diesem Kapitel geklärt.

4.1.1 Automatisierte Mustersuche

Es gibt verschiedene Programme, mit denen eine automatisierte Suche nach Mustern möglich ist. Eine Übersicht über verschiedene Ansätze und genutzte Werkzeuge bieten [PSR03, DZP07]. Die dort vorgestellten Programme müssten allerdings erst um Sicherheitsmuster erweitert werden, sofern dies möglich ist und die Programme überhaupt die Programmiersprache Java unterstützen. Meist wird von den Werkzeugen nur ein geringer Teil der in [GHJV95] vorgestellten Entwurfsmuster für eine Suche unterstützt, unter anderem, weil die verwendeten Bibliotheken keine ausreichende Unterstützung für die zur Suche notwendigen Merkmale bieten.

Vorteile eines automatisierten Vorgehens liegen darin, dass Strukturen und Abläufe zuverlässig gefunden werden, sofern diese korrekt beschrieben wurden. Die Überprüfung wäre zudem schnell durchführbar, nachdem die Muster für das Programm beschrieben wurden. Die Beschreibungen beim automatisierten Vorgehen können des Weiteren in jedem weiteren Framework erneut eingesetzt werden und die Idee, mehrere Frameworks zu betrachten, wäre effizient umsetzbar.

Jedoch hat das Vorgehen ebenfalls Nachteile. Neben den Defiziten der Werkzeuge ist teilweise ein automatisiertes Vorgehen nicht möglich. Manche der betrachteten Muster sind zu abstrakt, so dass eine konkrete Suche nach der Klassenstruktur, wie von den Werkzeugen vorgesehen, nicht möglich ist. Ein Beispiel ist das Policy Enforcement Point-Muster (PEP), das zur Durchsetzung von Richtlinien (Policies) eingesetzt werden kann. Sowohl die Richtlinien als auch die

Komponente zur Durchsetzung müssen zur Implementierung weiter verfeinert werden. Des Weiteren kann eine Rolle eines Musters von mehreren Klassen in der Implementierung übernommen werden; dies wird von den vorgestellten Werkzeugen nicht beachtet. Am Beispiel des PEP, kann die Komponente zur Durchsetzung von vielen interagierenden Klassen umgesetzt werden.

Der Aufwand zur Einarbeitung in ein Werkzeug zur automatisierten Suche nach Mustern steht vermutlich auch nicht im Vergleich zum Zeitgewinn durch die schnellere Suche, da die Anzahl der zu betrachteten Klassen eines Framework gering gegenüber dem eines gesamten Softwaresystems ist. Zudem müssen nicht alle Klassen hinsichtlich der Sicherheitsmuster überprüft werden, da die relevanten Klassen durch den Mustertyp eingegrenzt werden können. Mit relevanten Klassen sind die Klassen gemeint, die das Sicherheitsmuster umsetzten. Um das Beispiel des PEP fortzusetzen, dieser wird weder unter den Klassen zur Authentifizierung noch denen zur Autorisierung zu finden sein und diese Klassen können somit von der Suche dieses Musters ausgeschlossen werden.

Bei dem betrachteten Spring Security Framework werden zudem an vielen Stellen XML-Dateien verwendet. Automatisierte Ansätze können diese nicht auswerten, wodurch Informationen verloren gehen und eine Überprüfung auf Muster erschwert wird.

4.1.2 Manuelle Mustersuche

Eine weitere Möglichkeit Muster im Framework zu finden, ist dieses manuell zu überprüfen. Für dieses Vorgehen spricht die geringe Menge an relevanten Klassen. Wie zuvor dargelegt, kann die Menge der Klassen durch deren Einsatzgebiet bereits eingegrenzt werden. Durch das Wissen über die Nutzung des Frameworks kann die Menge an relevanten Klassen weiter eingeschränkt werden. In Spring Security ist es beispielsweise möglich, Gruppen und Rollen, die zur Umsetzung einer rollenbasierten Zugriffskontrolle benötigt werden, direkt in einer XML-Datei zu konfigurieren. Diese Konfiguration wird von Spring in Java-Klassen übertragen. An dieser Stelle kann angesetzt und durch die Klassen zum Laden der Konfiguration können direkt die relevanten Klassen identifiziert werden.

Jedoch muss die Einschränkung der Klassen je Muster vorgenommen und an die jeweiligen Eigenschaften angepasst werden. Bei ähnlichen Sicherheitsmustern wird diese Einschränkung ähnlich aussehen. Beispielsweise das PEP- und das Authorization Enforcer-Muster dienen beide der Zugriffskontrolle. Diese werden beim Einstiegspunkt zu einer geschützten Ressource zu finden sein.

Sofern das Sicherheitsmuster ein UML- oder OMT-Diagramm oder eine ähnliche Struktur- oder Ablaufbeschreibung aufweist, wird dieses zur Identifikation des Musters genutzt. Jedoch ist eine strukturelle Beschreibung nicht bei allen Sicherheitsmustern vorhanden; das Password Design and Use-Muster aus [SF+06:217] ist beispielsweise nur textuell beschrieben. Bei diesen Mustern würde die textuelle Beschreibung zur Überprüfung genutzt.

Insgesamt ist eine manuelle Prüfung im Vorteil gegenüber dem automatisierten Vorgehen. Deshalb wird die Überprüfung auf Muster im Folgenden manuell durchgeführt. Das konkrete Vorgehen ist, die Muster mit Hilfe von Kompositionsstrukturdiagrammen und Sequenzdiagrammen zu beschreiben. Für das Muster werden die Klassen, die eine Rolle im Muster einnehmen identifiziert. Eine Instanziierung des Musters als Kompositionsstrukturdiagramm ist hierbei das Ziel. Damit einhergehend wird die Interaktion der Komponenten des Frameworks betrachtet und mit

dem Sequenzdiagramm verglichen. Stimmt diese ebenfalls überein, gilt das Muster als von dem Framework umgesetzt.

4.2 Überprüfung auf Muster für Richtlinien

Die Autorisierungs- und Authentifizierungsmuster werden aus [SF+06, FH06, SPR04] und verschiedenen Artikeln, auf die speziell verwiesen wird, entnommen. Zu Beginn wird auf Muster eingegangen, die zur Definition der Zugriffskontrolle respektive zur Durchsetzung der Zugriffskontrolle genutzt werden können. Darauf folgen Muster, die zur Aufteilung und sinnvollen Strukturierung der anfallenden Daten genutzt werden können, beispielsweise zur Trennung von Zugriffskontrollregeln und deren Durchsetzung. Abschließend werden Muster betrachtet, die sich auf die Identitätsfeststellung beziehen. Es können nicht alle bekannten Muster betrachtet werden, deswegen wird die Auswahl auf in der Praxis übliche Muster beschränkt.

Jedes Sicherheitsmuster wird in einem eigenen Abschnitt betrachtet und jeder Abschnitt beginnt mit einer Zusammenfassung des Musters, wobei die Struktur mit einem Kompositionsstrukturdiagramm und den jeweiligen Rollen im Muster erklärt wird. Darauf folgen die Überprüfung mit einer Einführung in relevante Klassen des Spring Frameworks und gegebenenfalls ein bewährter Weg, um das Muster mit Spring Security umzusetzen. Abschließend wird entweder durch eine Instanziierung der Komposition, die am Anfang des Kapitels vorgestellt wurde, der Zusammenhang zwischen Muster und der Implementierung in Spring hergestellt oder erklärt warum diese nicht möglich ist.

Sollte es zu einem Muster zusätzliche Varianten geben, die nur eine geringe Änderung der Struktur vorsehen, so werden diese in einem eigenen Unterkapitel betrachtet. Wenn ein Muster in Spring nicht gefunden wird, aber mit einfachen Erweiterungen des Frameworks eine Umsetzung möglich ist, so werden diese ebenfalls in einem Unterkapitel genauer betrachtet.

4.2.1 Rollenbasierte Zugriffskontrolle

Die Idee bei der rollenbasierten Zugriffskontrolle (engl. Role-Based Access Control, RBAC) ist es, Rollen in der Menge der Benutzer zu identifizieren und diesen Rollen Rechte zuzuweisen. Benutzer erhalten ihre Rechten über die Rolle, die sie im System einnehmen. Hierdurch soll die Vergabe von Rechten vereinfacht werden. Einem geschützten Objekt müssen somit nicht für jeden Benutzer die Zugriffsrechte vergeben werden.

Die übliche Struktur der rollenbasierten Zugriffskontrolle ist in Abbildung 15 als Kompositionsstrukturdiagramm dargestellt und besteht aus einem Benutzer, Rollen, dem zu schützenden Objekt und Rechten. Der Benutzer kann mehrere Rollen besitzen und einer Rolle können mehrere Benutzer zugewiesen sein. Einer Rolle können für geschützte Objekte verschiedene Rechte gegeben werden, wobei umgekehrt einem geschützten Objekt mehrere Rollen mit ihren Rechten zugewiesen werden können. Eine tiefergehende Beschreibung der rollenbasierten Zugriffskontrolle kann in [SF+06:249, FP01:5] gefunden werden.

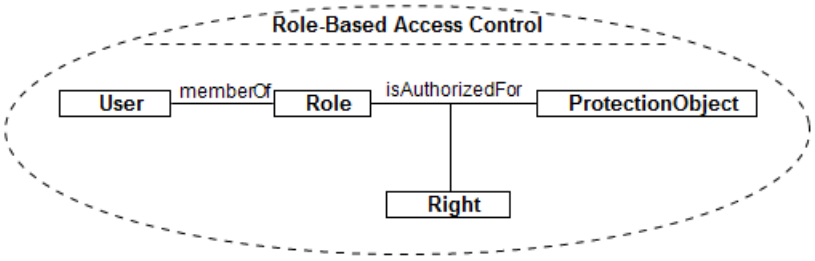

Abbildung 15: Kompositionsstrukturdiagramm des RBAC-Musters

Überprüfung

Zur Überprüfung wird ein ähnliches Konzept des Spring Security Frameworks betrachtet. Das Konzept von Benutzern und Rollen wird in Spring durch *User* und deren *Authorities* repräsentiert. Den *Authorities* können bestimmte Rechte in der Anwendung zugewiesen werden. Als Ansatzpunkt wird die Definition von *User* und *Authorities* in einer XML-Konfigurationsdatei betrachtet. Diese wird beispielsweise für Testzwecke angeboten. Es sei darauf hingewiesen, dass es ebenfalls möglich ist, eine Anbindung an Benutzerspeicher, wie beispielsweise eine Datenbank, einen LDAP- oder CAS-Server, zu nutzen. Bei dem Beispiel der XML-Konfiguration lassen sich allerdings, die für das Muster interessanten Klassen, zurückverfolgen. Eine Konfiguration könnte wie in Quellcodebeispiel 8 aussehen. Eine Zeile des Beispiels repräsentiert die Definition eines *User*-Objekts mit seinem Passwort und *Authorities* als kommaseparierte Auflistung.

```
<user name="student1" password="stud1" authorities="ROLE_STUDENT" />
<user name="admin" password="adm" authorities="ROLE_ADMINISTRATOR" />
<user name="lecturer" password="lect" authorities="ROLE_LECTURER" />
```

Quellcodebeispiel 8: Spring Konfiguration von Benutzern und Rollen

Spring lädt die XML-Konfiguration über sogenannte *BeanDefinitionParser* in denen das XML ausgelesen und die jeweiligen Klassen durch das Erbauer-Entwurfsmuster [GHJV95] erzeugt werden. Die obigen Elemente werden durch den *UserServiceBeanDefinitionParser* verarbeitet. Für jedes Benutzerelement wird ein *User*-Objekt und für jedes *authorities*-Element aus der kommaseparierten Auflistung ein Objekt vom Typ *GrantedAuthority* erzeugt; die *GrantedAuthority*-Objekte werden beim jeweiligen *User*-Objekt als Variable gespeichert.

Die Definition eines geschützten Objekts in Spring Security kann je nach Anwendungstyp und Ebene der Betrachtung unterschiedlich ausfallen. Beispielsweise könnte beim Zugriff auf ein persistiertes Objekt auf Quellcodeebene oder bei einer Web-Anwendung bereits beim Zugriff auf die Anwendungsinstanz mittels einer URL der Zugriff kontrolliert werden.

Wird die Anwendung nach dem REST-Paradigma (siehe Kapitel 2.6) umgesetzt, jede Ressource ist somit unter einer einheitlichen URL ansprechbar und stellt verschiedene Operationen bereit, oder können URLs zum Schutz genutzt werden, so bietet sich die Nutzung der Absicherung von URLs an, ansonsten können ebenfalls Methoden im Quellcode vor ungewolltem Zugriff geschützt werden.

In Spring Securiy ist der Schutz von URLs per Konfiguration möglich, wie in Quellcodebeispiel 9 zu sehen. Das Beispiel zeigt den Einsatz der XML-Elemente *http* und *intercept-url* von Spring Security. In einem *http*-Element können verschiedene Konfigurationen für HTTP-Anfragen vorgenommen werden. Im Beispiel wird durch das *intercept-url*-Element der Zugriff auf eine URL eingeschränkt. Die URL wird im *pattern*-Attribut definiert, ist relativ zum Anwendungs-Servlet zu verstehen und kann durch ein Muster mit Wildcards beschrieben werden. Durch das *access*-Attribut wird angegeben, wer Zugriff auf die URL hat. Zwischen den Klammern können die Rollen definiert werden, die Zugriff auf die URL haben sollen.

```
<http>
    <intercept-url pattern=”/reservation/*/delete”
access=”hasRole(PERM_DELETE_RESERVATION)”/>
</http>
```

Quellcodebeispiel 9: Schützen einer URL per Konfiguration

Durch diese Einstellung wird Spring alle Anfragen an URLs, die dem Ausdruck unter *pattern* entsprechen, überprüfen. Das Abfangen von Nachrichten zur Zugriffskontrolle stellt ein weiteres Entwurfsmuster dar, welches in Kapitel 4.3.1 beschrieben wird. In dem konkreten Beispiel werden Anfragen zum Löschen einer Reservierung abgefangen und es wird überprüft, ob der Anfragende die Berechtigung zum Löschen von Reservierungen besitzt.

Neben dem Schützen von URLs bei Web-Anwendungen kann mit Spring auch auf Quellcodeebene eine Methode mittels Annotationen als geschütztes Objekt markiert werden. Spring bietet hierfür eine Reihe von Annotationen an. Der Standard aus dem Java Specification Request (JSR) 250 wird beispielsweise durch das Spring Framework unterstützt.

Der JSR-250 beschreibt unter anderem die Annotationen *@RolesAllowed*, *@PermitAll* und @DenyAll für Methoden oder Klassen. Die letzten beiden Annotationen können genutzt werden, um den Zugriff auf eine Methode allgemein zu erlauben oder zu verbieten. Gemäß dem Standard kann der *RolesAllowed*-Annotationen direkt eine oder mehrere Rollen als Zeichenkette übergeben werden. Ist einem Benutzer eine der aufgeführten Rollen zugewiesen, so darf dieser auf die Methode der Klasse zugreifen. Es gibt weitere Spring-Annotationen, die denselben Zweck erfüllen können und teilweise einen größeren Funktionsumfang haben, auf diese wird in späteren Kapiteln eingegangen.

In [Wi11] wird auf die Möglichkeit hingewiesen, dass statt einer Rolle auch eine Berechtigung (*Permission*) angegeben werden kann. In diesem Fall wird eine Rolle mit verschiedenen Berechtigungen verknüpft. Um flexibler auf Änderungen der Berechtigungsstruktur der Anwendung reagieren zu können, wird die Nutzung von Berechtigungen statt Rollen empfohlen. Die Zuweisung von Berechtigung zu Rollen kann an einer zentralen Stelle verwaltet werden, wodurch die Übersicht über die Berechtigungen erhalten bleibt. Zudem ist bei Änderungen der Berechtigungsstruktur kein Umschreiben und Neukompilieren der Anwendung notwendig.

Quellcodebeispiel 10 zeigt die *RolesAllowed*-Annotation mit einer Berechtigung zur Absicherung der Methode zum Löschen von Reservierungen eines Arbeitsplatzes.

```
@RolesAllowed("PERM_DELETE_RESERVATION")
public void delete(Reservation reservation) {
…
}
```

Quellcodebeispiel 10: Spring-Annotation *RolesAllowed* zur Zugriffskontrolle vor Methodenausführung

Ein Anwender kann auf diese Methode zugreifen, sofern dieser die Berechtigung zum Löschen von Reservierungen direkt oder über eine Rolle hat. Quellcodebeispiel 11 zeigt wie der Administratorrolle die Berechtigung zum Löschen von Reservierungen per XML-Konfiguration zugewiesen werden kann.

```
<bean id="rightsToRoles"
class="oss.access.hierarchicalroles.RoleHierarchyImpl">
    <property name="hierarchy">
        ROLE_ADMINISTRATOR > PERM_DELETE_RESERVATION
        …
    </property>
</bean>
```

Quellcodebeispiel 11: Beispiel der Zuweisung von Rechten zu Rollen in Spring Security

Die Auswertung der Zugriffskontrolle wird für die vorgestellten Annotationen von der Klasse *Jsr250Voter* ausgeführt. Die Parameter der Annotationen werden in der *ConfigAttribute*-Klasse dem *Voter* bereitgestellt. In der *Jsr250Voter*-Klasse wird überprüft, ob dem Subjekt eine der benötigten Rollen oder Berechtigungen zugewiesen ist.

Überprüfungsergebnis

Der Zusammenhang der Spring-Klassen und deren Rolle in der rollenbasierten Zugriffskontrolle wird durch die Instanziierung der zuvor dargestellten Kollaboration im Kompositionsstrukturdiagramm in Abbildung 16 veranschaulicht. Das *User*-Objekt entspricht dem gleichnamigen Objekt im Muster. Die Rollen des Benutzers werden durch die Klasse *GrantedAuthority* repräsentiert. Ein geschütztes Objekt ist entweder eine Methode oder eine URL, welche dem *ProtectionObject* aus der rollenbasierten Zugriffskontrolle gleicht. Die gesicherten Objekte haben verschiedene Merkmale zugewiesen, welche in Spring als *ConfigAttribute* gespeichert und zu dem gesicherten Objekt in Verbindung gebracht werden. Sie entsprechen bei der Standardkonfiguration Rechten oder Rollen bei dem RBAC-Muster. Der Zusammenhang zwischen *GrantedAuthorities* und *ConfigAttributes* wird von dem *Jsr250Voter* überprüft und besteht, falls die *GrantedAuthority* des Benutzers und das *ConfigAttribute* des geschützten Objekts übereinstimmen.

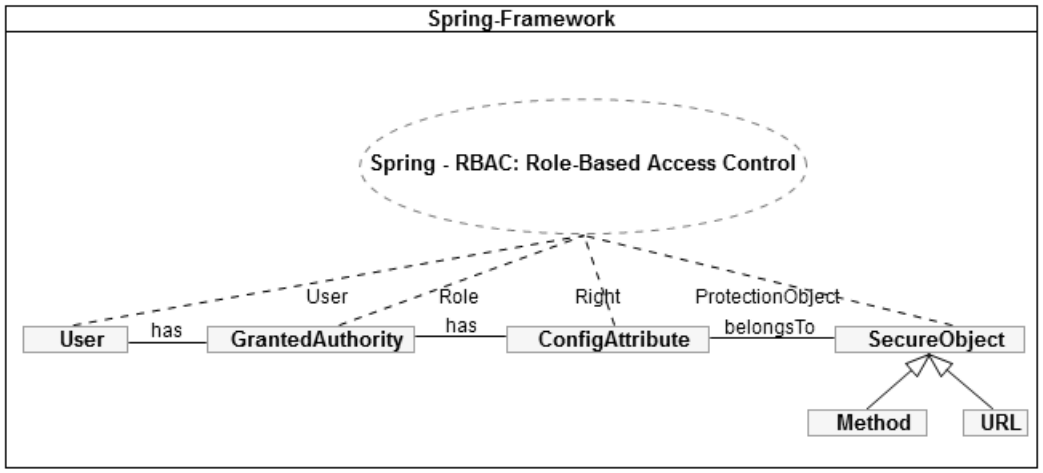

Abbildung 16: Instanziierung des RBAC-Musters durch das Spring Framework

Diese Instanziierung zeigt, dass Spring das RBAC-Muster unterstützt und weitere Varianten der RBAC überprüft werden können. Festzuhalten ist bei dieser Instanziierung allerdings, dass die Rollen in Spring Security nicht für eine Anwendung eingeschränkt oder vorgegeben werden können. Es kann keine Menge an Rollen in einer Anwendung definiert werden, aus welcher gewählt werden kann. Hingegen kann an jeder Methode oder in jeder Konfiguration eine beliebige Zeichenkette mit vorangestelltem *ROLE_* eingeführt werden, wodurch diese als Rolle interpretiert wird.

4.2.2 Varianten der Rollenbasierten Zugriffskontrolle

In [SF+06:251] werden vier weitere Varianten der RBAC eingeführt. Eine Variante ist die Erweiterung der Rollen um das Kompositum-Entwurfsmuster aus [GHJV95]. Des Weiteren wird die Einführung einer Administrationsrolle und von Benutzergruppen aufgeführt. Die vierte Variation ist der Einsatz einer Sitzung.

Variante: Rollen um Kompositum-Entwurfsmuster erweitern

Diese Variation, entnommen aus [SF+06:251], ermöglicht das Strukturieren der Rollen, indem mehrere Rollen zu einer übergeordneten Rolle zusammengefasst werden können. Beispielsweise könnte die Tutorenrolle in der Arbeitsplatzsuche auf der Studentenrolle aufbauen und die Rechte dieser Rolle übernehmen. Einem Tutor müssen somit nicht zwei Rollen (Student und Tutor) zugewiesen werden, sondern er erhält die Studenten Rolle durch die Tutorenrolle, die wiederum weitere Berechtigungen enthalten kann. Die Struktur des Musters mit den relevanten Klassen ist in Abbildung 17 dargestellt.

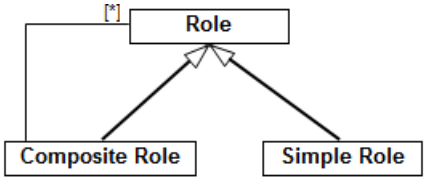

Abbildung 17: Rolle erweitert durch Composite-Entwurfsmuster

Wie bereits im vorigen Kapitel erläutert, bietet Spring die Möglichkeit Berechtigungen Rollen zuzuweisen, siehe Quellcodebeispiel 11. Auf dieselbe Art kann auch eine Rolle einer anderen

Rolle zugewiesen werden und so eine Hierarchie, wie bei einem Kompositum, aufgebaut werden.

```
<bean id="rightsToRoles"
class="oss.access.hierarchicalroles.RoleHierarchyImpl">
    <property name="hierarchy">
        ROLE_TUTOR > ROLE_STUDENT
        …
    </property>
</bean>
```

Quellcodebeispiel 12: Definition von Rollenhierarchien in einer Spring Security Konfigurationsdatei

Quellcodebeispiel 12 zeigt eine solche Zuweisung. Der Tutor übernimmt in dem Beispiel alle Berechtigungen der Rolle Student. Diese Variante der RBAC wird von Spring somit unterstützt.

Variante: Administratorrolle und -rechte

Die Administratorrolle als Variante der RBAC stammt aus [SF+06:251]. Die Idee hinter der Administratorrolle ist, dass es eine Rolle geben sollte, die andere Rollen aktivieren und administrieren kann. Als Erweiterung der Struktur sind eine zusätzliche Administratorrolle und Administratorrechte angedacht. Abbildung 18 zeigt den relevanten Ausschnitt aus der RBAC-Struktur.

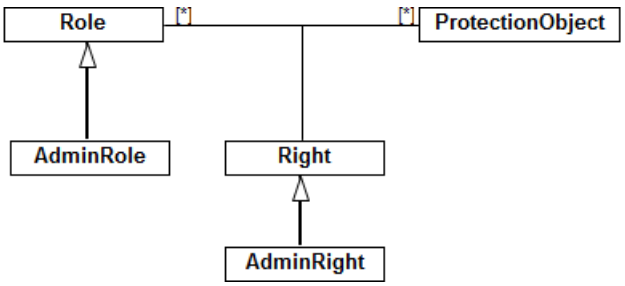

Abbildung 18: Administratorrolle und -recht als Erweiterung der RBAC

In Spring Security gibt es keine vordefinierten Rollen oder Rechte und zudem ist die Administration der Benutzer nicht Bestandteil des Frameworks. Um dieses Muster umsetzen zu können, muss es eine eigene Software zur Verwaltung der Rollen und Rechte geben. Eine Möglichkeit wäre der Einsatz eines LDAP-Verzeichnisses. Auf dieses haben Administratoren Zugriff und können die Benutzer verwalten.

Bietet der Benutzerspeicher eine solche Funktion nicht an, so müsste es eine eigene Software zur Verwaltung der Rollen und Rechte geben und diese müsste mit Spring Security abgesichert werden. Jede relevante Funktion, wie das Erzeugen oder Aktivieren von Benutzern, wird mit einem Administratorrecht versehen. Die Administratorrechte werden der Administratorrolle zugewiesen. Über diesen Weg könnte das Muster umgesetzt werden, allerdings ist der Aufwand relativ hoch. Diese Variante wird somit nicht direkt von Spring Security umgesetzt, ein Benutzerspeicher kann dies jedoch anbieten.

Variante: Benutzergruppen

Eine weitere Variante betrifft die Strukturierung der Benutzer in Gruppen. Eine Gruppe besteht aus mehreren Benutzern und die Gruppe kann verschiedenen Rollen zugewiesen sein. Die erweiterte Struktur der relevanten Klassen des RBAC-Musters zeigt Abbildung 19 und diese wird ebenfalls in [SF+06:251] genauer beschrieben.

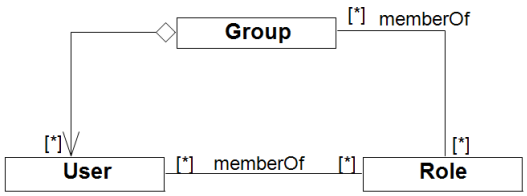

Abbildung 19: Erweiterung der RBAC durch Gruppen zur Strukturierung der Benutzer

Die Spring-Benutzerklasse steht in keiner Verbindung zu anderen Klassen, die eine Gruppierung erlauben würde, wie die Betrachtung der API ergibt. Mit Spring kann diese Struktur somit nicht direkt umgesetzt werden. Wenn jedoch das Benutzerverzeichnis eine Strukturierung nach Gruppen erlaubt, beispielsweise ein LDAP-Verzeichnis, und diese Spring Security zur Verfügung stellt, könnte die Gruppe des Benutzers als Attribut zur Umsetzung des Musters genutzt werden. Das nächste Kapitel beschreibt, wie zur Zugriffskontrolle auch Attribute des Benutzers genutzt werden können, um beispielsweise den Zugriff zu erlauben, wenn die Gruppe die Berechtigung besitzt. Mit dieser Erweiterung wäre es möglich diese Variante der RBAC mit Spring Security umzusetzen.

Variante: Benutzersitzungen zur Einschränkung von Rollen zur Laufzeit

Die letzte Variante aus [SF+06:251] ist die Einführung einer Sitzung auf der ein Benutzer arbeitet. Eine Sitzung gehört zu einem Benutzer und ist mit einer Teilmenge aller Rollen des Benutzers verknüpft. Auf diese Weise können zur Laufzeit die Berechtigungen des Benutzers beschränkt werden. Der Zusammenhang der verschiedenen Klassen der RBAC ist auf Abbildung 20 dargestellt.

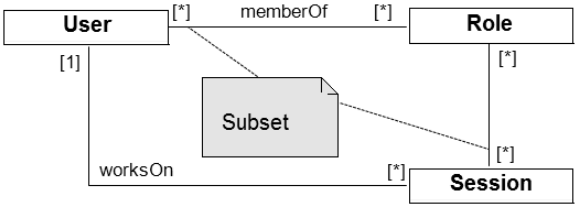

Abbildung 20: Sitzungen erweitern das RBAC-Muster und ermöglichen das Einschränken der Rollen eines Benutzers zur Laufzeit

Der aktuelle Benutzer wird in einem *SecurityContext* für seine aktuelle Sitzung gespeichert, allerdings ist nicht vorgesehen, dass die *Authorities* eines Benutzers in einem Kontext verändert werden. Durch ein Eingreifen beim Laden des Benutzers und seiner *Authorities*, könnten andere Rollen geladen werden. Beispielsweise ein eigener *AuthenticationManager*, dieser wird genauer in Kapitel 4.3.3 eingeführt, könnte diese Aufgabe an Hand von zusätzlichen Parametern beim Anmelden übernehmen. Als Parameter müsste übergeben werden, welche Rollen für die aktuelle Sitzung geladen werden sollen. Dies wäre eine Möglichkeit dieses Muster zu implementieren und für einen Benutzer bei der Authentifizierung die Rollen für die Sitzung einzuschränken.

4.2.3 Attributbasierte Zugriffskontrolle

Eine weitere Form der Zugriffskontrolle, die über die Möglichkeiten der rollenbasierten Zugriffskontrolle hinausgeht, ist die attributbasierte Zugriffskontrolle (engl. Attribute-Based Access Control, ABAC). Bei dieser wird die Entscheidung über den Zugriff nicht nur auf Grund der Rollen des Benutzers durchgeführt, sondern alle Attribute eines Benutzers, auch der geschützten Ressource und allgemeiner Umgebungsvariablen können bei der Entscheidung mit einbezogen werden.

Attribute der geschützten Ressourcen können der Name des Besitzers oder der Standort sein. Beispiele für Attribute des Benutzers sind sein Name, sein Geburtsdatum oder seine Rollen. Mit Rollen als Attribut des Benutzers, könnte je Rolle eine Berechtigung für eine Ressource vergeben werden, weswegen das RBAC-Muster als Verfeinerung des ABAC-Musters betrachtet werden kann.

ABAC wird in [YT05] vorgestellt und sieht die Absicherung von Operationen eines Webservice vor. Im Folgenden wird eine allgemeine Form des ABAC-Musters betrachtet, das heißt die Absicherung von Operationen eines Webservice wird auf die Absicherung von jeglichen Ressourcen verallgemeinert. Eine Ressource kann wie bei dem RBAC-Muster alles schützenswerte sein, beispielsweise ein Gebäude oder bei Web-Anwendungen URLs. Die resultierende allgemeine Struktur ist auf Abbildung 21 zu sehen.

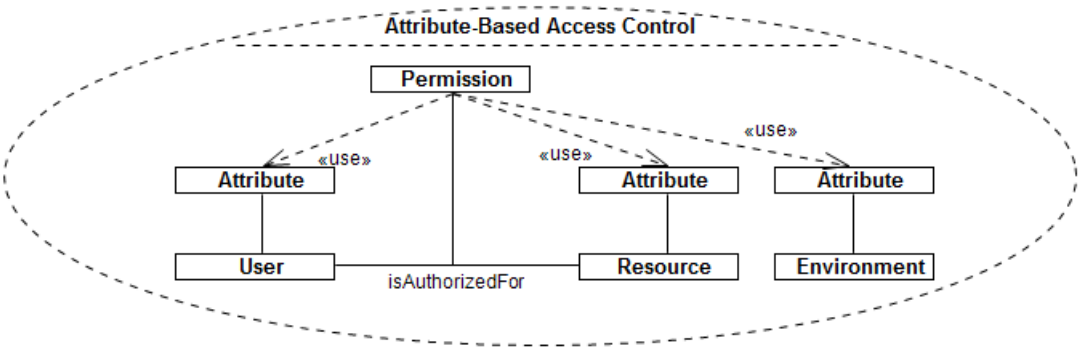

Abbildung 21: Struktur der attributbasierten Zugriffskontrolle

Überprüfung

In Kapitel 4.2.1 über RBAC wurde bereits das Absichern von URLs und Methoden mit dem Spring Security Framework gezeigt, allerdings mit dem Fokus auf der Umsetzung des JSR-250 zur Benutzung von Rollen. Die Bezeichnung *pattern* in Quellcodebeispiel 9 lässt bereits auf einen größeren Funktionsumfang schließen. Neben der Nutzung von Rollen als *pattern* ist es in Spring möglich, die Spring Expression Language (Spring EL) für komplexere Berechtigungsaussagen zu nutzen.

Um die Spring EL auch auf Methodenebene nutzen zu können, bietet Spring eigene Annotationen an. Im Folgenden wird genauer auf die Annotation *@PreAuthorize*, die eine Zugriffskontrolle vor Methodenausführung erzwingt, und *@PostFilter*, durch die nach Methodenaufruf die Berechtigung geprüft werden kann, eingegangen. Letztere Annotation ist für Methoden gedacht, die Geschäftsobjekte zurückgeben und bietet die Möglichkeit die Autorisierung von Objekte als Rückgabewert zu prüfen.

Quellcodebeispiel 9 zeigt die Nutzung des Ausdrucks *hasRole*, welche erfüllt ist, sobald der aktuelle Benutzer diese Rolle oder Berechtigung besitzt. Neben dem *hasRole*-Ausdruck bietet Spring EL beispielsweise die Ausdrücke *isAuthenticated*, *isRememberMe* oder *hasIpAddress*.

Der Ausdruck *isRememberMe* erlaubt den Zugriff, falls der aktuelle Benutzer sich in seiner aktiven Sitzung nicht direkt mit seinen Zugangsdaten authentifiziert hat, aber beispielsweise mittels eines *Cookie* in der Web-Anwendungen authentifiziert wurde. Hat der Benutzer sich mit seinen Zugangsdaten authentifiziert, kann er auf Ressourcen, die durch *isAuthenticated* geschützt sind, zugreifen. Falls nur bestimmten IP-Adressen der Zugriff gewährt werden soll, kann der *hasIpAddress*-Ausdruck eingesetzt werden, wobei ganze Bereiche ausgewählt werden können.

Neben der Nutzung von vorgefertigten Ausdrücken kann auch auf die Attribute des authentifizierten Benutzers zugegriffen werden und über die Attribute Ausdrücke gebildet werden. Bei dem Schutz von Methoden mit Annotationen kann zudem auf die Attribute der Methodenparameter zugegriffen werden. Eine solche Aussage und die Verknüpfung einzelner Ausdrücke mit booleschen-Operatoren zeigt Quellcodebeispiel 13.

Das Quellcodebeispiel sagt aus, dass der Benutzer auf die Methode zum Speichern von Reservierungen zugreifen darf, sofern er die Berechtigung *PERM_SAVE_RESERVATION* hat oder seine Kennung gleich dem Attribut *ownerId* des Reservierungsobjekts ist bzw. er der Besitzer der Reservierung ist. Auf die Attribute greift Spring gemäß dem JavaBeans-Standard [Ora11a] per *Getter*-/*Setter*-Methode zu. Das Doppelkreuz # zeigt dabei an, dass auf einen Methodenparameter zugegriffen werden soll.

```
@PreAuthorize("principal.id == #reservation.ownerId or
hasRole(PERM_SAVE_RESERVATION)")
public void saveReservation(Reservation reservation) {
}
```

Quellcodebeispiel 13: Aussage über das Benutzerobjekts und einen Methodenparameter bei der Methodenabsicherung

Zusätzlich zur Zugriffskontrolle vor der Ausführung der Methode bietet Spring Annotationen zur Filterung des Rückgabewerts einer Methode. Eine solche Annotation ist *@PostFilter*. Zu den zuvor genannten Spring-EL Elementen kommt bei dieser Annotation die Variable *filterObject* hinzu. Hierbei wird angenommen, dass der Rückgabewert bzw. die Parameter einem geschützten Geschäftsobjekt entsprechen, welches mit der URL oder Methode angefragt werden soll. Ist der Rückgabewert eine Menge, so repräsentiert die Variable ein Element aus der Menge und ermöglicht den Zugriff auf Attribute der zurückgegebenen Objekte. Der Einsatz dieser Filtermethode ist auf Quellcodebeispiel 14 zu sehen.

Nach der Ausführung der Methode *getAllReservations* wird für jede Reservierung der Ausdruck ausgewertet und falls dieser nicht zutrifft, wird das Element aus der Liste entfernt. Mit dem Ausdruck wird überprüft, ob der aktuelle Benutzer der Besitzer der Reservierung ist oder ob er die Berechtigung besitzt alle Reservierungen zu sehen. Ist er nicht der Besitzer und hat auch nicht die notwendigen Berechtigungen, so hat er keinen Zugriff auf die Reservierung und diese wird aus der Menge entfernt. Bei einer Manipulation durch die Methode, muss gewährleistet werden, dass die Manipulation bei fehlender Berechtigung, nicht durchgeführt wird.

```
@PostFilter("(principal.id == filterObject.ownerId) or
hasRole(PERM_GET_ALL_RESERVATIONS)")
public Collection<Reservation> getAllReservations() {
}
```

Quellcodebeispiel 14: Annotation zur Filterung von Objekten im Rückgabewert einer Methode

Überprüfungsergebnis

Durch die vorgestellten Möglichkeiten des JSR-250 und der Spring-EL kann nicht auf Attribute der Umgebung zugegriffen werden. Attribute des geschützten Objekts können abgefragt werden, falls diese als Parameter übergeben oder in einer Liste zurückgegeben werden. Bei einem Ausdruck über die Attribute eines Parameters müsste jedoch zusätzlich überprüft werden, ob die gesetzten Werte des übergebenen Objekts mit denen der persistierten Version übereinstimmen, da korrumpierte Werte von außen übergeben werden könnten. Somit ist nur ein Teil des ABAC-Muster im Spring Security Framework zu finden und es wird nicht direkt umgesetzt.

Erweiterungen des Spring Frameworks für ABAC

In diesem Abschnitt soll gezeigt werden, was nötig ist, um das ABAC-Muster mit Spring Security umzusetzen. Auf Grund des modularen Aufbaus und der guten Struktur des Spring Frameworks, ist der erforderliche Aufwand relativ gering. Das Vorgehen wird beispielsweise in [Wi11] vorgestellt.

Überprüfung

Zusätzlich zu den bereits vorgestellten Sprachelementen der Spring-EL, kann der Ausdruck *hasPermission* eingesetzt werden. Dieser Ausdruck nimmt zwei Parameter entgegen, ein Objekt und die Berechtigung, die beim Zugriff geprüft werden soll. Bei dem Ausdruck *hasPermission* wird angenommen, dass mit der URL oder Methode ein geschütztes Geschäftsobjekt angesprochen wird und dieses an die Methode übergeben wird. Statt des geschützten Objekts kann auch seine Kennung und sein Typ angegeben werden. Sollte sich hinter der URL kein geschütztes Geschäftsobjekt verbergen, sondern die URL das geschützte Objekt sein, so kann für die zu prüfende Berechtigung auch ein passender Wert genutzt werden, beispielweise *PERM_SAVE_RESERVATION*.

Eine beispielhafte Anwendung zeigt Quellcodebeispiel 15. Durch das Beispiel wird Spring Security dazu angehalten, die Berechtigung des Benutzers zum Lesen des Objekts mit einem *PermissionEvaluator* zu prüfen. Die Überprüfung geschieht, wegen der Nutzung der *@PostAuthorize*-Annotation, nach dem Methodenaufruf. Da die Berechtigung nach dem Methodenaufruf geprüft wird, muss bei fehlender Berechtigung die Manipulation der Methode rückgängig gemacht werden. Dies kann beispielsweise durch Transaktionen geschehen, für welche das Spring Framework ebenfalls Unterstützung anbietet

.

```
@PostAuthorize("hasPermission(filterObject, 'save')")
public Reservation saveReservation(Reservation reservation) {
    …
}
```

Quellcodebeispiel 15: Benutzung des Ausdrucks *hasPermission* der Spring-EL

Die Schnittstelle *PermissionEvaluator* kann zur Auswertung des Ausdrucks und somit zur Überprüfung der Berechtigung selbst implementiert werden. Die Schnittstelle hat zwei Methoden, die einen Wahrheitswert zurückgeben, je nachdem ob der Zugriff gewährt wird oder nicht. Der Unterschied zwischen den beiden zu implementierenden Methoden besteht darin, dass entweder das geschützte Objekt oder nur die Kennung und der Typ des geschützten Objekts als Parameter übergeben wird. Im obigen Beispiel wird das Objekt selbst übergeben.

Da die Klasse selbst implementiert wird, kann über die Berechtigung an Hand beliebiger Merkmale und Attribute entschieden werden. Es können Umgebungsvariablen abgefragt, die

Attribute des Benutzers ausgelesen und verglichen oder auf das geschützte Objekt und seine Attribute zugegriffen werden. Richtlinien, wie sie für die attributbasierte Zugriffskontrolle in [YT05] beschrieben werden, können auf diese Weise programmatisch umgesetzt werden.

Überprüfungsergebnis

Durch diese Erweiterung des Spring Security Frameworks ist eine Instanziierung der Kollaboration, wie in Abbildung 22 dargestellt möglich. Der Benutzer wird, wie bei dem RBAC-Muster, durch die *User*-Klasse repräsentiert. Die geschützte Ressource ist ebenfalls wie zuvor entweder eine URL oder eine Methode. Hinzu kommt, dass auf das konkret geschützte Geschäftsobjekt bei der Evaluierung der Richtlinie ebenfalls zugegriffen werden kann. Die Umgebungsvariablen werden in [YT05] nicht allzu genau umschrieben, allerdings sollte auf die vorgestellte Weise auf alle nötigen Zusatzinformationen zugegriffen werden können. Je nach Anwendung nehmen verschiedene Klassen die Rolle *Environment* ein, beispielsweise die Java eigene *Date*-Klasse, für ein Datum und eine Uhrzeit, oder eine *Location*-Klasse, falls der Standort der Anwendung in einer Richtlinie benutzt wird. Die Attribute werden durch Java-Klassenattribute repräsentiert und sind beispielsweise nach dem JavaBeans-Standard abrufbar. Richtlinien können im Spring Security Framework durch eine Implementierung eines *PermissionEvaluators* umgesetzt werden. Die Ausführung des Codes berechnet die konkrete Berechtigung eines Benutzers für eine bestimmte geschützte Ressource.

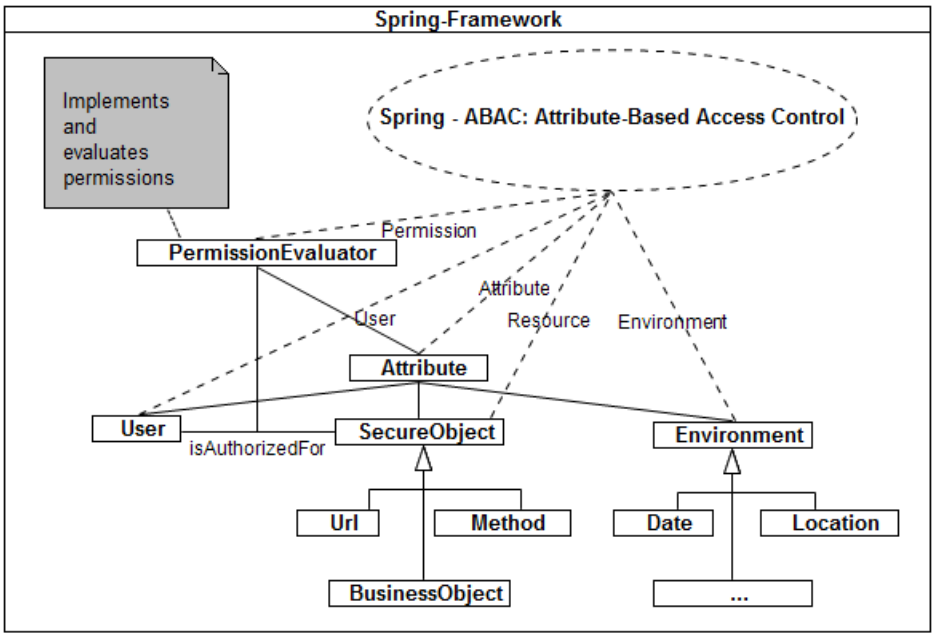

Abbildung 22: Instanziierung des ABAC-Musters durch Spring Security

Mit der gezeigten Erweiterung des Frameworks ist es somit möglich das ABAC-Muster zu nutzen. Die Erweiterung umfasst das Erzeugen einer neuen Klasse und eine Konfiguration, dass diese genutzt werden soll. Der Aufwand für die Erweiterung ist relativ gering. Allerdings muss bei einer Änderung der Richtlinien die Anwendung neukompiliert und ausgeliefert werden, wodurch zusätzlicher Aufwand entstehen kann. Die Verwendung der eXtensible Access Control Markup Language (XACML) wäre eine Möglichkeit dies zu umgehen. Hierfür müsste ein entsprechender *PermissionEvaluator*, der XACML interpretieren und entsprechend die Berechtigung berechnen kann, implementiert werden oder es könnte ein bestehendes Framework zur Auswertung integriert werden, beispielsweise die Open Source Implementierung von Sun Microsystem Inc. [Sun04].

4.2.4 Identitätsbasierte Zugriffskontrolle

Die identitätsbasierte Zugriffskontrolle (engl. Identity-Based Access Control, IBAC) wird in [YT05] kurz beschrieben. In [SF+06:245] wird sie als Konkretisierung des dort vorgestellten Autorisierungsmusters vorgestellt. Das Autorisierungsmuster hat dieselbe Struktur, wie die IBAC, allerdings wird das Subjekt abstrakt verstanden, wohingegen bei der IBAC mit Subjekt eine konkrete Identität gemeint ist. Das Muster sieht vor, dass die Berechtigungen je Anwender für eine bestimmte Zugriffsart, damit ist beispielsweise das Lesen, Schreiben oder Löschen gemeint, vergeben wird. Das Subjekt ist somit direkt mit dem zu schützenden Objekt verknüpft, wobei diese Verknüpfung mit einer Berechtigung für eine bestimmte Aktion verbunden ist. Die Struktur sieht wie folgt aus:

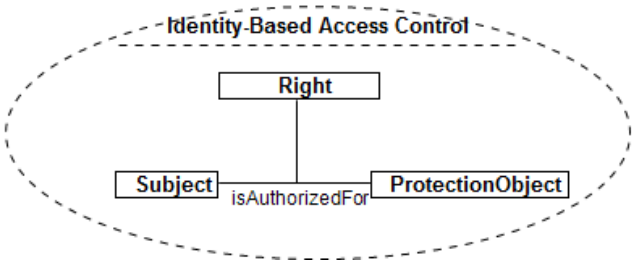

Abbildung 23: Struktur der Identitätsbasierten Zugriffskontrolle-Musters

Überprüfung

In [SF+06:246] wird auf eine mögliche Implementierung des Musters durch Zugriffskontrolllisten (engl. Access Control Lists, ACL) hingewiesen. Spring unterstützt den Einsatz von ACLs, jedoch wird standardmäßig nur eine Anbindung an eine Datenbank zum Laden der Berechtigungen angeboten. In [Mul10:226] wird das Vorgehen zur Erstellung der Tabellen und zur Konfiguration der Datenbank mit Spring an Hand eines Beispiels eingeführt. Weitere Speicherorte neben Datenbanken sind möglich, der Zugriff muss aber selbst implementiert werden.

Für die verschiedenen Aktionen auf den geschützten Objekten, wird ein Bitmuster verwendet, wie es in Unix-Dateisystemen ebenfalls zu finden ist. Die in der Datenbank abgelegte Zahl für die Berechtigung eines Benutzers berechnet sich aus Der Summe der in Tabelle 1 gezeigten Zugriffstypen und deren Bit.

Bit	16	8	4	2	1
Access type	Administer	Delete	Create	Write	Read

Tabelle 1: Tabelle zur Berechnung der Berechtigungen in Zugriffskontrolllisten

Neben der Bereitstellung der ACL-Einträge müssen die geschützten Ressourcen markiert werden. Die Ressourcen, die geschützt werden können, sind wie zuvor Methoden und URLs, die beispielsweise zum Zugriff auf Geschäftsobjekte genutzt werden. Auch bei ACLs können Annotationen für Methoden bzw. eine XML-Konfiguration für URLs genutzt werden. Der Ausdruck aus der Spring-EL ist bereits bekannt, *hasPermission*. Diesem wird wie zuvor das geschützte Objekt oder dessen Klasse und der Zugriffstyp als Zeichenkette, beispielsweise *Read* oder *Write*, übergeben. Aus der Zeichenkette wird das jeweilige Bitmuster errechnet und mit dem hinterlegten Wert in der Datenbank verglichen.

Die Zugriffskontrollliste wird von der *AclService*-Klasse bereitgestellt und je nach Implementierung beispielsweise aus der Datenbank geladen. Die Liste selbst ist als *Acl*-Klasse verfügbar

und bietet eine Liste von *AccessControlEntry*-Objekten an. Ein *AccessControlEntry* entspricht einem Eintrag in der Datenbank, der die Zugriffsrechte für einen Benutzer auf eine Ressource beschreibt. Die Klasse *ConfigAttribute* enthält den zu prüfenden Zugriffstyp, der bei *hasPermission* übergeben wird (*Read, Write, ...*). Die Entscheidung, ob ein Benutzer auf Grund der ACL-Konfiguration Zugriff auf eine Methode oder URL hat, wird von der Klasse *AclEntryVoter* getroffen. Diese fragt für den aktuellen Benutzer in der *Acl*-Klasse ab, ob dieser die nötigen Berechtigungen für die aktuelle Aktion besitzt, welche an Hand des *AccessControlEntry* und des *ConfigAttribute* entscheidet.

Überprüfungsergebnis

Die Instanziierung der identitätsbasierten Zugriffskontrolle durch die Spring Security Umsetzung von Zugriffskontrolllisten sieht folglich wie in Abbildung 24 aus. Das Subjekt ist wie bei RBAC und ABAC durch die Spring Klasse *User* repräsentiert. Das geschützte Objekt ist entweder eine URL oder eine Methode, hinter der sich potentiell ein Geschäftsobjekt verbirgt. Das Recht für den Zugriff ergibt sich aus der Entscheidung der *AclEntryVoter*-Klasse, die auf die Zugriffskontrollliste und die Parameter der Annotationen oder Konfiguration zugreift.

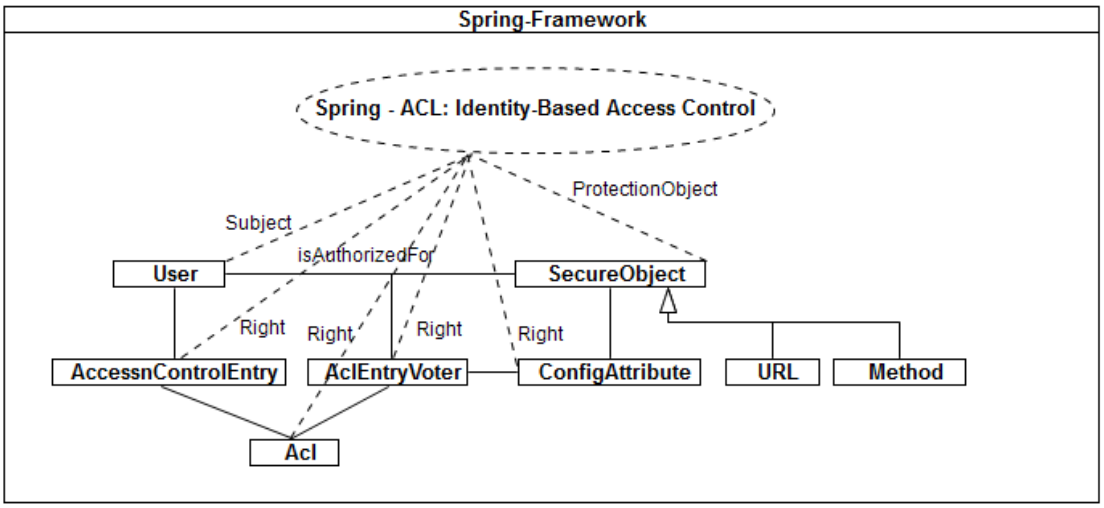

Abbildung 24: Instanziierung des IBAC-Entwurfsmusters durch Spring Security

Folglich wird das IBAC-Muster von Spring durch Zugriffskontrolllisten umgesetzt. Für jeden Benutzer kann der Zugriff auf ein Objekt für eine bestimmte Aktion gewährt oder verweigert werden. Die Verwaltung der Berechtigungen wird nicht von dem Spring Security Framework angeboten. Durch Spring Security wird die Anbindung an eine Datenbank als Quelle der Berechtigungen angeboten. Die Konfiguration hierzu wird in [Mul10:218ff] beschrieben. Dies kann jedoch erweitert werden, indem ein eigener *AclService* implementiert wird, der die Zugriffskontrollliste aus einer belieben Quelle lädt. Die Konfiguration des der Klasse wird in [Mul10:220f] beschrieben.

4.2.5 Autorisierung

Das Autorisierungsmuster abstrahiert von allen bisher gezeigten Zugriffskontrollmustern und ist in [SF+06:246, FH06:32, FP01:4] genauer beschrieben. Es kommt überall dort zum Einsatz, wo der Zugriff auf Ressourcen kontrolliert werden soll. Die Strukturen des Autorisierungs- und des IBAC-Musters aus Kapitel 4.2.4 sind gleich. Der grundlegende Unterschied zwischen den Mustern liegt in der Interpretation der Elemente.

Überprüfung und Ergebnis

Das Autorisierungsmuster bietet auf Grund des hohen Abstraktionsniveaus eine Reihe von Verfeinerungsmöglichkeiten. Daraus ergibt sich, dass sich weniger die Frage stellt, ob Spring das abstrakte Autorisierungsmuster umsetzt, sondern viel mehr, ob die konkrete Ausprägung mit Spring umgesetzt werden kann. Folgend soll gezeigt werden, dass die zuvor gezeigten Zugriffskontrollmuster Konkretisierungen des Autorisierungsmusters sind. Es soll dabei darauf hingewiesen sein, dass die folgenden Annahmen und Aussagen anders gesehen werden können. Muster und deren Rollen und Abläufe bieten einen gewissen Interpretationsspielraum, da ihre Beschreibung auch in natürlicher Sprache vorgenommen wird und es in der Regel keine exakte Definition der Elemente gibt.

Eine mögliche Verfeinerung der Autorisierung bietet die Subjektrolle. Die Kombination aus Rolle und Benutzer der rollenbasierten Zugriffskontrolle (Kapitel 4.2.1) kann als Verfeinerung der Subjektrolle angesehen werden, womit die RBAC eine Verfeinerung der Autorisierung ist [FP01:4]. Auch der Begriff Recht oder Berechtigung des Autorisierungsmusters ist abstrakt auszulegen; es ist nicht weiter definiert, wie es zu den Rechten kommt. Das Auswerten von Aussagen über Attribute, wie es in der attributbasierten Zugriffskontrolle (Kapitel 4.2.3) genutzt wird, um das Recht des Benutzers für eine Ressource zu berechnen, kann als eine Verfeinerung des Rechts bei der Autorisierung verstanden werden. Die ABAC ist dementsprechend auch eine Form der Autorisierung. Die IBAC (Kapitel 4.2.4) ist, wie bereits beschrieben, eine Verfeinerung des Autorisierung-Musters [SF+06:245].

In den vorigen Kapiteln wurde gezeigt, dass eine Umsetzung von drei Verfeinerungen der Autorisierung mit dem Spring Security Framework möglich ist. Da Spring Security Einstiegsmöglichkeiten zur Erweiterung des Frameworks anbietet, beispielsweise das Implementieren eines eigenen *PermissionEvaluators* zur individuellen Auswertung von Rechten oder das Laden eigener Benutzerobjekte mit beliebigen Attributen, sollte eine Vielzahl von Formen der Autorisierung mit dem Spring Security Framework umgesetzt werden können.

4.3 Überprüfung auf Architekturmuster

4.3.1 Intercepting Web Agent

Das Intercepting Web Agent-Muster kann eingesetzt werden, um eine Anwendung nachträglich um Authentifizierungs- und Autorisierungsfunktionen zu erweitern oder diese Aspekte von der Anwendungslogik zu trennen. Wie der Name bereits suggeriert, ist das Muster spezifisch für Web-Anwendungen gedacht, da ein Web Server Rollen im Muster übernimmt und zudem Cookies eingesetzt werden sollen. Vorgestellt wird das Muster in [SPR04:606].

Die Struktur des Musters ist auf Abbildung 25 zu sehen. Diese sieht einen *Client* vor, der eine Anfrage an einen Web Server stellen kann. Der *InterceptingWebAgent* fängt Anfragen an die Anwendung ab und überprüft, ob der *Client* authentifiziert und autorisiert für die Anfrage ist. Der *PolicyServer* verwaltet die Richtlinien für die Autorisierung und kann in den *InterceptingWebAgent* integriert oder als externe Komponente verfügbar sein.

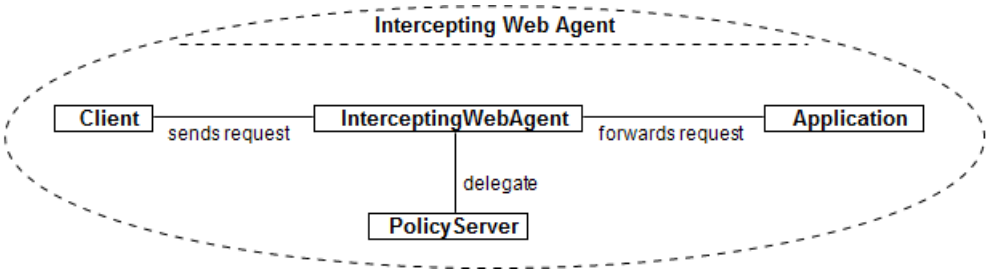

Abbildung 25: Struktur des Intercepting Web Agent-Musters

Abbildung 26 zeigt den vorgesehenen Ablauf. Jede Anfrage an die Anwendung wird von dem *InterceptingWebAgent* abgefangen. Dieser bietet dem *Client* eine Anmeldemöglichkeit an und veranlasst den *Client*, ein Cookie mit verschlüsselten Sitzungsinformationen zu speichern. Bei einer Anfrage verifiziert der *InterceptingWebAgent* das Cookie des *Clients* und überprüft an Hand der Richtlinien des *PolicyServers*, ob der Anwender für die Anfrage autorisiert ist. Falls der Anwender die nötige Berechtigung hat, wird die Anfrage an die Anwendung weitergeleitet.

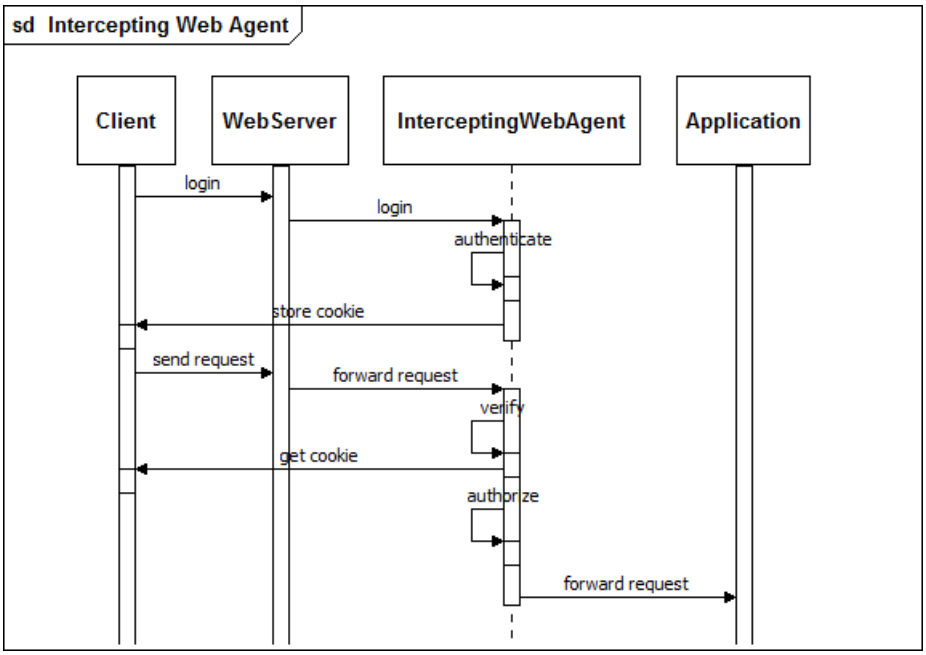

Abbildung 26: Ablauf des Intercepting Web Agent-Musters

Überprüfung

Bei der Betrachtung dieses Musters ist vor allem die Nutzung von Filtern durch Spring nach der Java Servlet Spezifikation [JCP02:43] von Interesse. Als angedachte Verwendung der Filter wird unter anderem die Authentifizierung genannt, [JCP02:44].

Spring Security setzt auf dieser Spezifikation auf und nutzt diese für verschiedene Aufgaben. Um das Spring Security Framework in einer Web-Anwendung verwenden zu können, muss die Spring Klasse *DelegatingFilterProxy* als Servlet-Filter eingestellt werden. Dieser leitet bei einer Anfrage an Spring-spezifische Filter weiter. In [Mul10:34] werden die verschiedenen Spring Filter-Klassen vorgestellt, die bei der Standardkonfiguration zur Filterkette hinzugefügt werden. Für das Intercepting Web Agent-Muster interessant, sind eine Reihe von Authentifizierungsfiltern, der *SecurityContextPersistenceFilter* und der *FilterSecurityInterceptor*.

Die Authentifizierungsfilter dienen der Authentifizierung des Benutzers durch verschiedene Methoden, beispielsweise per Formular. Der *SecurityContextPersistenceFilter* speichert die

Anmeldeinformationen des Benutzers in einer Sitzung und weist diese dem Benutzer per Cookie zu. Falls Cookies clientseitig deaktiviert sind geschieht dies per Anfrageparameter. Abschließend dient der *FilterSecurityInterceptor* der Durchsetzung der Zugriffskontrolle.

Gemäß dem vorgesehenen Ablauf im Muster wird zunächst die Durchsetzung der Authentifizierung und Erstellung einer Sitzung für den Benutzer und anschließend der Zugriffskontrolle betrachtet.

Überprüfung der Authentifizierungsdurchführung

In der Standardkonfiguration sind laut [Mul10:35] der *DefaultLoginPageGeneratingFilter*, der *UsernamePasswordAuthenticationFilter* und der *BasicAuthenticationFilter* voreingestellt.

Der *DefaultLoginPageGeneratingFilter* überprüft bei einer Anfrage, ob eine Authentifizierung notwendig ist. Im positiven Fall erzeugt der Filter eine HTML-Anmeldeseite und setzt diese zur Darstellung als Antwort der Anfrage. Der Benutzer kann seine Anmeldedaten auf der HTML-Seite eingeben und das Formular abschicken.

Der *UsernamePasswordAuthenticationFilter* ist für die Authentifizierung mit Benutzername und Passwort als Anfrageparametern, beispielsweise abgeschickt per zuvor generierter HTML-Anmeldeseite, zuständig. Die Daten leitet er an den Spring-*AuthenticationManager* weiter, der die Daten überprüft und konkret die Authentifizierung durchführt. Der *AuthenticationManager* kann frei konfiguriert werden und lädt die Benutzerdaten, die zur Anmeldung notwendig sind, aus dem konfigurierten Benutzerverzeichnis, beispielsweise ein LDAP-Server oder eine XML-Konfiguration, wie bereits in Kapitel 4.2.1 beschrieben.

Falls Anmeldeinformationen gemäß dem Basic Authentication Scheme des RFC 1945 [NWG96:47] bei der HTTP-Anfrage gesendet werden, so entnimmt der *BasicAuthenticationFilter* diese Daten aus der Anfrage und übergibt sie ebenfalls dem Spring-*AuthenticationManager*, der wiederum die Validierung und Authentifizierung des Benutzers übernimmt.

Nachdem ein Filter den Benutzer authentifiziert hat, legt der *SecurityContextPersistenceFilter* mittels des *HttpSessionSecurityContextRepository* einen Sicherheitskontext für den Benutzer in der HTTP-Sitzung ab. Bei jeder weiteren Anfrage des Benutzers lädt der Filter den abgelegten Sicherheitskontext. Wie die HTTP-Sitzung dabei dem Benutzer zugewiesen wird, verwaltet der sogenannte *Servlet Container*, auf dem die Anwendung ausgeführt wird. Dies kann beispielsweise über die Speicherung von Cookies gemäß dem RFC 6265 [IETF11] auf Seite des Benutzers oder durch ein Ergänzen der URL um einen Parameter zur eindeutigen Kennung geschehen.

Neben den hier dargestellten Authentifizierungsmechanismen durch die Spring-Standardfilter, sind weitere Mechanismen, wie die Authentifizierung mittels eines Zertifikats bereits von Spring implementiert. Es ist ebenfalls möglich eigene Authentifizierungsfilter zu schreiben. Auf dieses Thema wird in Kapitel 4.3.3 weiter eingegangen.

Überprüfung der Autorisierungsdurchführung

Durch die bisherigen Filter wurde die Authentifizierung des Benutzers übernommen und ein Sicherheitskontext aufgebaut, der bei jeder Anfrage des Benutzers wieder hergestellt wird. Der nächste Schritt im Ablauf des Intercepting Web Agent-Musters ist die Durchführung der Zugriffskontrolle.

Anfangs wurde bereits auf den Filter *FilterSecurityInterceptor* und dessen Funktion zur Zugriffskontrolle hingewiesen. Der Filter entnimmt aus dem Sicherheitskontext das Objekt des authentifizierten Benutzers. Weiter übergibt der Filter die für die Zugriffskontrolle relevanten Daten, wie beispielsweise das Benutzerobjekt samt Rollen und Attributen des geschützten Objekts, an den *AccessDecisionManager*. Der *AccessDecisionManager* entscheidet konkret, ob die Anfrage autorisiert ist. Bei der Entscheidung werden die in den Kapiteln über die Muster zur Zugriffskontrolle bereits eingeführten *Voter* genutzt.

Allerdings führt der *FilterSecurityInterceptor* nur eine Zugriffskontrolle für URLs durch, die zur Absicherung vorkonfiguriert wurden, wie in Quellcodebeispiel 9 gezeigt. Daneben wurde bereits die Absicherung von Methoden erwähnt. Der Ablauf sieht bei der Durchsetzung der Zugriffskontrolle anders aus. Das Spring Security Framework setzt hierbei auf das Programmierparadigma der aspektorientierten Programmierung (AOP, siehe Kapitel 2.4).

Die Annotationen der Methoden werden durch AOP so erweitert, dass eine Zugriffskontrolle vor bzw. nach der Ausführung der Methode, je nach Annotation, durchgeführt wird. Die Spring-Aspekt-Klasse ist *AnnotationSecurityAspect*. Bei einem Aufruf einer gesicherten Methode oder Klasse leitet der Aspekt an den *AspectJMethodSecurityInterceptor* weiter. Diese erweitert die abstrakte Klasse *AbstractSecurityInterceptor*.

Vor Methodenausführung wird im *AbstractSecurityInterceptor* überprüft, ob der Benutzer authentifiziert ist, wobei die Daten aus dem Sicherheitskontext geladen werden. Der Benutzer muss sich somit schon an dem System authentifiziert haben, beispielsweise durch ein der zuvor genannten Filter. Daraufhin wird vor und nach dem Methodenaufruf die Autorisierung des Benutzers geprüft und gegebenenfalls den Zugriff verweigert. Wie auch zuvor wird bei der Methodensicherheit ebenfalls der *AccessDecisionManager* für die Zugriffsentscheidung genutzt.

Bei der Zugriffskontrolle auf Methodenebene stellt sich die Frage, ob der Ablauf des Musters eingehalten wurde. Der Ablauf beschreibt den Eingriff des *Intercepting Web Agents* vor der Ausführung der Anwendung. Eine geschützte Methode gehört allerdings zur Anwendung und in [SPR04:614] wird darauf hingewiesen, dass das Muster oft zum Schutz auf URL-Ebene genutzt wird und es nicht für einen feingranulareren Schutz gedacht ist. Aus diesem Grund passt das Muster nicht zur Methodenzugriffskontrolle.

Überprüfungsergebnis

Das Intercepting Web Agent-Muster wird im Fall der Zugriffskontrolle von URLs von dem Framework umgesetzt. Die Instanziierung des Musters durch das Spring Security Framework sieht wie in Abbildung 27 aus. Der Client wird durch einen Web-Client repräsentiert. Die Spring Klassen *AuthenticationManager* und *AuthorizationManager* können als *Policy Server* angesehen werden. Allerdings liegen die Klassen innerhalb des Frameworks und eine Zuweisung zur Rolle *Intercepting Web Agent* wäre ebenfalls möglich, wenn bei dem Muster der *Policy Server* als eigenständige Anwendung außerhalb der gesicherten Anwendung verstanden wird. Die verschiedenen vorgestellten Filter und die Filterkette von Spring nehmen die Rolle des *Intercepting Web Agent* ein. Die durch das Framework geschützte Anwendung nimmt die gleichnamige *Application*-Rolle ein.

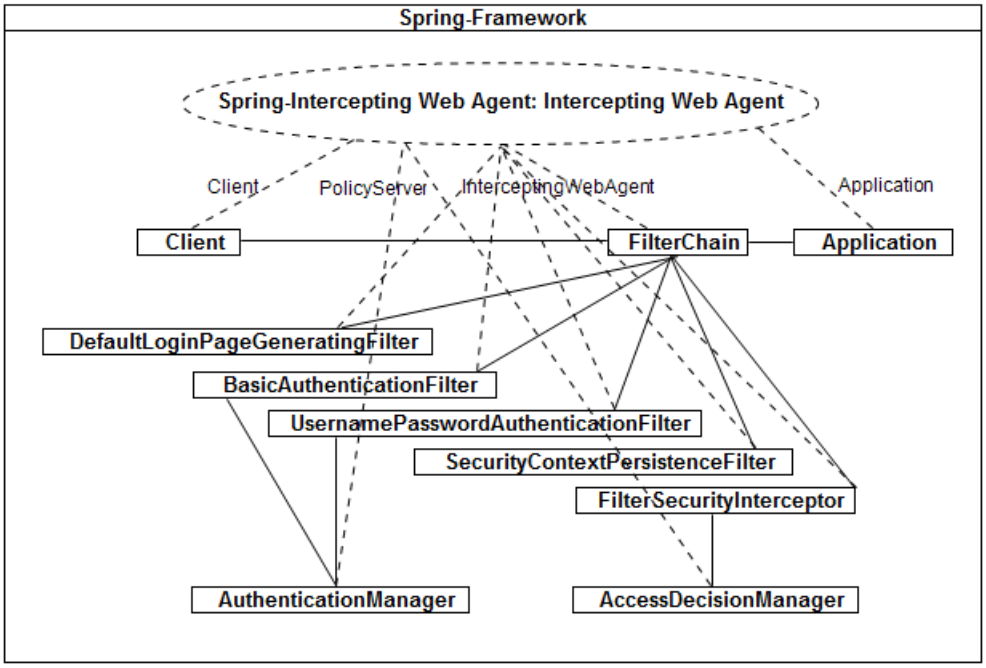

Abbildung 27: Instanziierung des Intercepting Web Agent-Musters durch Spring Security

4.3.2 Authorization Enforcer

Das Authorization Enforcer-Muster soll die Zugriffskontrolle in einer Web-Anwendung mit mehreren Zugriffspunkten an einer zentralen Stelle kapseln. Zudem kann das Muster genutzt werden, um die Authentifizierungs- von der Autorisierungslogik zu trennen. Eine Beschreibung des Musters, sowie die Struktur und Abläufe sind in [SPR04:549] zu finden. Dort werden ebenfalls mehrere Implementierungsstrategien aufgezeigt. Eine Strategie ist die Nutzung des Java Authentication and Authorization Service (JAAS)-Frameworks. Eine weitere Strategie ist die Nutzung der Java Security API-Klassen und die letzte Strategie sieht die Einbindung von Fremdsoftware vor.

In [SPR04:549] wird strukturell nur die Strategie, die den Einsatz der Java Security API vorsieht, dargestellt, weshalb auf diese im Folgenden eingegangen wird. Die Struktur dieser Strategie sieht den Einsatz einer *SecureBaseAction* vor, auf die der Benutzer bei einer Anfrage zugreift. Von dort wird die Anfrage an den *AuthorizationEnforcer* weitergeleitet. Die Berechtigungen des Subjekts werden von dem *AuthenticationProvider* geladen und für den *AuthorizationEnforcer* und die *SecureBaseAction* zugänglich gemacht. Diese Struktur zeigt Abbildung 28.

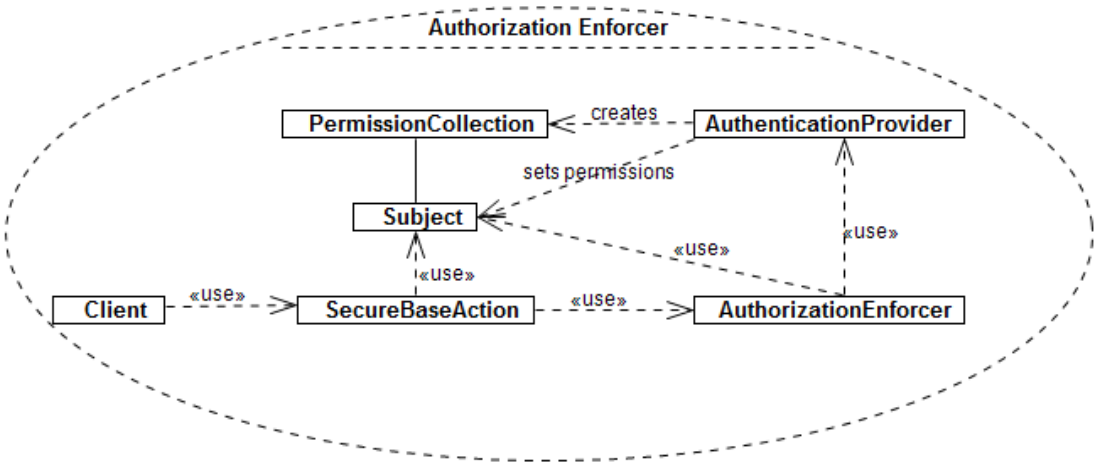

Abbildung 28: Struktur des Authorization Enforcer-Musters

Der konkrete Ablauf ist auf Abbildung Abbildung 29 dargestellt und sieht wie folgt aus:

1) Die *SecureBaseAction* lädt das Benutzerobjekt aus dem Java-*RequestContext*

2) Die *SecureBaseAction* fragt beim *AuthorizationEnforcer* nach der Autorisierung des Benutzers an

   2.1) Der *AuthorizationEnforcer* leitet die Anfrage an den *AuthenticationProvider* weiter

      2.1.1) Die Berechtigungsliste des Benutzers

      2.1.2) und seine Anmeldedaten werden bereitgestellt

3) Die *SecureBaseAction* fragt bei dem *AuthorizationEnforcer* ab, ob das aktuelle Subjekt für die Anfrage autorisiert ist und leitet gegebenenfalls an die Anwendung weiter.

   3.1) Der *AuthorizationEnforcer* benutzt die Benutzerdaten und die Berechtigungsliste, um eine Zugriffsentscheidung zu treffen

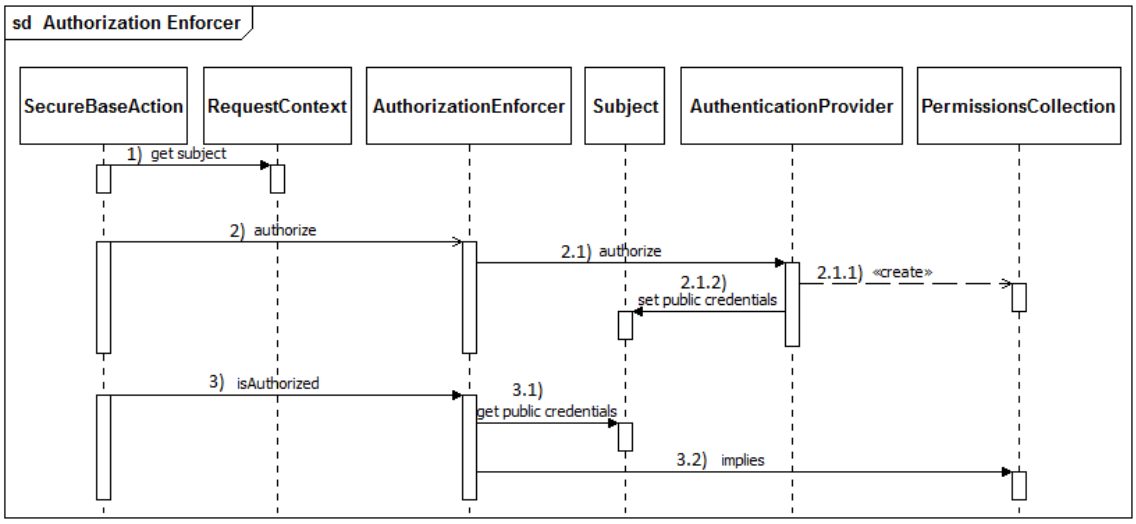

Abbildung 29: Ablauf des Authorization Enforcer-Musters

Das Muster baut auf dem Secure Base Action-Muster auf, das ebenfalls in [SPR04:569] beschrieben wird. Dieses sieht eine Klasse vor, die alle Sicherheitsfunktionen in sich kapselt, wie beispielsweise Autorisierung, Authentifizierung, sicheres Protokollieren oder das Aufbauen einer sicheren Verbindung. Anfragen an die Web-Anwendung werden zuerst von der *SecureBaseAction* verarbeitet und daraufhin an die Anwendung weitergeleitet. So kann verhindert

werden, dass Änderungen an der Sicherheitsfunktionalität Änderungen in jedem Anwendungsteil zur Folge haben.

Abstrahiert auf den Kern des Musters, wäre das Secure Base Action-Muster nicht notwendig zur Umsetzung des Authorization Enforcer-Musters. Es ist nur wichtig, dass es eine Komponente gibt, die zwischen der Anwendungslogik und dem *AuthorizationEnforcer* vermittelt und dafür sorgt, dass der *AuthorizationEnforcer* Zugriff auf ein gültiges Benutzerobjekt hat. Dabei kann beispielsweise auf das Authentication Enforcer-Muster aus Kapitel 4.3.4 zurückgegriffen werden.

Überprüfung

Im Kapitel über das Intercepting Web Agent-Muster wurden bereits die Mechanismen, die das Spring Security Framework zur Durchsetzung der Autorisierung durchführt, angesprochen. Diese sind zum einen Filter nach der Java Servlet Spezifikation [JCP02:43] und zum anderen Aspekte gemäß der aspektorientierten Programmierung (AOP, siehe Kapitel 2.4). Für die Überprüfung auf das Authorization Enforcer-Muster sind diese Mechanismen der Einstiegspunkt.

Für URL-Zugriff und Methodenzugriff muss geprüft werden, ob das Secure Base Action-Muster umgesetzt wird. Zuerst wird die URL-Ebene betrachtet. Das Muster sieht vor, dass eine einzelne Java-Klasse die Koordination aller sicherheitsrelevanten Aufgaben vereint. Die Sicherheitsfilterkette [SprS11e] des Spring Security Frameworks besteht aus einer Reihe von konfigurierbaren Filtern und keiner koordinierenden Einheit die kontrolliert, dass alle Sicherheitsfilter aufgerufen werden. Es gibt keine einzelne Klasse, die jeden Aufruf verarbeitet. Somit stellt die Filterkette keine Umsetzung des Secure Base Action-Musters dar.

Nun zur Betrachtung des Methodenzugriffs. Die Nutzung von Aspekten bietet keine Möglichkeit zur Authentifizierung an, da diese voraussetzt, dass der Benutzer sich bereits authentifiziert hat. Dies wurde in der Überprüfung des Intercepting Web Agent-Muster in Kapitel 4.3.1 bereits gezeigt. Da Authentifizierung und Autorisierung somit nicht in einer Komponente gekapselt sind, setzt auch dieser Mechanismus das Secure Base Action-Muster nicht um.

Zur Umsetzung der abstrakten Form des Musters ist das Secure Base Action-Muster auch nicht notwendig, wie bereits beschrieben. Allerdings zeigt dies, dass die Strategie basierend auf der Java Security API mit *SecureBaseAction* nicht umgesetzt wird. Weiter wird überprüft, ob das abstrakte Muster durch die Filterkette und deren Kombination mit Aspekten von dem Spring Security Framework umgesetzt wird.

Durch die verschiedenen Authentifizierungsfilter des Spring Security Frameworks, vorgestellt in Kapitel 4.3.1, wird sichergestellt, dass der Benutzer sich auf verschiedene Wege authentifizieren kann. Die Daten des Benutzers aus dem Benutzerspeicher, darunter notwendige Informationen für die Zugriffskontrolle, werden nach der Authentifizierung mit dem *AuthenticationManager* durch den Sicherheitskontext des Spring Security Frameworks bereitgestellt und, wie in dem Muster modelliert, im *RequestContext* der Java Web-Anwendung von dem *SecurityContextPersistenceFilter* gespeichert.

Zur Überprüfung der Autorisierung mit dem Spring Security Framework werden die bereits benannten Mechanismen für den Schutz von URLs und Methoden genutzt, deren Umsetzung in Kapitel 4.3.1 beschrieben ist.

Die Kombination der Filter und der Filterkette und zusätzlich der Aspekte kann als abstrakte *SecureBaseAction* angesehen werden, da diese die geforderte Verteilung der Sicherheitsaufgaben basierend auf dem Prinzip der losen Kopplung umsetzt.

Das Muster soll von der konkreten Zugriffskontrolle abstrahieren und die jeweilig notwendigen Informationen soll der *AuthorizationEnforcer* selbst laden. Dies geschieht hinsichtlich der Benutzerdaten durch den *AuthenticationManager* und bei Informationen die das geschützte Objekt respektive die Umgebung betreffen durch den *PermissionEvaluator*.

Überprüfungsergebnis

Das Ergebnis der Überprüfung ist, dass das Spring Security Framework das Authorization Enforcer-Muster umsetzt, allerdings nicht nach der in [SPR04] vorgestellten Strategie mit der Java Security-API, sondern in seiner abstrakten Form. Die Instanziierung des Musters zeigt Abbildung 30.

Der Client nimmt die gleichnamige Rolle ein und seine Anfrage wird von der *FilterChain* abgefangen und weiterverarbeitet, diese nimmt die Rolle einer abstrakten *SecureBaseAction* ein. Das Laden des Benutzers aus dem *RequestContext* geschieht durch den *SecurityContextPersistenceFilter*, weshalb dieser ebenfalls die Rolle der *SecureBaseAction* einnimmt. Durch die Filterkette wird an Authentifizierungsfilter weitergeleitet, die für die Autorisierung notwendige Daten des Benutzers über den *AuthenticationManager* laden und bereitstellen. Die Authentifizierungsfilter nehmen somit die Rolle des *AuthorizationEnforcers* ein und der *AuthenticationManager* die Rolle des *AuthenticationProviders*. Der *FilterSecurityInterceptor* und *AspectJMethodSecurityInterceptor* leiten an den *AuthorizationManager* weiter, der die Autorisierung des Benutzers an Hand der Zugriffskontrollemechanismen prüft. Die Interceptor nehmen somit die Rolle der *SecureBaseAction* ein, der *AuthorizationManager* die Rolle des *AuthorizationEnforcers* und die Zugriffskontrollmechanismen die Rolle der *PermissionCollection*. Das Subjekt ist im Spring Security Framework durch die *User*-Klasse umgesetzt.

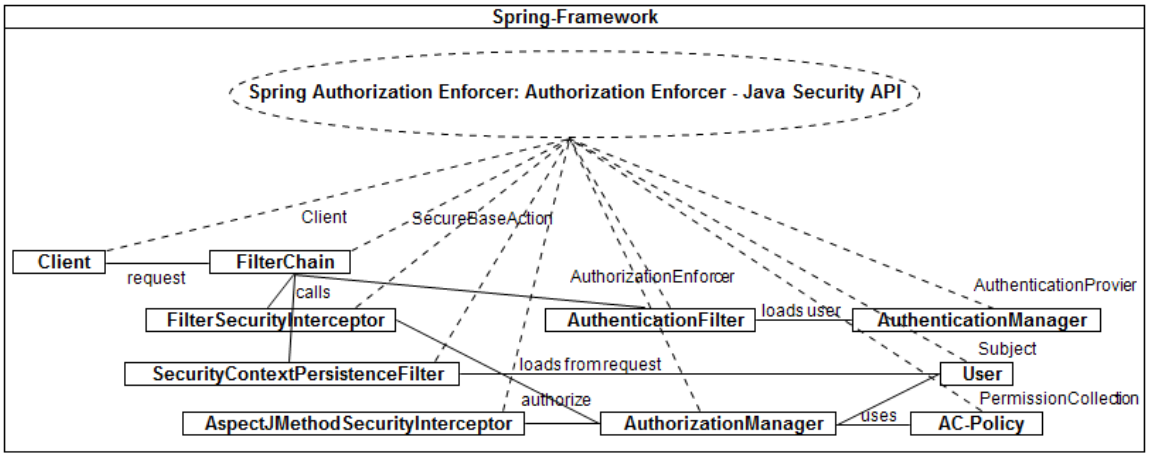

Abbildung 30: Instanziierung des Authorization Enforcer-Musters durch das Spring Framework

4.3.3 Reference Monitor oder Policy Enforcement Point

Die Definition des Reference Monitor-Musters wird in [SF+06:256, FH06:40] vorgestellt mit dem Verweis auf das Synonym Policy Enforcement Point (PEP). Das Reference Monitor-Muster ist recht abstrakt und beschreibt die Struktur und Abläufe zur Durchsetzung der Autorisierungsregeln von Ressourcen in Software-Systemen. Was genau eine konkrete Ressource ist,

wie eine Anfrage an eine Ressource abgefangen wird oder wie die Autorisierungsregeln aussehen, ist in dem Muster nicht weiter definiert.

Die Rollen des Musters zeigt Abbildung 31. Es gibt ein Subjekt, das auf geschützte Objekte zugreifen will, und einen Reference Monitor, der die Anfragen abfängt. In der Struktur wird das geschützte Objekt nicht erwähnt, jedoch im Sequenzdiagramm. Der Verweis auf konkrete Reference Monitors soll darauf hinweisen, dass das abstrakte Prinzip für eine Anwendung konkretisiert werden muss. Der abstrakte Reference Monitor ist mit verschiedenen Autorisierungsregelmengen verbunden, mit denen er überprüfen kann, ob das Subjekt für den Zugriff autorisiert ist. Über die Autorisierungsregeln wird keine konkrete Aussage getroffen, diese können durch Zugriffskontrollmechanismen aufgestellt werden, beispielweise die zuvor vorgestellten Muster (rollenbasierte-, attributbasierte- und identitätsbasierte Zugriffskontrolle).

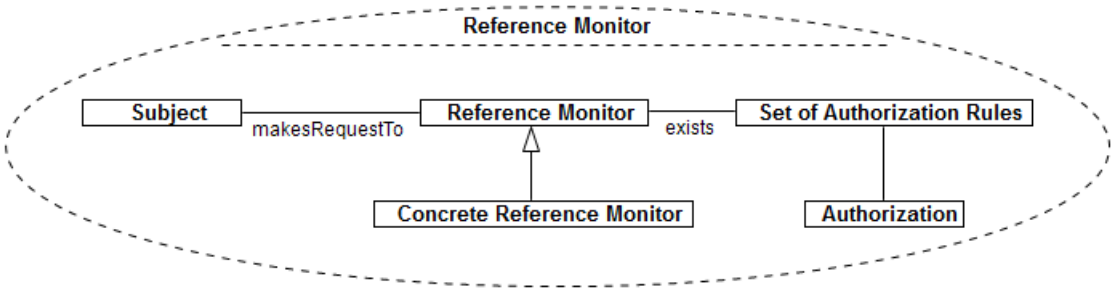

Abbildung 31: Struktur des Reference Monitor-Musters

Die Interaktion zwischen den Rollen stellt Abbildung 32 dar. Jede Anfrage auf geschützte Objekte wird vom Reference Monitor abgefangen. Dieser überprüft mittels der Menge von Autorisierungsregeln, ob eine Autorisierung für das Subjekt auf das geschützte Objekt besteht und leitet gegebenenfalls die Anfrage an das geschützte Objekt weiter.

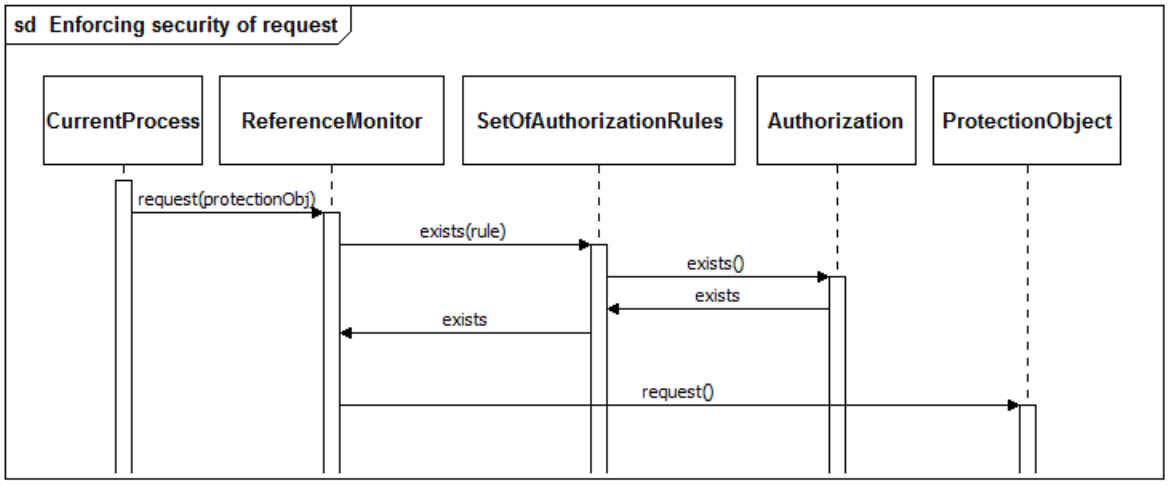

Abbildung 32: Darstellung der Anfrageverarbeitung beim Reference Monitor-Muster

Überprüfung und Ergebnis

Da das Muster ist recht abstrakt und es ist nicht genauer spezifiziert, wann und in welcher Form der *ReferenceMonitor* die Anfrage an das geschützte Objekt abfangen soll. Beispielsweise könnten ein AOP-Aspekt, durch den der Zugriff auf eine Methode geschützt wird, oder ein Webservice, der als Einstiegspunkt für eine Anwendung genutzt wird und die Zugriffskontrolle vor der Weiterleitung an die eigentliche Anwendungslogik durchführt, die Rolle des konkreten *ReferenceMonitor* auf unterschiedlichen Ebenen übernehmen. Im Folgenden soll geprüft wer-

den, ob das Intercepting Web Agent-Muster oder das Authorization Enforcer-Muster Verfeinerungen des Reference Monitor-Musters sind. Auch in diesem Fall gilt, wie bereits bei der Überprüfung des Autorisierungs-Musters (Kapitel 4.2.5) hingewiesen wurde, dass ein gewisser Interpretationsspielraum besteht.

Das bereits vorgestellte Intercepting Web Agent Muster aus Kapitel 4.3.1 führt ebenfalls eine Zugriffskontrolle durch und könnte das Reference Monitor-Muster umsetzen. Das Muster sieht vor, dass Anfragen an eine web-basierte Anwendung abgefangen werden und deren Autorisierung geprüft wird. Bei dem Muster wird nicht konkret auf Autorisierungsregeln eingegangen. Allerdings müssen solche Regeln vorhanden sein, um die Autorisierung beim Intercepting Web Agent-Muster prüfen zu können. Das Muster legt den Fokus auf den Ablauf bei der Anfragever-arbeitung, weswegen die Autorisierungsregeln vermutlich nicht explizit modelliert wurden. Das Intercepting Web Agent-Muster setzt somit das Reference Monitor-Muster spezialisiert für Web-Anwendungen um.

Das *AuthorizationEnforcer*-Muster ist ein weiterer Kandidat als Verfeinerung des Reference Monitor-Musters. Das Muster setzt auf Methoden- und URL-Ebene die Zugriffskontrolle um und benutzt hierfür eine *PermissionCollection*, die synonym zur Menge von Autorisierungsregeln interpretiert werden kann. Der konkrete Reference Monitor wird bei dem Muster durch die abstrakte *SecureBaseAction* und den *AuthenticationEnforcer* repräsentiert. Alle Anfragen an die Anwendung werden durch diese Mechanismen abgefangen und auf ihre Autorisierung geprüft. Dieses Muster kann somit wegen der Zugriffskontrolle für Methoden als Verfeinerung des Reference Monitor-Musters für Java-Anwendungen angesehen werden.

Spring Security setzt den Reference Monitor für Java-Anwendungen um und es werden zwei Verfeinerungen für diese Technologie angeboten. Für den Schutz der laufenden Java Anwendung auf dem Betriebssystem oder Server müssten weitere Sicherheitsmaßnahmen getroffen werden, weswegen ein allgemeines Idiom zur Umsetzung des Musters mit dem Spring Security Framework nicht gegeben werden kann.

4.3.4 Authentication Enforcer

Durch dieses Muster soll die Web-Anwendungs- von der Authentifizierungslogik entkoppelt werden. Der Code für die Authentifizierung des Benutzers wird nicht in jedem Teil der Anwendung implementiert, sondern an einer zentralen Stelle, um beispielsweise bei Änderungen der Authentifizierungslogik nicht jeden Anwendungsteil ändern zu müssen. Zudem soll von dem konkreten Authentifizierungsmechanismus abstrahiert werden, so dass ohne größeren Implementierungsaufwand zwischen beispielsweise der Anmeldung mit einem Formular und der Anmeldung mit Zertifikaten gewechselt werden kann. Das Authentication Enforcer-Muster wird in [SPR04:535] detaillierter beschrieben.

Das Muster sieht vor, dass Authentifizierungsanfragen von dem *AuthenticationEnforcer* verarbeitet werden. Dieser entnimmt je nach Authentifizierungsmechanismus die notwendigen Daten aus dem *RequestContext* und vergleicht diese mit den hinterlegten Daten des Benutzers. Sind die Daten des Benutzers korrekt, erzeugt der *AuthenticationEnforcer* ein Benutzerobjekt und speichert dieses für die Anfrage ab.

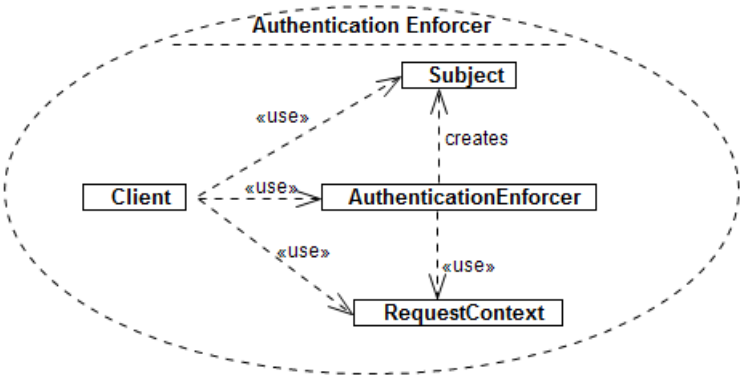

Abbildung 33: Struktur des Authentication Enforcer-Musters

Überprüfung

In den vorigen Kapiteln wurde bereits mehrfach auf die Authentifizierung des Benutzers und den Einsatz von Filtern zu diesem Zweck eingegangen. Beispielsweise sieht das Intercepting Web Agent-Muster die Authentifizierung des Benutzers vor und setzt im Falle des Spring Security Frameworks das Authentication Enforcer-Muster um. Dass dem so ist, soll im Folgenden gezeigt werden.

Durch die Umsetzung der Filterkette und der Java Servlet Spezifikation [JCP02] wird sichergestellt, dass jede Anfrage an die Web-Anwendung durch die Filter verarbeitet wird. Verschiedene Anmeldemechanismen bieten je einen eigenen Filter an, wie es in Kapitel 4.3.1 unter Überprüfung der Authentifizierungsdurchführung beschrieben ist. Dort wird auch der *DefaultLoginPageGeneratingFilter* eingeführt, der zur Darstellung eines HTML-Anmeldeformulars genutzt werden kann, falls der Benutzer noch nicht authentifiziert ist.

Falls ein gewünschtes Authentifizierungsverfahren von Spring derzeit noch nicht unterstützt wird, kann ein entsprechender Filter für das Verfahren implementiert werden. Hierfür kann eine Java-Klasse geschrieben werden, die die abstrakte Spring Klasse *AbstractAuthenticationProcessingFilter* erweitert.

Beim Erweitern der abstrakten Klasse muss die Methode *doFilter(request, response)* implementiert werden. In dieser kann auf das aktuelle Anfrageobjekt zugegriffen werden und beispielsweise Formulardaten ausgelesene werden. Gegebenenfalls kann auch das Antwortobjekt verändert werden und beispielsweise ein HTML-Formular zur Eingabe des Benutzernamens und -passworts dargestellt werden, so wie es bei dem *DefaultLoginPageGeneratingFilter* geschieht.

Die abstrakte *AbstractAuthenticationProcessingFilter*-Klasse bietet unter anderem bereits Methoden zur Behandlung von Ausnahmen an, beispielsweise wenn der Benutzer durch den Filter nicht authentifiziert werden konnte. Ebenfalls wird durch die abstrakte Klasse ein *AuthenticationManager* vorausgesetzt und ist als Objekt verfügbar, so dass dieser zur Authentifizierung gegenüber den Benutzerspeichern genutzt werden kann.

Die *AuthenticationManager*-Schnittstelle bietet die Möglichkeit ein Benutzerobjekt mit Kennung und anderen notwendigen Informationen für die Anmeldung, wie beispielsweise ein Passwort, gegenüber Benutzerspeichern zu authentifizieren. Dabei leitet die Standardimplementierung an verschiedene *AuthenticationProvider* weiter. Je nach Benutzerverzeichnis gibt es unterschiedliche Implementierungen dieser *Provider*, vergleichbar mit den unterschiedlichen *Votern* bei der Autorisierung je nach Zugriffskontrollmechanismus. Es gibt beispielsweise den *JaasAuthenticationProvider*, *LdapAuthenticationProvider* und *OpenIDAuthenticationProvider*,

die eine Authentifizierung gemäß dem JAAS-Standard [Ora11b], gegenüber einem LDAP-Server oder einem OpenID-Anbieter durchführen.

Falls ein Benutzerspeicher noch nicht vom Spring Security Framework unterstützt wird, kann ein *Provider* für den Benutzerspeicher ergänzt werden. Dabei muss die Schnittstelle *AuthenticationProvider* implementiert werden. Diese sieht zwei Methoden vor. Zum einen die Methode *supports(Object)*, mit der überprüft werden kann, ob der Provider die vom Benutzer gelieferten Daten zur Authentifizierung unterstützt oder nicht und zum anderen die Methode *authenticate(Object)*, die die Authentizität des Benutzers prüft und bei korrekten Anmeldeinformationen ein Benutzerobjekt vom Typ *Authentication* aus dem Spring Security Framework mit den Daten aus dem Benutzerspeicher zurückliefert.

Durch diese Erweiterungsmöglichkeiten ist eine Variabilität hinsichtlich der Benutzerspeicher sowie Authentifizierungsmechanismen möglich und die Integration externer Software ist möglich.

Das zeigt, dass alle Anfragen durch die Authentifizierungsfilter vor der Ausführung der Anwendung verarbeitet werden und das Anmelden des Benutzers durch die Daten im Anfragekontext ermöglicht wird. Die Authentifizierungsmethode sowie der Benutzerspeicher sind variabel und das Framework bietet Einstiegsmöglichkeiten, um andere Methoden oder Speichern zu integrieren.

Überprüfungsergebnis

Die Instanziierung des Authentication Enforcer-Musters durch das Spring Security Framework ist somit auf Grund der Authentifizierungsfilter und der Architektur möglich, wie in Abbildung 34 dargestellt.

Anfragen des Clients werden von den Filtern verarbeitet. Die Filter leiten entweder Maßnahmen ein, damit der Benutzer seine Anmeldeinformationen angeben kann oder die Anmeldeinformationen aus dem *RequestContext*, der die gleichnamige Rolle einnimmt, weiterverarbeitet werden. Die Weiterverarbeitung übernimmt der *AuthenticationManager*, der wiederum spezielle *AuthenticationProvider*, je nach eingesetztem Benutzerspeicher, aufruft, um eine Authentifizierungsentscheidung zu erhalten. Dieses Zusammenspiel repräsentiert die Rolle des *AuthenticationEnforcer*. Bei einer erfolgreichen Authentifizierung werden die Benutzerdaten in der Klasse *Authentication* gespeichert, welche die Rolle des Subjekts einnimmt.

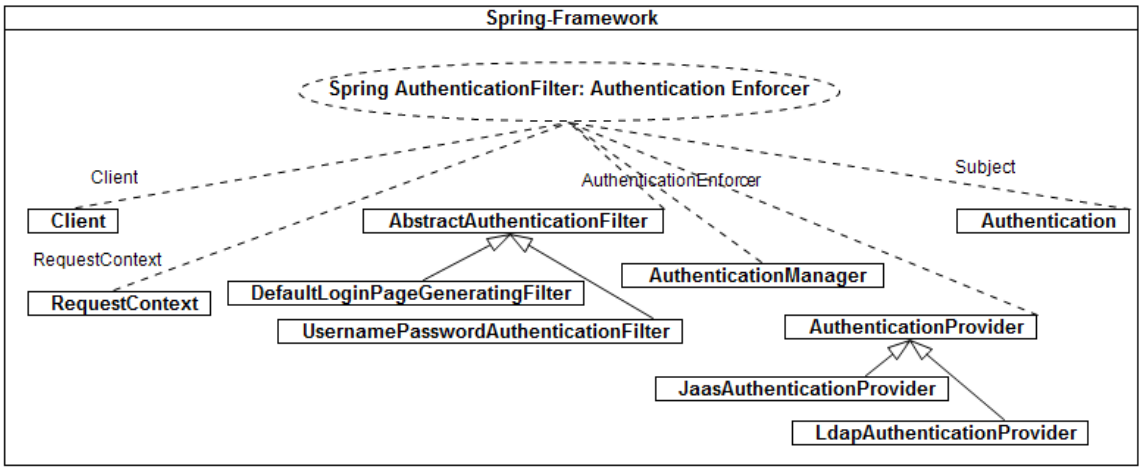

Abbildung 34: Instanziierung des Authentication Enforcer-Musters durch das Spring Framework

4.4 Umsetzung des Zugriffskontrollbeschreibung und -durchsetzung am Beispiel der Arbeitsplatzsuche

Dieser Abschnitt soll die Implementierung der notwendigen Muster für die Zugriffskontrollbeschreibung und -durchsetzung am Beispiel der Arbeitsplatzsuche mit dem Spring Security Framework vorstellen. Für diesen Zweck wird die Umsetzung des Authorization Enforcer-Musters und der attributbasierten Zugriffskontrolle gezeigt. In den Grundlagen wurde bereits auf die Konfiguration einer Anwendung zur Einbindung des Spring Security-Frameworks eingegangen (siehe Kapitel 2.8.2). Auf dieser wird aufgebaut und die notwendigen Anpassungen und Implementierungen erklärt.

Die Arbeitsplatzsuche wird nicht nach dem REST-Paradigma implementiert, weshalb die Zugriffskontrolle von Methoden statt von URLs eingesetzt wird. Dies verhindert, dass URLs bei der Konfiguration vergessen werden. Um Spring Security dazu zu bringen, die Methoden auf Sicherheitsannotationen zu überprüfen und ggf. eine Zugriffskontrolle durchzuführen, muss dies in der Sicherheitskonfiguration aktiviert werden. Dies geschieht durch das XML-Element *global-method-security*. Das Element bietet die Attribute *secured-annotations* und *pre-post-annotations* an, welche zum Schutz von Methoden auf *enabled* gesetzt werden können, siehe Quellcodebeispiel 16. Auf das *expression-handler*-Element wird in einem der nächsten Abschnitten eingegangen.

Die erste Anforderung an die Zugriffskontrolle ist, dass die Anwendung nur für Studenten verfügbar sein soll. Als Vorbedingung hierfür gilt, dass sich nur Studenten authentifizieren können. Das letzte *intercept-url*-Element definiert, führt dazu, dass nur authentifizierte Benutzer auf die Anwendung zugreifen können. Durch die restlichen *intercept-url*-Elemente des Beispiels, wird der Zugriff auf die Anmelde- und Registrierungsseite, sowie unkritische Ressourcen, wie Bilder, Skripte und Formatierungselemente, erlaubt, ansonsten könnte sich kein Benutzer authentifizieren oder registrieren.

```
    <global-method-security secured-annotations="enabled" pre-post-
annotations="enabled">
          <expression-handler ref="expressionHandler"/>
    </global-method-security>


    <http auto-config="true" use-expressions="true">
          <intercept-url pattern="/login.html*"
access="isAnonymous()" />
          <intercept-url pattern="/register.html*"
access="isAnonymous()" />
          <intercept-url pattern="/css/**" filters="none" />
          <intercept-url pattern="/images/**" filters="none" />
          <intercept-url pattern="/scripts/**" filters="none" />
          <intercept-url pattern="/**" access="isAuthenticated()" />
    </http>
```

Quellcodebeispiel 16: Ausschnitt der Sicherheitskonfiguration der Arbeitsplatzsuche

Eine weitere Anforderung an die Arbeitsplatzsuche ist, dass Studenten nur ihre eigenen Reservierungen detailliert betrachten, bearbeiten und löschen können. Dies erfordert die Implementierung eines eigenen *PermissionEvaluator*, wie es im Kapitel 4.2.3 über ABAC beschrieben wird.

Die Methode, die über die Berechtigung einer Anfrage entscheidet, des *BookingPermissionEvaluator* ist auf Quellcodebeispiel 17 zu sehen. Die Reservierung wird aus der Datenbank geladen und der Besitzer mit dem aktuell authentifizierten Benutzer verglichen. Das konkrete Vorgehen wird durch die Kommentare beschrieben.

```
@Override
    public boolean hasPermission(Authentication authentication,
                Serializable targetId, String targetType,
                Object permission) {
          //reject access, if targetId is not set
          if (targetId == null || (targetId.toString().isEmpty())) {
                return false;
          }
          // allow anyone to view, edit and delete their own bookings
          if (targetType.equalsIgnoreCase("Booking")
                      && (permission.equals("edit") ||
                          permission.equals("delete") ||
                          permission.equals("view"))) {
                Reservation resv;
                try {
                      resv =
                      bookingDAO.getReservation(targetId.toString());
                } catch (Exception e) {
                      /* deny access, on retrieval error */
                      return false;
                }
                if (authentication.getName().equals(
                                        resv.getBookingUserId())) {
                      return true;
                }
          }
          return false;
    }
```

Quellcodebeispiel 17: Implementierung des *BookingPermissionEvaluator* der Arbeitsplatzsuche

Damit Spring Security den selbstimplementierten *PermissionEvaluator* benutzen kann, muss dieser durch Konfiguration bekannt gemacht werden. Die Ergänzung der Sicherheitskonfiguration zeigt Quellcodebeispiel 18. Für den *PermissionEvaluator* wird eine eigene Bean, gemäß dem Bean-Konzept von Spring (siehe Kapitel 2.8.1), definiert. Zur Auswertung von Ausdrücken, wie *hasPermission*, *ExpressionHandler* eingesetzt. Um den *PermissionEvaluator* einsetzen zu können, muss dieser einem solchen *ExpressionHandler* für Methoden hinzugefügt werden, dies geschieht bei der Definition der *expressionHandler*–Bean. Die Bekanntmachung der Bean beim Framework wurde bereits in Quellcodebeispiel 16 gezeigt. Dies geschieht durch das

Kindelement *expression-handler* des *global-method-security*-Elements, welches auf die Bean verweist.

```
<!-- Package names of classes cutted due to overview reasons -->
<beans:bean id="bookingPermissionEvaluator"
            class="[…].BookingPermissionEvaluatorImpl">
</beans:bean>
<beans:bean id="expressionHandler"
            class="[…].DefaultMethodSecurityExpressionHandler">
      <beans:property name="permissionEvaluator"
                  ref="bookingPermissionEvaluator"/>
</beans:bean>
```

Quellcodebeispiel 18: Konfiguration des selbstimplementierten *PermissionEvalutor* der Arbeitsplatzsuche

Zur Verwaltung der Reservierungen wurde die Schnittstelle *BookingInterface* implementiert. Die Schnittstelle bietet die sicherheitsrelevanten Methoden an, für die der Zugriff eingeschränkt werden soll. Das Quellcodebeispiel 19 zeigt die Verwendung der *@PreAuthorize*-Annotation zur Absicherung der *deleteReservation*-Methode. Der Ausdruck *hasPermission* wird in der Annotation so eingesetzt, dass vor der Ausführung der Methode die Autorisierung des Benutzers durch den *BookingPermissionEvaluator* geprüft wird. Im positiven Fall, wird die Methode ausgeführt und die Reservierung gelöscht.

```
@Override
@PreAuthorize("hasPermission(#reservationID, 'Booking', 'delete')")
@Transactional
public void deleteReservation(String reservationID)
                                    throws IllegalArgumentException {
    if (reservationID == null || reservationID.isEmpty()) {
          throw new IllegalArgumentException("One parameter is null
or empty");
    }
    bookingDAO.deleteReservation(reservationID);
}
```

Quellcodebeispiel 19: Benutzung einer Annotation zur Absicherung des Löschens von Reservierungen in der Arbeitsplatzsuche

Mit der gezeigten Konfiguration und Implementierung von Klassen, wurde die Arbeitsplatzsuche um das Authorization-Muster und die attributbasierte Zugriffskontrolle ergänzt. Die Anforderung, den Zugriff auf die Anwendung nur Studenten zu erlauben und, dass Reservierungen nur von Besitzern verwaltet werden dürfen, wurde durch die Implementierung erfüllt. Weitere Schritte waren das Ergänzen der Anwendung um die Möglichkeit zur Authentifizierung durch den Anwender.

4.5 Zusammenfassung der Überprüfung

In diesem Kapitel wurden eine Reihe von Sicherheitsmustern aus dem Bereich der Authentifizierung und Autorisierung betrachtet und vorgestellt. Für jedes Muster wurde überprüft, ob es mit dem Spring Security Framework umgesetzt werden kann. Für jedes unterstützte Muster wurde eine Instanziierung aufgezeigt, welche die Rollen der Klassen des Frameworks einordnete. Zudem wurden für einige Muster bewährte Implementierungsmöglichkeiten mit Konfiguration und Quellcode aufgezeigt.

Das Ziel war es, zum einen zu überprüfen, welche Muster von Spring Security unterstützt werden, und zum anderen einen Zusammenhang zwischen Sicherheitsmustern aus der Entwurfsphase und der Implementierung mit Quellcode und Konfigurationen in der Implementierungsphase herzustellen. Der erste Punkt wurde durch die Instanziierung der betrachteten Muster durch Spring Security mittels Kompositionsstrukturdiagrammen gezeigt. Der zweite Punkt ging mit der Überprüfung des Musters einher. Bei der Überprüfung wurde an Hand von Beispielen gezeigt, wie das Muster mit Hilfe des Frameworks implementiert werden kann.

Das aufgezeigte Vorgehen soll als Beispiel dienen und könnte mit anderen Sicherheits-Frameworks gleichermaßen durchgeführt werden. Auf diese Weise würde ein umfassendes Wissen gesammelt werden, dass bei dem Software-Entwicklungsprozess in der dargestellten Weise unterstützt.

Die erste Frage nach der Ableitung von Sicherheitsmustern aus Sicherheits-Frameworks, die mit dieser Arbeit bearbeitet werden sollte, konnte somit beantwortet werden. Kompositionsstrukturdiagramme sowie Konfiguration und Quellcode wurden zur Beantwortung der Frage herangezogen und das Spring Security Framework als konkretes Sicherheits-Framework betrachtet. Die zweite Frage betrifft die Modellierung von Abhängigkeiten zwischen den Sicherheitsmustern und wird im nächsten Kapitel betrachtet.

5 Abhängigkeiten zwischen Mustern in einer Sicherheitsarchitektur

Zur Unterstützung der Erstellung einer Sicherheitsarchitektur in der Entwurfsphase soll eine geeignete Methode gefunden werden, die Zusammenhänge zwischen den Sicherheitsmustern zu modellieren. Bei der Modellierung der Sicherheitsarchitektur ist es wichtig, welche Muster wie voneinander abhängen. Beispielsweise ist es wichtig, ob ein Sicherheitsmuster ein anderes voraussetzt oder ausschließt. Ebenfalls interessant ist die Hierarchie der Muster und damit einhergehend, ob ein Muster eine Verfeinerung eines Anderen ist. So könnte schrittweise ein konkreteres Modell erstellt werden.

Zur Lösung dieses Problems wird eine Mustersprache für Zugriffskontrolle vorgestellt. Die Mustersprache besteht aus Sicherheitsmustern und deren Beziehungen zueinander, um die Sicherheitsarchitektur für diesen Bereich zu entwerfen, gemäß der Idee von Alexander [Ale79:185]. Zur schrittweisen Modellierung der Sicherheitsarchitektur für die Zugriffskontrolle werden Feature Model genutzt. Durch deren Visualisierung in Form von Merkmaldiagrammen bzw. Feature Diagrams wird der Zusammenhang der Muster in der Sprache dargestellt. Die Diagramme enthalten die bereits eingeführten Sicherheitsmuster und deren Beziehungsgeflecht. Zudem soll diese Darstellung Unterstützung bei der Wahl der Sicherheitsmuster für die Architektur der konkreten Anwendung bieten.

Das Kapitel ist wie folgt gegliedert. Zuerst wird eine Einführung in Feature Models zur Darstellung von Musterzusammenhängen gegeben. Darauf folgt die Beschreibung zur Abbildung von Mustern und deren Beziehungen in die Diagramme. Abschließend wird auf die Benutzung der Diagramme in der Entwurfsphase eingegangen, die konkrete Mustersprache für Zugriffskontrolle mittels Diagrammen eingeführt und auf die Weiterentwicklung der Sprache und Diagramme eingegangen.

5.1 Feature Model zur Darstellung der Mustersprache

Dieser Abschnitt soll erläutern, warum sich der Einsatz von Feature Models zur Darstellung anbietet, welche Erweiterungen vorgenommen werden, um alle Anforderungen zu erfüllen, und wie eine Darstellung der Mustersprache mit Feature Models beispielhaft aussehen könnte. Zuerst soll jedoch auf die Notwendigkeit der Darstellung der Musterzusammenhänge und damit eines Hauptbestandteils der Sprache eingegangen werden.

„A Whole Language Says More than a Thousand Diagrams" [BHS07:287]

Dieses Zitat von Buschmann et al. stellt die Frage in den Raum, ob die angedachte Darstellung der Mustersprache für Zugriffskontrolle sinnvoll ist. Bei der Betrachtung der verschiedenen Darstellungsformen ohne formalen Hintergrund und ohne Semantik, wie in [GHJV95, SPR04:478], lässt sich der Satz untermauern. Diese Darstellungen bieten kaum Mehrwert für die Entwicklung, da die Entscheidung für oder gegen ein Muster bei der Problemlösung kaum unterstützt wird. Gleiches gilt für die Modellierung von Beziehungen zwischen den Mustern, wie bei [Zim95]. Zwar zeigen die Diagramme, wie die Muster untereinander agieren und eingesetzt werden können, der Zusammenhang zu einem Problem ist aber nicht gegeben.

Mit Merkmaldiagrammen zur Visualisierung wird ein anderes Ziel verfolgt. Die Diagramme sollen eine Hilfe bei dem Entwurf der Sicherheitsarchitektur bieten. Sie sollen Entscheidungs-

möglichkeiten zwischen verschiedenen Mustern und Idiomen aufzeigen und so eine schrittweise Problemlösung von dem Grob- über den Feinentwurf bis zur Implementierung bieten. Des Weiteren ist der Anwendungsbereich auf Authentifizierung und Autorisierung beschränkt, was die Größe des Diagramms einschränkt. Deshalb sollte die Lesbarkeit des Diagramms und der fehlende Nutzen, den Buschmann et al. anmahnen, erhalten bleiben. Zudem bieten Feature Model im Gegensatz zu den genannten Darstellungsformen eine Formalisierung, die in den folgenden Abschnitten ebenfalls angegeben wird. Dies ermöglicht beispielsweise eine Werkzeugunterstützung bei der Entwicklung.

5.1.1 Merkmale einer Produktlinie im Vergleich zu Mustern einer Mustersprache

In diesem Abschnitt sollen die Eigenschaften von Produktlinien und Mustersprachen gegenübergestellt werden. Feature Model sind ein Werkzeug aus der Software-Produktlinienentwicklung und beschreiben die veränderbaren Merkmale einer Produktlinie, beispielsweise dass ein Produkt entweder mit oder ohne Zugriffskontrolle ausgestattet werden kann. Das Ziel der Feature Model ist es, die veränderlichen Merkmale einer Produktlinie modellieren zu können und bei der Wahl einer Konfiguration eines konkreten Produkts zu unterstützen. Die Merkmaldiagramme zur Darstellung der Feature Models stellen die alternativen Merkmale einer Produktlinie grafisch als Baumstruktur dar. Für ein Produkt kann an Hand des Merkmaldiagramms eine Konfiguration nach den jeweiligen Anforderungen für ein Produkt gewählt werden.

Im Vergleich dazu bestehen Mustersprachen aus Mustern und Regeln über die Muster. Die Regeln beschreiben die Möglichkeiten zur Nutzung der Muster, um beispielsweise bei Mustersprachen für Architektur ein Gebäude oder bei Mustersprachen für Software eine Software-Architektur aus Mustern zu entwerfen. Speziell eine Mustersprache für Entwurfsmuster soll das Entwerfen einer Software-Architektur unterstützen. Grundlage für die Benutzung einer solchen Mustersprache sind die Anforderungen an die Software und die daraus resultierenden Problemen. Das Ziel bei der Benutzung der Mustersprache ist es, bewährten Lösungsansätzen für die Probleme aufzuzeigen. Bewährte Lösungsansätze beschreiben, wie Muster miteinander kombiniert werden können.

Dies zeigt bereits eine Reihe von Ähnlichkeit zwischen Produktlinien und Mustersprachen auf. Beispielsweise können Merkmale einer Produktlinie optional oder verpflichtend sein, ebenso wie Muster bei der Lösung eines Problems oder der Umsetzung eines anderen Musters. Ein Muster kann eher abstrakt sein und mehrere alternative Verfeinerungen haben, beispielsweise das Autorisierungsmuster (Kapitel 4.2.5) oder der Reference Monitor (Kapitel 4.3.3), dies ist bei Merkmalen einer Produktlinie ebenfalls möglich. Die Verfeinerungen von Merkmalen könnten sich dabei auch gegenseitig ausschließen, so dass ein Merkmal exklusiv unter vielen ausgewählt werden muss. Vergleichbares gilt auch für Muster in einer Mustersprache.

Aus den aufgezeigten Gemeinsamkeiten der Produktlinien und Mustersprachen ergibt sich die Idee der Darstellung durch Merkmaldiagramme. Eine Instanziierung eines Merkmaldiagramms, vormals Konfiguration genannt, repräsentiert Muster die zur Erstellung der Sicherheitsarchitektur für die zu entwickelnde Software benötigt werden und zeigt ihr Zusammenspiel auf.

5.1.2 Darstellung von hierarchischen Beziehungen

Merkmaldiagramme bilden einen Graphen, der im Allgemeinen einer Baumstruktur nahe kommt. Es kann zu Zyklen kommen, diese sollen in der folgenden Darstellung verhindert wer-

den. Aus dieser Struktur ergibt sich eine Hierarchie durch die verschiedenen Schichten des Baumes. Diese soll für die Darstellung der Mustersprache genutzt werden, um von abstrakten Mustern durch Verfeinerung hin zu konkreteren Mustern zu kommen. Auf diese Weise können die Sicherheitsmuster schrittweise verfeinert werden bis die implementierungsnahen Muster erreicht werden.

Auf Abbildung 35 ist diese Hierarchie schematisch dargestellt. Oben ist das Wurzelelement, das sich in zwei Muster aufspaltet, welche sich wiederum in konkretere Muster aufspalten und so weiter. Die Idee ist, dass an oberster Stelle Architekturmuster stehen, die der Software einen groben Rahmen geben. Aus den Architekturmustern ergeben sich Muster für den Entwurf, die mit den Architekturmustern in Beziehung stehen. Für die abstrakteren Muster soll es Idiome geben, die die Implementierung für eine spezielle Technologie oder Sprache beschreiben. Wenn möglich sollte sich eine klare Abgrenzung ergeben, die mit einer gestrichelten Linie hervorgehoben werden kann.

Diese Hierarchie kann nicht immer eingehalten werden, da zum einen die Differenzierung zwischen Architekturmustern und Entwurfsmustern nicht eindeutig ist [BHS07:213ff]. Es sollte aber versucht werden, Muster mit ähnlichem Abstraktionsniveau auf einer Ebene darzustellen. Zum anderen kann es durchaus vorkommen, dass ein Entwurfsmuster ein Architekturmuster benötigt. Dargestellt ist somit der Idealfall, bei dem der Entwurf immer konkreter wird.

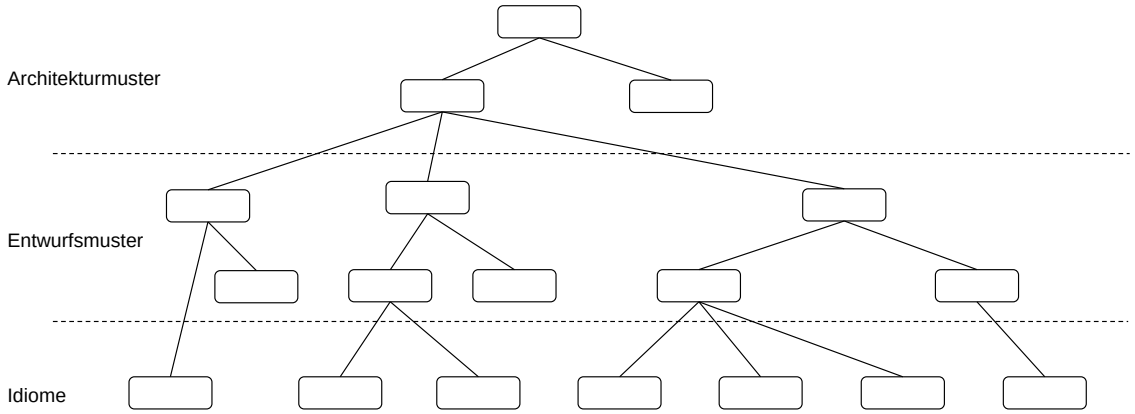

Abbildung 35: Abstraktionsebenen dargestellt durch die Baumschichten der Merkmaldiagramme

Durch die Unterteilung in verschiedene Baumschichten, wird zwischen den Abstraktionseben aller Muster unterschieden. Zur Darstellung von unterschiedlichen Abstraktionsebenen zwischen konkreten Mustern, kann eine Verfeinerungsbeziehung genutzt werden. Diese ist auf Abbildung 36 durch die gestrichelten Linien vom oberen Muster zu den unteren dargestellt. Die unteren Muster sind eine Verfeinerung des oberen Musters. Jedes der unteren Muster setzt das obere Muster um und erweitert es, vergleichbar mit dem Autorisierungsmuster, welches von dem RBAC-, IBAC- und ABAC-Muster verfeinert wird. Hinsichtlich der Abstraktionsebenen ist das obere Muster abstrakter als die unteren Muster.

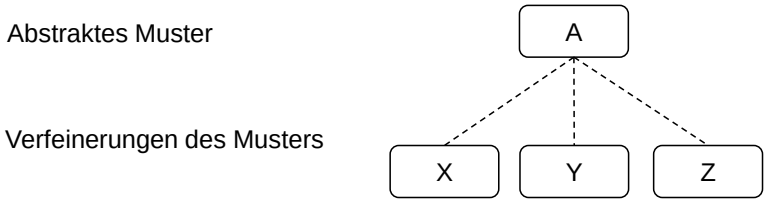

Abbildung 36: Verfeinerung eines Musters als Merkmaldiagramm

5.1.3 Darstellung von weiteren Beziehungen zwischen Mustern

Beziehungen zwischen einzelnen Mustern

Neben der Verfeinerung kann dargestellt werden, dass ein Muster ein anderes Mustern benutzt oder benutzen kann. Ein Beispiel für den Zwang der Nutzung ist das Reference Monitor-Muster, das in seiner Lösung das Autorisierungsmuster verwendet und nicht ohne dieses funktioniert. Die Beziehung zwischen dem Kompositum-Entwurfsmuster und dem Iterator-Entwurfsmuster aus [GHJV95] ist hingegen eine nicht verpflichtende Beziehung, da zum Durchlaufen der Komposition ein Iterator verwendet werden kann, aber nicht muss.

Diese Benutzt-Beziehung wird durch eine durchgezogene Linie zwischen den Mustern dargestellt. Da eine Baumstruktur vorliegt gilt, dass das Muster näher zur Wurzel der Benutzer ist und das entferntere Muster von diesem benutzt wird. Durch einen Kreis am Ende der Beziehungskante, zu sehen auf Abbildung 37, kann dargestellt werden, ob es sich um eine optionale oder obligatorische Beziehung handelt. Die Kreise an den Endpunkten können ebenfalls für Verfeinerungs-Beziehungen genutzt werden.

Ein ausgefüllter Kreis bedeutet, dass das Muster Y bei Wahl des Musters A genutzt werden kann; Y ist ein optionales Muster für A. Bei einer Verfeinerungsbeziehung bedeutet dies formal: $Y \Rightarrow A$. Bei einer Benutzt-Beziehung ist dieser Schluss nicht zwingend möglich, da das Muster auch von einem anderen Muster im Baum benutzt werden kann und somit die Implikation nicht zwingend gilt. Der ausgefüllte Kreis zeigt im Gegensatz dazu an, dass das Muster Z genutzt werden muss, wenn A ausgewählt wurde.

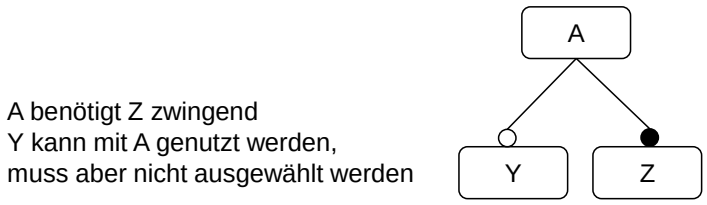

Abbildung 37: Optionale und verpflichtende Beziehungen als Merkmaldiagramm

Die nächsten beiden Beziehungen zwischen Mustern sind das Ausschließen und das Benötigen. In Abgrenzung zur Benutzt-Beziehung bedeutet die Benötigt-Beziehung ausschließlich, dass ein Muster von einem anderen zur Umsetzung benötigt wird und nicht, dass das Muster auch in der Strukturbeschreibung benutzt wird. Die Schließt Aus- und die Benötigt-Beziehungen werden durch einen gestrichelten Pfeil zwischen den betreffenden Mustern und der Beschriftung *excludes* oder *requires* dargestellt. Abbildung 38 zeigt ein Beispiel; die dargestellte Beziehung bedeutet, dass X nicht genutzt werden kann, wenn Y eingesetzt wird respektive dass W gewählt werden muss, falls Z ausgewählt wurde. Formal ausgedrückt: $\overline{(X \wedge Y)}$ bzw. $Z \Longrightarrow W$

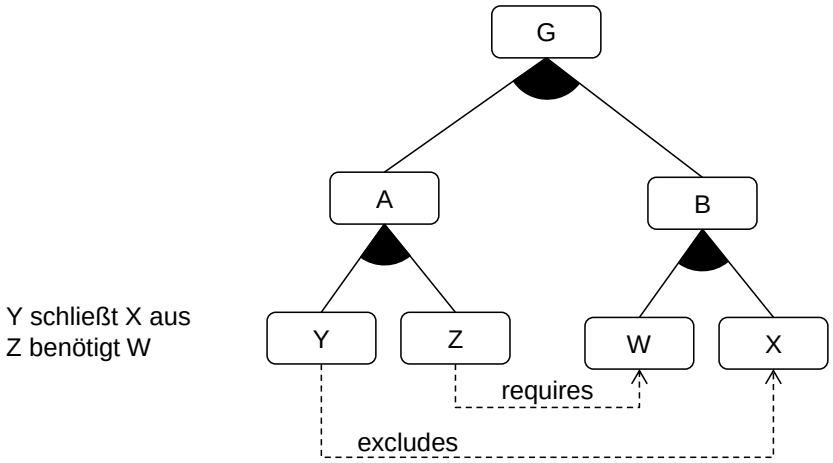

Abbildung 38: Schließt Aus- und Benötigt-Beziehung als Merkmaldiagramm

Beziehungen zwischen einem Muster und einer Gruppe von Mustern

Zur Gruppierung der Optional- und Verpflichtend-Beziehung gibt es die Und-, die Oder- und die Exklusiv-Oder-Beziehung. Mit diesen können mehrere Verfeinerungen oder Benutzungen eines Musters zusammengefasst werden, ohne an jedes Beziehungsende einen Kreis modellieren zu müssen.

Auf Abbildung 39 ist die Und-Beziehung abgebildet, die Kreise werden weggelassen. Muster X, Y und Z müssen umgesetzt werden, falls A gewählt wird. Formal ausgedrückt: $A \Longrightarrow X \wedge Y \wedge Z$

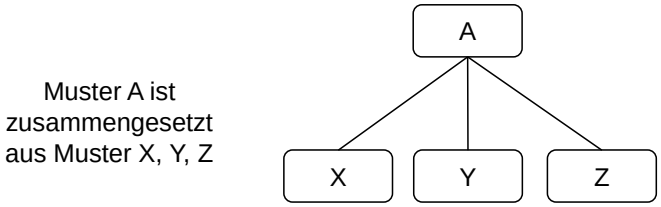

Abbildung 39: Zusammensetzung eines Musters aus anderen als Merkmaldiagramm

Neben der Und-Beziehung gibt es die Oder-Beziehung, um zu beschreiben, dass ein oder mehrere Muster benutzt werden müssen. Dies entspricht der Spezialisierung des Autorisierungsmusters. Es muss mindestens ein Zugriffskontrollmodell ausgewählt werden, es können allerdings auch mehrere eingesetzt werden. Dargestellt wird die Beziehung mit einem ausgefüllten Teilkreis, der die entsprechenden Muster einschließt, siehe Abbildung 40. Formal ausgedrückt heißt dies: $A \Leftrightarrow X \vee Y \vee Z$

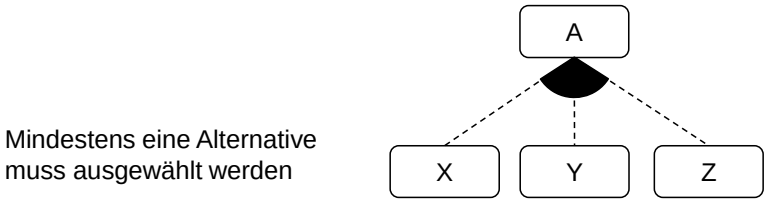

Abbildung 40: Oder-Beziehung als Merkmaldiagramm

Die Exklusiv-Oder-Beziehung stellt dar, dass nur ein Muster unter vielen ausgewählt werden muss. Dies trifft zum Beispiel für die verteilte oder zentrale Speicherung von Benutzerinformationen zu. Entweder die Daten werden für eine Anwendung zentral an einer Stelle gespeichert oder sie werden über mehrere Systeme verteilt gespeichert. Ähnlich zur Oder-Beziehung wird

ein Teilkreis verwendet, der diesmal, wie auf Abbildung 41, nicht ausgefüllt ist. Die formale Darstellung des Beispiels ist: $(A \Leftrightarrow X \vee Y \vee Z) \wedge \overline{(X \wedge Y)} \wedge \overline{(X \wedge Z)} \wedge \overline{(Y \wedge Z)}$

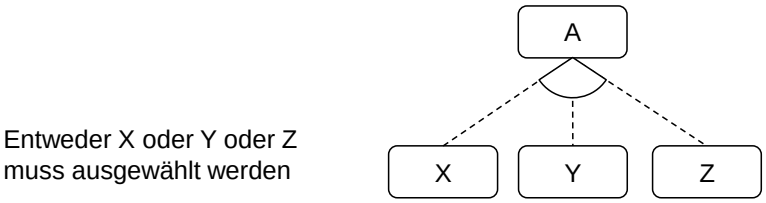

Abbildung 41: Exklusiv-Oder-Beziehung als Merkmaldiagramm

5.1.4 Erweiterung der Merkmaldiagramme um Teilbäume

Durch die Baumstruktur der Merkmaldiagramme entstehen Probleme bei der Darstellung. Unter anderem kann die Baumstruktur bei einer großen Anzahl Muster unübersichtlich werden, da die Musteranzahl je Baumlevel annähernd quadratisch steigt. Jedes verfeinerte Muster hat potentiell weitere Verfeinerungen. Zudem könnten Teilbäume an mehreren Stellen im Diagramm benötigt werden, was neben der Unübersichtlichkeit zu Redundanz führt. Die Erweiterung um Teilbäume soll diese Probleme lösen und könnte ebenfalls genutzt werden, um Bereiche mit hoher Kohärenz gesondert zu modellieren.

Die Darstellung mittels Merkmaldiagrammen zeigt Abbildung 42. Ein „+“ im rechten unteren Eck des Musters weist darauf hin, dass die Beziehungen des Musters in einem gesonderten Teilbaum weiter spezifiziert werden. Die Wurzel des Teilbaums mit der weiteren Beschreibung wird hingegen mit einem „-“ im rechten unteren Eck markiert. Der Verweis auf den Teilbaum für dasselbe Muster kann an mehreren Orten im Diagramm genutzt werden. Falls die Spezifikation im selben Diagramm vorgenommen wird, sollte der Teilbaum auf Höhe seines abstraktesten Verweises modelliert werden.

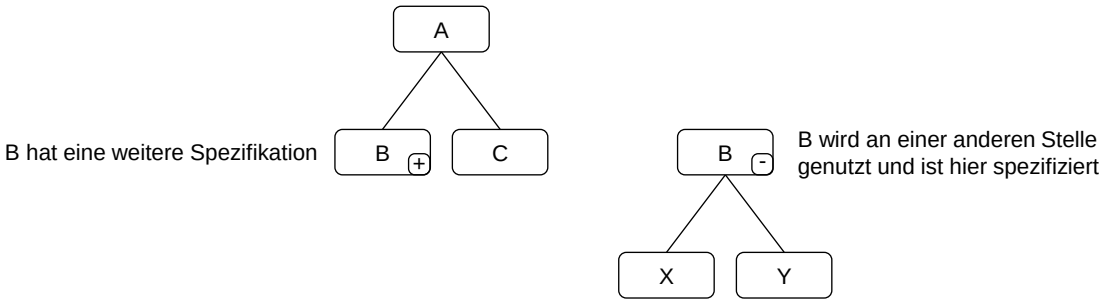

Abbildung 42 Verweis auf Spezifikation der Zusammenhänge eines Musters an einer anderen Stelle

5.2 Abbildung der Mustersprache auf Merkmaldiagramme

An Hand von Beispielen wird in diesem Abschnitt vorgestellt, wie das Wissen über Beziehungen zwischen den Mustern auf die vorgestellte Idee der Feature Models übertragen werden kann. Die Beziehungen werden beispielhaft mit Merkmaldiagrammen dargestellt, die die Grundlage für die Mustersprache für Zugriffskontrolle dienen sollen.

Als Beispiel für die Vorstellung der Abbildung werden die Informationen zu den Beziehungen zwischen den Mustern aus den Kapiteln 4.2 und 4.3 aufgegriffen und modelliert. Bei der Identifizierung der Beziehungen gibt es nicht nur bei Verfeinerungen einen gewissen Interpretationsspielraum, sondern bei allen Beziehungstypen. Der Hauptgrund für diese Problematik bei den restlichen Beziehungstypen ist ebenfalls die Beschreibung der Muster in natürlicher Sprache.

Neben der Ungenauigkeit der natürlichen Sprache, werden manche Informationen in der Beschreibung weggelassen, da diese für das Muster nicht notwendig sind. Deshalb soll das im Folgenden modellierte Wissen als erweiterbare Grundlage dienen. In Kapitel 5.3.3 wird auf diesen Aspekt vertieft eingegangen und die Evolution der Modelle thematisiert werden.

5.2.1 Abbildung des Abstraktionsniveaus der Muster

Dieser Abschnitt zeigt, wie das allgemeine Abstraktionsniveau der Muster und im Speziellen Verfeinerungen, die bei der Überprüfung auf Sicherheitsmuster beschrieben wurden, in die Darstellung mit Feature Model übertragen werden können.

Grobe Abbildung durch Hierarchiestufen

Die grobe vertikale Einteilung des Merkmaldiagramms sollte wie in Kapitel 5.1.2 beschrieben vorgenommen werden. Das heißt, die Muster sollten in Architekturmuster, Entwurfsmuster und Idiome eingeteilt werden.

Bei den vorgestellten Sicherheitsmustern wurde eine solche Einteilung bereits vorgenommen. Muster, die eine Anwendung in Subsysteme und Komponenten untergliedern, wurden in Kapitel 4.3 vorgestellt, diese sollte auf oberster Ebene dargestellt werden. In Kapitel 4.2 wurden Muster zur Umsetzung von Richtlinien untersucht. Diese beschreiben das Zusammenspiel der Klassen innerhalb der Komponenten und sind somit den Entwurfsmustern zuzuordnen und werden im Merkmaldiagramm gemeinsam in einer mittleren Ebene modelliert. Zudem wurde in den Kapiteln 4.2 und 4.3 für die Überprüfung auf eine empfohlene Implementierung des Musters eingegangen. Diese Umsetzung des Musters kann als Idiom für das Spring Security Framework angesehen werden, da es sich gemäß Buschmann et al. in [BHS07:216] um die bewährte Umsetzung mit einer bestimmten Technologie handelt. Diese Idiome werden auf unterster Ebene dargestellt.

Die einzelnen Gruppen von Mustern, Architekturmuster, Entwurfsmuster und Idiome, sollten vertikal getrennt sein. Zwischen den Gruppen sollte eine gestrichelte Linie gezogen werden, um die Einteilung hervorzuheben, wie auf Abbildung 35 zu sehen.

Abbildung von Verfeinerungen

Allgemein lässt sich die Verfeinerungsbeziehung wie zuvor gezeigt mit gestrichelten Linien zwischen den Mustern darstellen. Das abstrakte Muster wird als Wurzel des Baumes eingesetzt und die Verfeinerungen werden mit diesem verbunden und unterhalb dargestellt. Die Beziehung ist transitiv, was dazu führen kann, dass das Diagramm durch zu viele Verbindungen unübersichtlich wird, daher sollte nur eine der Verfeinerungen dargestellt werden. In der Regel ist dies die konkreteste Beziehung, jedoch sollte auch auf das Abstraktionsniveau der Muster geachtet werden und Muster mit gleichem Abstraktionsniveau auf derselben Ebene dargestellt werden. Daneben muss auch betrachtet werden, wie verschiedene Verfeinerungen eines Musters miteinander in Beziehung stehen. Darf nur eine der Verfeinerungen gewählt werden darf, ist eine Mehrfachauswahl möglich oder besteht ein anderer Zusammenhang?

Ein Beispiel für die Verfeinerungsbeziehung ist in Kapitel 4.2.5 über das Autorisierungsmuster beschrieben. Die dort genannten Verfeinerungen der Autorisierung sind die RBAC (Kapitel 4.2.1), ABAC (Kapitel 4.2.3) und IBAC (Kapitel 4.2.4). Des Weiteren wurde bei der Beschreibung des ABAC-Musters darauf eingegangen, dass dieses Muster als Verallgemeinerung des RBAC-Musters angesehen werden kann. Die rollenbasierte Zugriffskontrolle ist somit die Verfeinerung von Autorisierung und attributbasierter Zugriffskontrolle. Die Beziehung zwischen

RBAC und ABAC wird nicht dargestellt, da wegen der Transitivität nur eine Auswahl getroffen werden muss und das Abstraktionsniveau von ABAC, RBAC und IBAC ähnlich ist. Zwischen den Zugriffskontrollmodellen kann ohne Einschränkung gewählt werden und diese schließen sich nicht aus. Das resultierende Diagramm ist auf Abbildung 43 zu sehen.

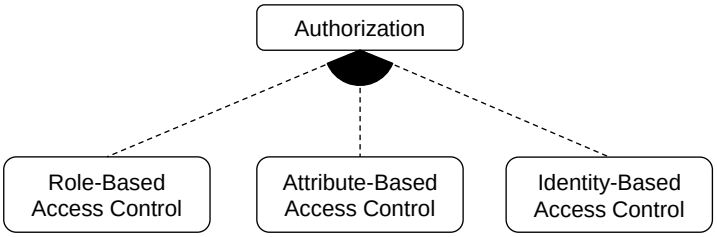

Abbildung 43: Verfeinerungen des Autorisierungsmusters mit Merkmaldiagrammen

Des Weiteren wurde auf das Benutzerspeichermuster eingegangen, dessen Verfeinerungen von der Art der Benutzerdaten abhängen, beispielsweise Benutzernamen und Passwort-Speicher oder Zertifikatsspeicher. Für die Benutzernamen und Passwort-Speicher wurden die Alternativen LDAP-Server, Datenbank und XML vorgestellt. Beim Zertifikatsspeicher sind XML und Dateisystem möglich. Zwischen allen genannten Alternativen kann frei gewählt werden. Diese Struktur zeigt Abbildung 44.

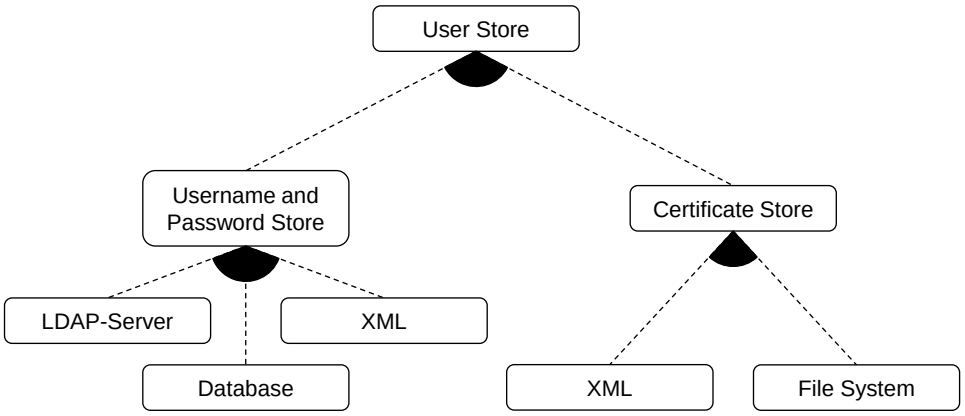

Abbildung 44: Verfeinerungen des User Store-Musters

Das nächste betrachtete Muster mit Verfeinerungen ist das Authentication Information-Muster. Dieses besteht aus den drei Bereichen Knowledge, Having und Being. Für das Knowledge-Muster gibt es wiederum zwei vorgestellte Verfeinerungen, Benutzername und Passwort sowie Zertifikate. Insgesamt ergibt sich ein Diagramm wie auf Abbildung 45.

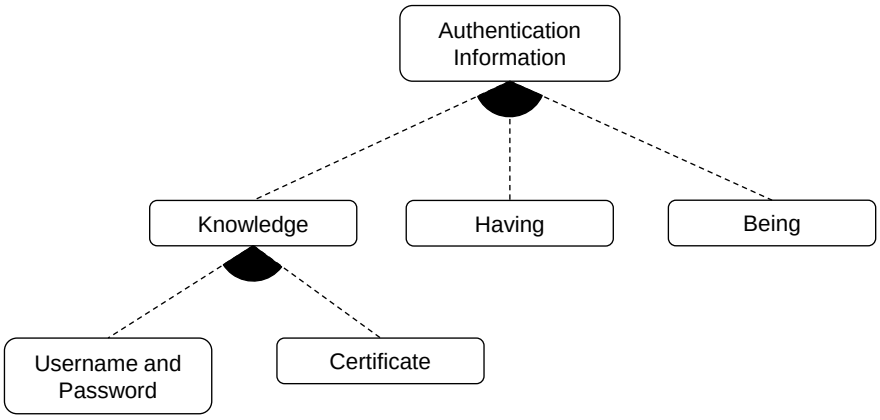

Abbildung 45: Verfeinerungen des Authentication Information-Musters

Eine weitere Verfeinerungsbeziehung besteht zwischen dem Reference Monitor-Muster und dem Authorization Enforcer- bzw. Intercepting Web Agent-Muster. Hinsichtlich der Anzahl der auswählbaren Verfeinerungen gibt es im Allgemeinen keine Einschränkung, beispielsweise kann es sinnvoll sein, ein Muster zur Durchsetzung der Zugriffskontrolle auf Web-Ebene und Systemebene zu verwenden, da diese beiden Zugriffsmöglichkeiten auf das System bestehen und eine mögliche Bedrohung darstellen. Die Beziehungsstruktur zeigt Abbildung 46 (links).

Als letzte Verfeinerung wurde der Zusammenhang zwischen Authenticator-Muster, Authentication Enforcer- und Intercepting Web Agent-Muster vorgestellt. Das Authenticator-Muster stellt das abstrakte Vorgehen zur Authentifizierung eines Subjekts dar und es macht ähnlich wie bei dem Reference Monitor-Muster Sinn, mehrere Verfeinerungen des Musters wählen zu können. Die Beziehung zwischen den Mustern zeigt das Merkmaldiagramm auf Abbildung 46 (rechts).

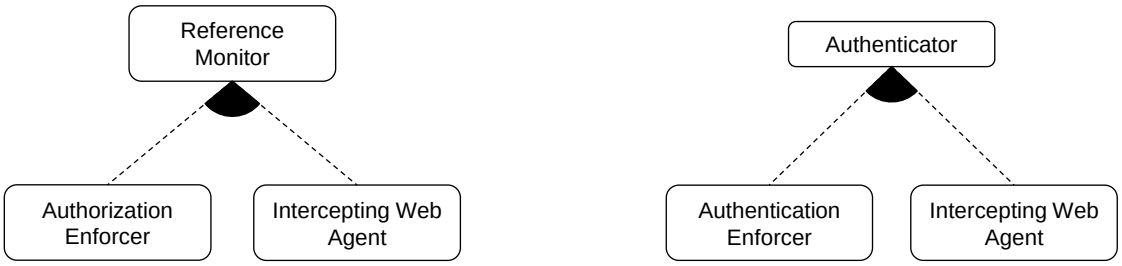

Abbildung 46: Verfeinerungen des Reference Monitor-Musters (links) und die des Authenticator-Musters (rechts) als Merkmaldiagramm

Noch nicht betrachtet wurden die Idiome der Muster. Für die vorgestellten Muster, die mit Spring umgesetzt werden können, schließen die Idiome die Verfeinerung ab. Für jede weitere bewährte Implementierungsmöglichkeit eines Musters mit einem Framework würde ein weiteres Idiom als Auswahl hinzukommen.

5.2.2 Abbildung der weiteren Beziehungen zwischen den Mustern

Neben der Abbildung der hierarchischen Beziehungen muss abgebildet werden, welche Muster sich gegenseitig benutzen, benötigen oder ausschließen. Dies wurde ebenfalls in der Überprüfung genannt und teilweise modelliert. Die obligatorische Benutzt-Beziehungen der Muster ergibt sich aus deren Beschreibung und Struktur. Wird ein anderes Muster für die Umsetzung genannt und eingesetzt, so ist dieses eine verpflichtende Benutzung.

Die Verwendung des Autorisierungsmusters durch das Reference Monitor-Muster entspricht diesem Typ von Beziehungen und ist auf Abbildung 48 zu sehen. Dieser Schluss kann auch für das Authentication Information-Muster gezogen werden, welches von dem Authenticator-Muster genutzt wird, um die Identität des Benutzers zu prüfen. Dieser Zusammenhang ist auf Abbildung 49 dargestellt.

Allerdings gibt es auch implizite obligatorische-Beziehungen, die nicht direkt aus der Beschreibung des Musters abgelesen werden können. Dies ist beispielsweise der Fall, wenn der Lösungsfokus der Muster sich unterscheidet und das benutzte Muster für die Lösung des Problems des nutzenden Musters nicht wichtig ist.

Ein Beispiel ist das Authenticator-Muster gegenüber dem Benutzerspeicher-Muster; die Authentifizierungsinformationen des Benutzers müssen bei dem Authenticator-Muster in irgendeiner Form zum Vergleich mit der Eingabe des Benutzers aus einem Speicher abgerufen werden und

somit wird auch ein Benutzerspeicher genutzt. Das Benutzerspeichermuster wird in der Beschreibung des Authenticator-Musters vermutlich nicht genannt, da der Fokus des benutzenden Musters auf der Authentifizierung liegt und die Speicherung und das Laden der Daten für das Authenticator-Muster nicht im Vordergrund steht.

Im Vergleich zur obligatorischen ist die optionale Benutzt-Beziehung meist nicht durch die Strukturbeschreibung des Musters zu identifizieren. Die Beschreibung verwandter Muster könnte eine Quelle zur Identifizierung der Beziehung sein; beispielsweise in der Beschreibung des Kompositum-Musters in [GHJV95:253] wird auf das Iterator-Muster zur Traversierung hingewiesen. Varianten eines Musters können ebenfalls auf eine optionale Benutzt-Beziehung hinweisen. Ein Beispiel ist die Nutzung des Kompositum- oder des Sitzung-Musters der Varianten des RBAC-Musters. Diesen Zusammenhang zeigt Abbildung 48.

Weiter sollen die Benötigt- und Schließt Aus-Beziehung abgebildet werden. Die Identifizierung der Beziehung zwischen den Mustern lässt abermals Interpretationsspielraum, da diese oft nicht explizit durch das Muster beschrieben werden. Die beiden Beziehungen können durch einen Überblick über die Muster und deren Beschreibung identifiziert werden.

Ein Beispiel ist das Authorization-Muster, unter anderem wird in dessen Struktur ein Subjekt mit Namen und Kennung beschrieben. Dies setzt implizit voraus, dass es ein Subjekt gibt und dass das Subjekt sich zuvor in irgendeiner Form authentifiziert hat. Das Authenticator-Muster beschreibt auf abstrakter Ebene das Vorgehen zur Authentifizierung und wird somit von dem Authorization-Muster benötigt; diese Beziehung ist auf Abbildung 48 zu sehen. Ein weiteres Beispiel sind die konkreten Authentifizierungsinformation-Muster, Zertifikat und Benutzername und Passwort, welche implizit ein konkretes Benutzerspeichermuster benötigen, den Zertifikat Speicher respektive den Benutzernamen und Passwort Speicher, wie auf Abbildung 47 modelliert.

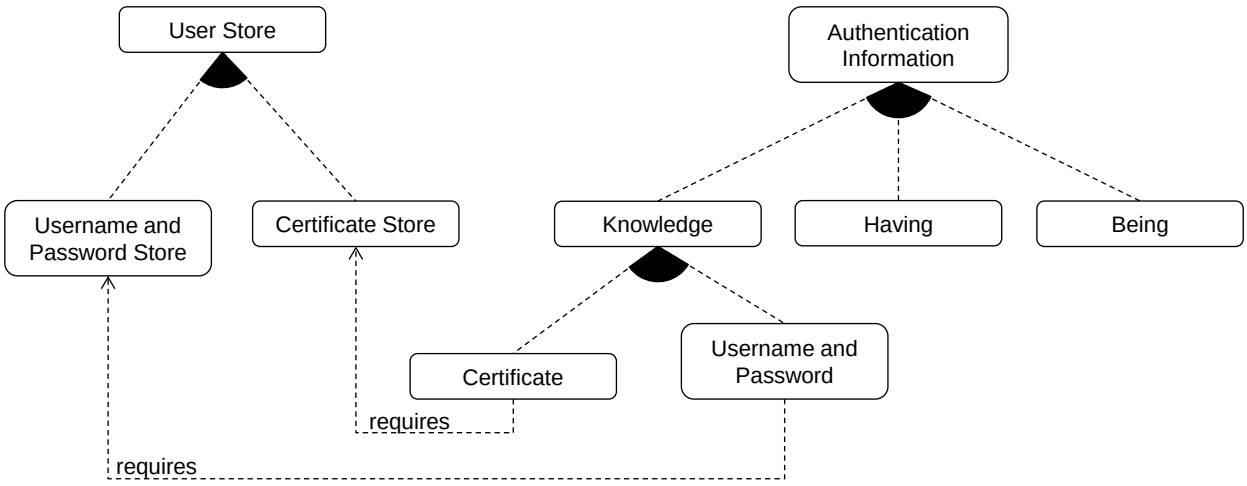

Abbildung 47: Benötigt-Beziehung zwischen Authentication Information- und User Store-Muster

5.3 Merkmaldiagramme zur Unterstützung der Entwurfsphase

In den letzten Kapiteln wurde gezeigt, wie Merkmaldiagramme zur Darstellung der Musterzusammenhänge genutzt und wie eine Abbildung von bekannten Zusammenhängen zwischen den Mustern modelliert werden kann. Darauf aufbauend wird in diesem Kapitel gezeigt, wie die Merkmaldiagramme im Entwicklungsprozess eingesetzt werden können, um die Effizienz bei der Erstellung der Sicherheitsarchitektur zu steigern und zudem Gewissheit über die Kompatibi-

lität mit verwendeten Frameworks und Software zu haben. Des Weiteren wird eine erweiterbare Mustersprache für Zugriffskontrolle mit Merkmaldiagrammen vorgestellt. Im letzten Abschnitt dieses Kapitels wird auf den Aspekt der Erweiterbarkeit eingegangen und die Evolution der Mustersprache vorgestellt.

5.3.1 Benutzung der Merkmaldiagramme

Das Ziel der beschriebenen Idee zur Nutzung von Merkmaldiagrammen zur Darstellung einer Mustersprache ist es, an Hand eines Problems respektive von Anforderung an die Software die Erstellung einer Sicherheitsarchitektur zu unterstützen und das Finden und die Wahl geeigneter Muster durch den Entwickler effizienter zu gestalten. Durch die Anforderungen an eine konkrete Software soll das Merkmaldiagramm helfen schrittweise Muster zur Lösung des Problems der Anforderung auszuwählen, indem die modellierten Zusammenhänge genutzt werden.

Als Ausgangspunkt für die Entwurfsphase dienen die Ergebnisse, darunter die Anforderungen, aus der Analysephase. Um einen einfachen Übergang von der Analyse- zur Entwurfsphase zu gewährleisten, sollten die Anforderungen in Verbindung zu Merkmaldiagrammen gebracht werden. Eine Anforderung entsteht meist aus einem Problem, weshalb in Pflichtenheften oft auch das Ausgangsproblem für eine Anforderung beschrieben wird. Zur Lösung des Problems oder zur Erfüllung der Anforderung können verschiedene Muster genutzt werden, optional oder obligatorisch. Mit Merkmaldiagrammen könnte diese Beziehung modelliert werden, indem die Anforderung beispielsweise die Wurzel des Diagramms darstellt und von diesem die notwendigen und optionalen Muster ausgehen. Aus der Anforderung ergeben sich die abstrakten Muster, die zur Erfüllung der Anforderung benötigt werden deren Beziehungsgeflecht zur Verfeinerung.

Die Ausgangsanforderung soll der Schutz von Ressourcen vor unbefugtem Zugriff, kurz die Zugriffskontrolle, sein. Diese vereint Autorisierung und Authentifizierung, da zum Schutz vor unbefugtem Zugriff sowohl die Autorisierung von Subjekten als auch die Identitätsfeststellung benötigt wird. Diese Wahl ist angelehnt an die Sicherheitsarchitektursprache von Fægri und Hallsteinsen [FH06:303], die diese Struktur ebenfalls vorsieht. Diese Anforderung soll somit als gemeinsamer Nenner für alle Anforderungen und Problembeschreibungen, die zu Autorisierung und Authentifizierung führen, dienen. Um die Zugriffskontrolle durchsetzen zu können wird auf abstrakter Ebene ein Reference Monitor benötigt [SF+06:256]. Aus dem Reference Monitor-Muster ergeben sich die weiteren notwendigen und optionalen Muster gemäß dem Merkmaldiagramm, dass durch Abbildung der Musterbeziehungen im vorigen Kapitel entstanden ist.

Mit der allgemeinen Anforderung, dass eine Zugriffskontrolle benötigt wird, beginnt der Entwurf mit dem Merkmaldiagramm an der Wurzel. An jedem Knoten des Merkmaldiagramms sollen die modellierten „Entscheidungen“ getroffen werden. Das heißt, obligatorische Muster müssen für die Architektur gewählt werden, optionale Muster können gewählt werden. Bei den Und-, Oder- und Exklusiv-Oder Beziehungen sind jeweils alle Muster, mindestens ein oder exakt ein Muster zu wählen. Gewählt bedeutet, dass das Muster in der entworfenen Sicherheitsarchitektur eingesetzt wird. Für die gewählten Muster wiederholt sich der Prozess der Auswahl, bis ein Blatt des Baumes erreicht ist. Optimal wäre es, wenn die Blätter Idiome des einzusetzenden Frameworks sind, da diese direkt implementiert werden können.

Ist das Muster optional wird die Wahl des Musters durch die Probleme der zu entwickelnden Anwendung beeinflusst. Das soll heißen, dass für jede Entscheidung im Merkmaldiagramm die zur Wahl stehenden Muster und deren Problembeschreibung und Kontext betrachtet werden müssen. Diese werden mit dem zu lösenden Problem der Anwendung verglichen und dement-

sprechend die geeignetsten Muster gewählt. Sollte keines der angebotenen Muster das vorliegende Problem brauchbar lösen, so muss entweder ein geeignetes Muster gesucht werden oder eine eigene Lösung entwickelt werden. Falls die eigene Lösung ein wiederkehrendes Problem aufdeckt, so sollte dies als eigenes Muster beschrieben. In beiden Fällen, sollte das Muster gemäß Kapitel 5.3.3 zur Sprache hinzugefügt werden.

Durch die Auswahl eines Musters soll dieses in die Architektur übernommen und in der Anwendung umgesetzt werden. Beispielsweise heißt dies für Muster, die eine Struktur beschreiben, dass diese in die Architektur der Anwendung einfließt. Beschreibt das Muster ein Verhalten, so soll die Anwendung das beschriebene Verhalten aufzeigen. Wird durch das Muster ein konzeptionelles Vorgehen beschrieben, wie bei dem Password Design and Use-Muster aus [SF+06:217], so soll dieses ebenfalls umgesetzt werden und in dem konkreten Fall beispielsweise entsprechende Maßnahmen für die Passwortvalidierung getroffen werden. Das Muster soll von der Anwendung umgesetzt werden, da es für das zu lösende Problem benötigt wird.

Die Wahl eines Musters zeigt den Zweck der Unterscheidung zwischen Verfeinerungs- und Benutzt-Beziehung auf. Die Wahl einer Verfeinerung kann die Struktur des Ganzen Musters und der Abläufe beeinflussen, das Muster erfüllt aber immer noch das gleiche Ziel, wie das gröbere Muster. Dahingegen wird bei der Wahl einer Benutzt-Beziehung lediglich eine Komponente ergänzt oder wird konkreter beschrieben. Wird eine Benötigt-Beziehung verfolgt, so ist der Zusammenhang noch weiter losgelöst, als bei einer Benutzt-Beziehung. Die Strukturen und Änderungen von benötigten Mustern haben in den meisten Fällen keinen direkten Einfluss auf die Strukturen des benutzenden Musters.

Ein Problem bei der Benutzung ist das Abstraktionsniveau der Anforderung. Beispielsweise die Anforderung der Zugriffskontrolle kann für ein Gesamtsystem, für ein Subsystem oder einzelne Komponenten betrachtet werden und es kann sinnvoll sein, die Anforderung an verschiedenen Stellen erneut zu betrachten. Beispielsweise könnte das das Gesamtsystem mit einer schwächeren Abwehrmaßnahme ausgestattet werden als einzelne sicherheitskritische Dienste des Systems. Das Reference Monitor-Muster ist ein Beispiel, dass dieses Problem aufzeigt. Der konkrete Reference Monitor kontrolliert den Zugriff auf geschützte Objekte jeder Art. Somit kann mit diesem Muster sowohl der Zugriff auf die Gesamtsoftware als auch auf einzelne Komponenten der Anwendung kontrolliert werden, z. B. wenn die Anwendung aus Sicherheitsgründen, gemäß der *Compartmentalization* aus [SF+06:301], unterteilt wird. Dies legt eine iterative Betrachtung der Anforderungen an das System, Subsysteme und deren Komponenten, und somit ein mehrfaches Anwenden der Merkmaldiagramme, nahe.

5.3.2 Mustersprache für Zugriffskontrolle als Merkmaldiagramm

Aus den Abbildungsbeschreibungen des vorigen Kapitels ergibt sich ein Merkmaldiagramm, das die Sicherheitsmuster aus Kapitel 3.5 beinhaltet und die Mustersprache für Zugriffskontrolle darstellt. Die vorgestellte Mustersprache soll eine erweiterbare Grundlage einer umfassenden Mustersprache für Zugriffskontrolle sein.

Ein Teil des Bereichs für Java Web-Anwendungen wird durch die vorgestellten Muster bereits abgedeckt und die abstrakten Muster *Reference Monitor*, *Authorization* und *Authenticator*, die den Bereich auf abstrakter Ebene technologieunabhängig beschreiben, wurden bereits in das Diagramm integriert. Diese abstrakten Muster sollten für jede Form der Zugriffskontrolle notwendig sein. Bei der Zugriffskontrolle wird immer eine Komponente benötigt, die diese durchsetzt (vgl. Reference Monitor-Muster); es müssen Berechtigungen definiert werden (vgl. Autho-

rization-Muster) und die Identität des Subjekts muss festgestellt werden (vgl. Authenticator-Muster). Die konkreten Umsetzungen werden jedoch je Anwendungsdomäne, in dem Fall von Spring Security hauptsächlich die Benutzerinterkation mit Web-Anwendungen, unterschiedlich sein.

Das Diagramm besteht aus mehreren Teilbäumen. Der Teilbaum beginnend mit der Anforderung ist auf Abbildung 48 zu sehen. Auf diesem Diagramm sind die Muster für Autorisierung und die Durchsetzung der Zugriffskontrolle dargestellt. Des Weiteren wird auf das *Authenticator*-Muster verwiesen und dass dies an anderer Stelle weiter spezifiziert wird. Die Muster für die Authentifizierung sind auf Abbildung 49 dargestellt, wobei auf zwei weitere Teilbäume verwiesen wird. Die Teilbäume der beiden Muster *Username and Password Store* und *Certificate Store* sind auf Abbildung 50 zu sehen.

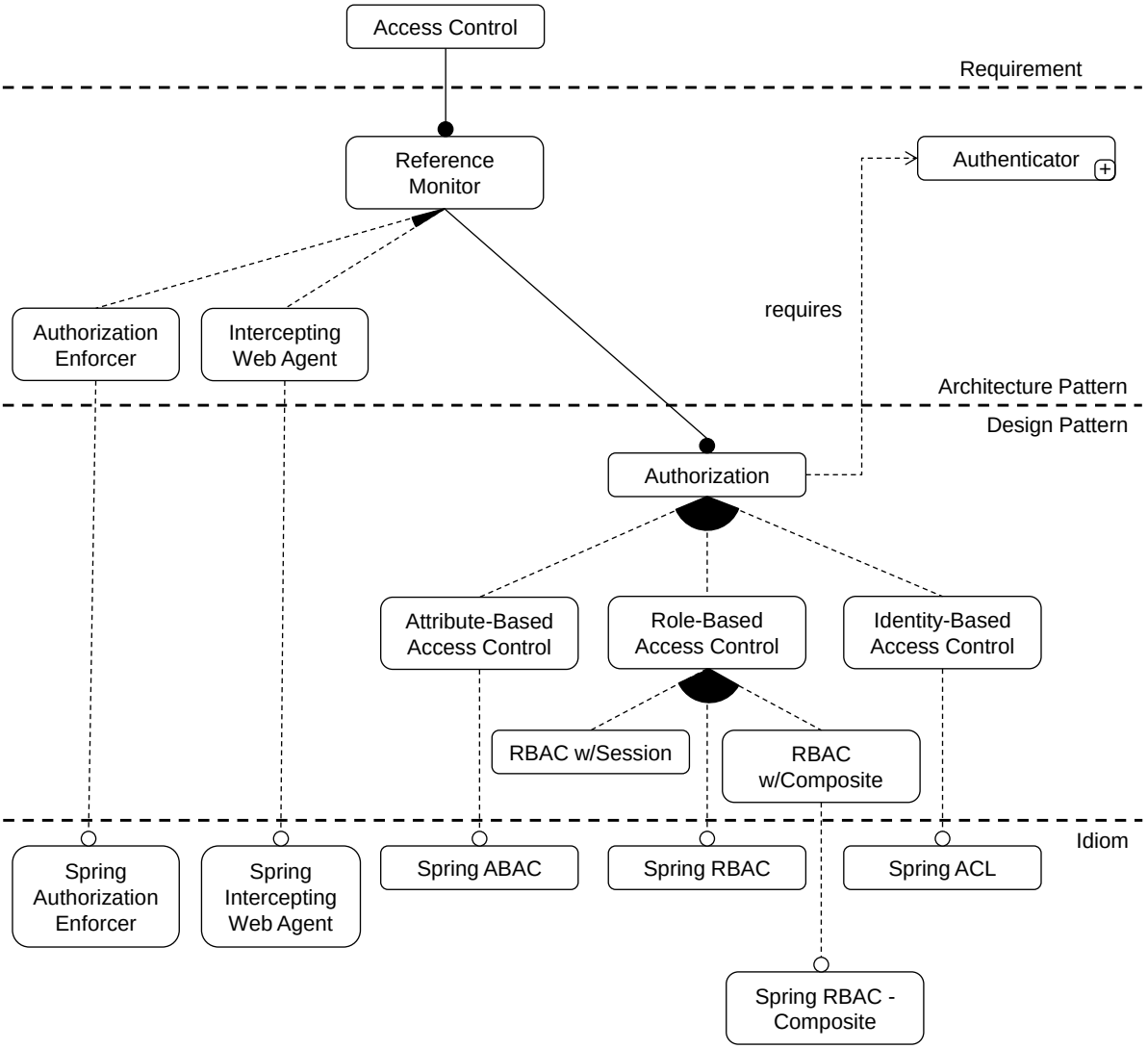

Abbildung 48: Mustersprache für Autorisierung und Authentifizierung als Merkmaldiagramm beginnend bei der Wurzel

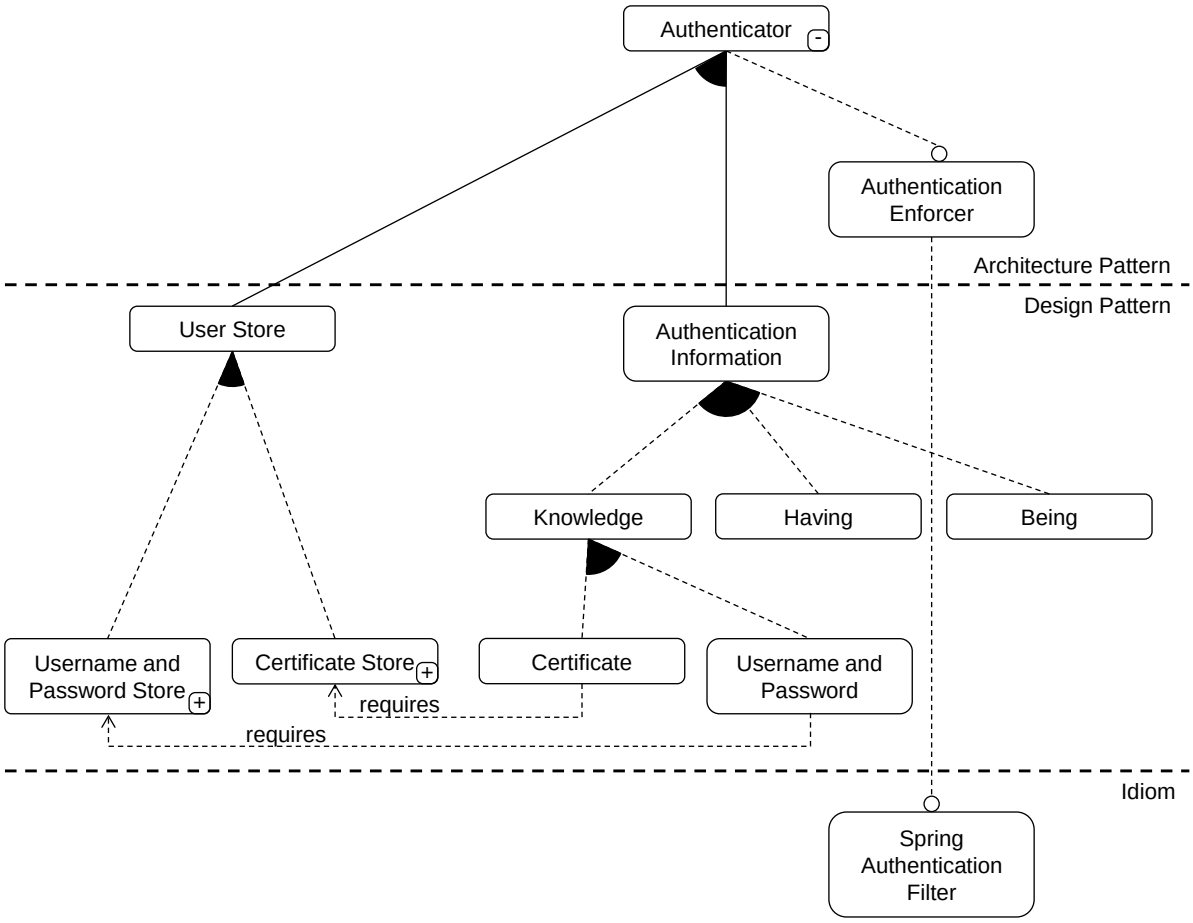

Abbildung 49: Fortsetzung der Mustersprache für Autorisierung und Authentifizierung als Merkmaldiagramm

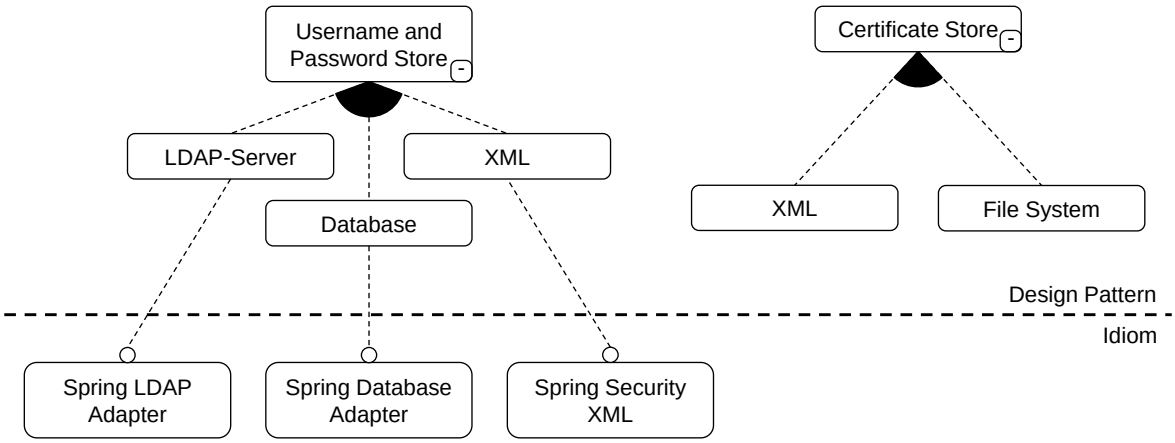

Abbildung 50: Genauere Definition der Muster Benutzername und Passwortspeicher sowie Zertifikatspeicher

5.3.3 Evolution der Mustersprache für Zugriffskontrolle

Der Fokus der vorgestellten Mustersprache liegt auf dem Entwurf der Sicherheitsarchitektur für die Zugriffskontrolle von Web-Anwendungen, die die Java Technologie einsetzen. Deshalb ist es wichtig zu zeigen, wie die Sprache und die Diagramme beispielsweise für andere Domänen oder durch Mustersprachen für den Sicherheitsbereich erweitert und angepasst werden können. Ein weiterer Grund für die Anpassung der Sprache ist die sich ändernde Umwelt. Neue Technologien entstehen und ältere werden kaum noch verwendet und gleiches geschieht analog mit den

zur Technologie passenden Mustern. Ebenfalls können Muster Anpassungen bedürfen, womit sich auch deren Beziehungsgeflecht ändern kann. Zuletzt wurden für die Vollständigkeit der Mustersprache einige Annahmen getroffen, z.B. dass es für die Muster Reference Monitor, Authenticator und Autorisierung keine abstrakteren Muster auf ihrem jeweiligen Gebiet gibt; dies muss nicht zwingend korrekt sein. Die Idee der Evolution dieser Mustersprache wurde von der Anpassung von Mustersystemen [BMR+96:373] und der Evolution der Referenzarchitektur für Sicherheit [FH06:319] übernommen.

Das Vorgehen zur Integration anderer Quellen, wie Mustersprachen und -systemen sollte etwa wie folgt aussehen. Zuerst sollten gleiche und ähnliche Elemente identifiziert werden, um deren modellierten Zusammenhang zu vergleichen. Sollten die Zusammenhänge der Quelle den bereits vorhandenen der Mustersprache entsprechen, so können die weiteren unbekannten Muster gemäß der Quelle integriert werden. Falls sich die Zusammenhänge unterscheiden, muss zur Integration ein Zusammenhang hergestellt werden. Eventuell unterscheiden sich die Musterbeschreibung der Quelle und der Mustersprache und es kann eine Variante des Musters eingeführt werden, die als Verfeinerung des Musters in die Sprache eingefügt werden kann. Es ist ebenfalls möglich, dass Beziehungen nicht modelliert wurden, da das Modell ein anderes Ziel verfolgt. In diesem Fall müsste für jedes Muster überprüft werden, in welchem Zusammenhang es zu den anderen Mustern in der Sprache steht oder ob es als eigenständiger Teilbaum für die Zugriffskontrolle notwendig ist.

Zwei unterschiedliche Quellen werden in die Mustersprache integriert. Das Mustersystem für Zugriffskontrolle von Priebe et al. aus [PF+04] wird zuerst betrachtet. In dem Artikel werden verschiedene Muster vorgestellt und in einen Zusammenhang gebracht. Zum einen die bereits in der Sprache vorhandenen Muster Autorisierung und Rollenbasierte Zugriffskontrolle und zum anderen die Metadatenbasierte Zugriffskontrolle, im englischen Metadata-Based Access Control (MBAC), mit zwei Varianten. Es wird zudem ein Composite MBAC-Muster eingeführt, das jedoch nur die beiden Varianten der MBAC vereint und somit einer Oder-Verknüpfung zwischen den Varianten entspricht. Die modellierten Beziehungen im Mustersystem entsprechen denen der Mustersprache; das MBAC-Muster ist eine weitere Verfeinerung der Autorisierung. Auf Abbildung 51 ist der relevante Ausschnitt der Erweiterung dargestellt.

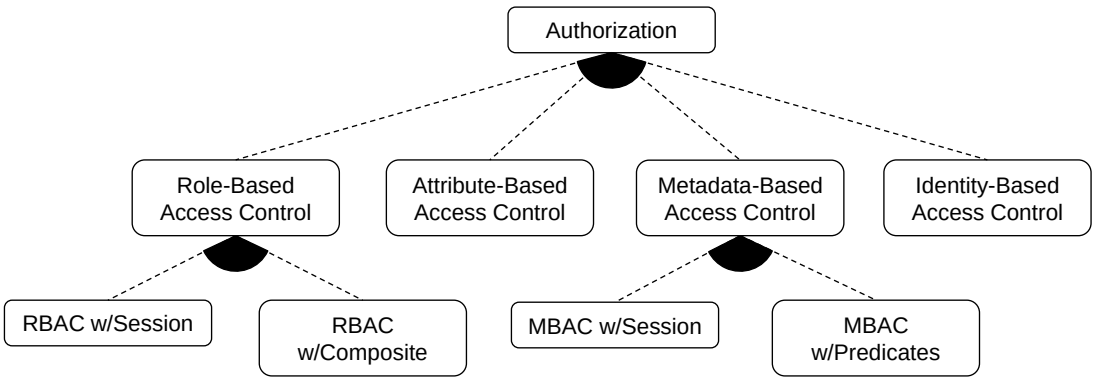

Abbildung 51: Erweiterung des Autorisierung-Musters um Muster aus [PF+04]

Als zweites sollen die Muster aus der Sicherheitsarchitektursprache von Fægri und Hallsteinsen aus [FH06:303] in die vorliegende Darstellung übernommen werden. Die Struktur der Taktiken und Muster ist hierarchisch aufgebaut. Der für die Sprache relevante Ausschnitt über Authentifizierung und Autorisierung ist auf Abbildung 13 zu sehen. Die Beziehung von Zugriffskontrolle zu den Mustern Authentifizierung und Autorisierung ist ähnlich dem der Mustersprache

Es gibt allerdings einige Unterschiede zwischen der Sicherheitsarchitektursprache und der Mustersprache aus Kapitel 5.3.2. Das Reference Monitor-Muster ist nicht unter der Zugriffskontrolle modelliert. Die Sprache von Feagri und Hallsteinsen soll bei der Erstellung einer Sicherheitsarchitektur unterstützen, somit ist es wichtig die Wahl zwischen einem verteilten Zugriffsschutz (*Compartmentalization*) oder einem Zugriffsschutz nur an einer Stelle (*Single Access Point*) aufzuzeigen. Beide Lösungstaktiken werden vermutlich aus diesem Grund als Präventionstaktik aufgeführt und können ebenfalls, wie in der Mustersprache für Zugriffskontrolle, modelliert werden. Zudem ist der Zusammenhang von Reference Monitor zur Autorisierung nicht dargestellt. Die Darstellung der Sicherheitsarchitektursprache sieht keine Modellierung von Benutzungen vor, weswegen die Beziehung zur Autorisierung vermutlich weggelassen wurde.

Weiterhin wird nicht das Authenticator-Muster, sondern die Authentifizierung als Muster in der Sprache aus [FH06:303] genutzt. Der Authenticator beschreibt den Ablauf und die benötigten Komponenten zur Authentifizierung, somit benötigt das Authentifizierungsmuster den Authenticator. Das Authentifizierungsmuster nimmt die Position des Authenticator-Musters ein.

Ein weiterer Unterschied besteht darin, dass Authentifizierung und Autorisierung direkt mit der Zugriffskontrolle verbunden sind. Dies wäre in der Mustersprache ebenfalls denkbar, da jeder Kindknoten der Zugriffskontrolle sowohl Authentifizierung als auch Autorisierung benötigt oder benutzt. Die Modellierungen entsprechen sich in diesem Bereich und die weiteren Muster können in die Mustersprache übernommen werden. Zwischen den Unterelementen kann jeweils ausgewählt werden, weshalb eine Oder-Beziehung genutzt werden kann. In [FH06:303] wird das Muster *Contextual* eingeführt, welches eine Kategorisierung der Verfeinerungen des Autorisierungsmusters vorsieht. Bei der geringen Anzahl von Verfeinerungen ist eine solche Kategorisierung nicht notwendig, weswegen diese erst einmal nicht modelliert wird. Die neuen Zusammenhänge sind auf Abbildung 52 und Abbildung 53 dargestellt.

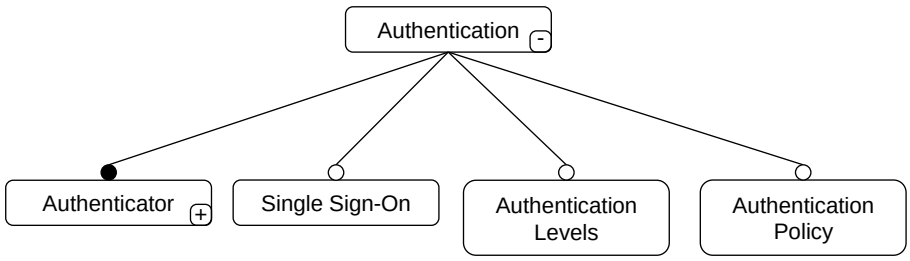

Abbildung 52: Ergänzung des Authenticator-Musters um Sicherheitsarchitektursprache aus [FH06:303]

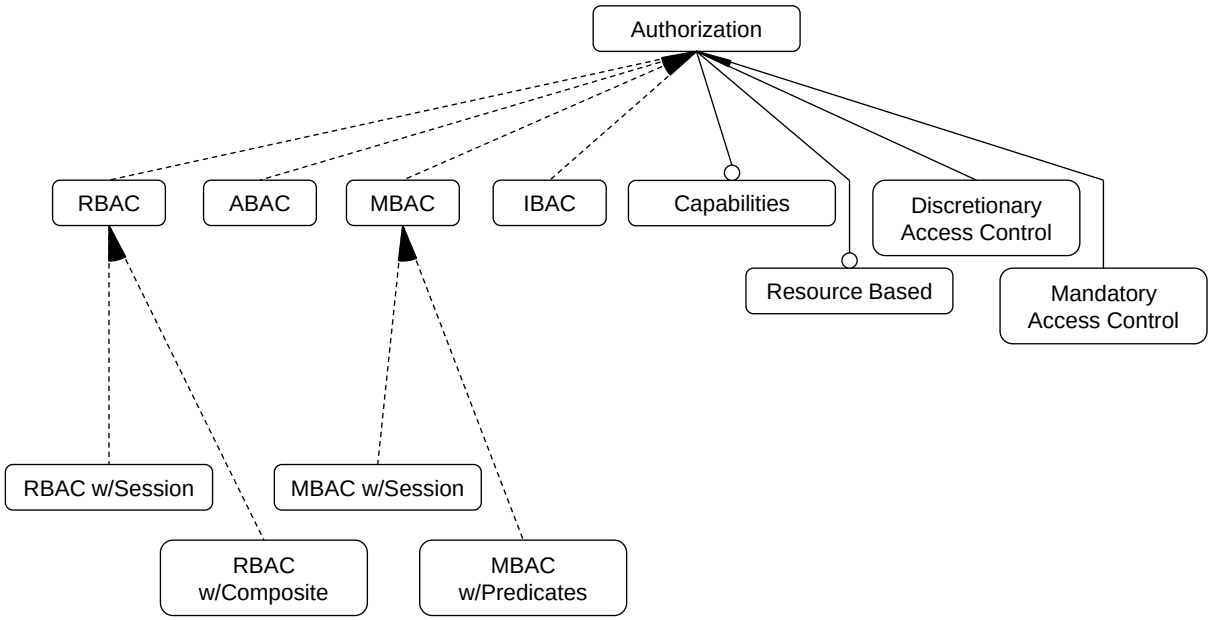

Abbildung 53: Ergänzung des Autorisierungsmusters um Sicherheitsarchitektursprache aus [FH06:303]

Mit diesen unterschiedlichen Beispielen wurden einige Probleme, die beim Zusammenführen von verschiedenen Modellen auftreten können, exemplarisch aufgezeigt. Die Zielsetzung des Modells hat einen Einfluss auf die Darstellung, ebenso wie der Interpretationsspielraum der Muster. Dies muss bei der Integration beachtet werden und führt möglicherweise zu einer Neustrukturierung der Merkmaldiagramme. Es sollte immer im Blick gehalten werden, wer die Zielgruppe der Mustersprache ist und wozu diese dient, weswegen es sinnvoll sein kann, Informationen, wie beispielsweise das *Contextual*-Muster, nicht zu modellieren. Bei der Evolution der Mustersprache sollten zudem die Frameworks im Blick behalten werden. Sollten neue Muster integriert werden, so sollte ebenfalls jedes Framework auf diese überprüft werden, um die Vollständigkeit der Mustersprache hinsichtlich der Idiome zu gewährleisten.

5.4 Benutzung der Merkmaldiagramme am Beispiel der Arbeitsplatzsuche

In diesem Abschnitt wird die Benutzung der Merkmaldiagramme an einem konkreten Beispiel gezeigt. Das Problem der Autorisierung der Arbeitsplatzsuche wird genutzt, um den Ausschnitt der Sicherheitsarchitektur der Arbeitsplatzsuche mit der vorgestellten Mustersprache zu entwerfen. Die Merkmaldiagramme der Mustersprache aus Kapitel 5.3.2 werden als Grundlage für das weitere Vorgehen eingesetzt.

Zuerst zur Problembeschreibung und den Anforderungen der Arbeitsplatzsuche. Das Ziel ist die Entwicklung einer Web-Anwendung. Als zentrales Framework soll Spring Security zur Absicherung verwendet werden. Mit der Anwendung sollen Studenten Arbeitsplätze auf dem Campus suchen und Reservieren können. Es sollen einzelne Arbeitsplätze zum Lernen, Gruppenarbeitsplätze und Räume zum Üben von Vorträgen angeboten werden. Ein Arbeitsplatz kann verschiedene Eigenschaften haben, beispielsweise einen Internetanschluss oder Steckdosenanschluss besitzen. Die Verwaltung der Räume und Benutzer ist vorliegend nicht Teil der Anwendung, ebenso wie Räume oder Bereiche, die geschützt sind und nicht öffentlich gefunden werden sollen.

Allgemein soll der Zugriff auf die Anwendung nur für Studenten möglich sein. Als sicherheitskritischer Teil wurde die Verwaltung der Reservierungen identifiziert. Die Anforderungen in diesem Bereich sind, dass Studenten nur eigene Reservierungen detailliert anschauen, bearbeiten und löschen können sollen. Daneben soll jeder Student Reservierung vornehmen können. Der Zugriff auf Reservierungen muss somit kontrolliert werden.

Gemäß dem Merkmaldiagramm auf Abbildung 48 muss ein Reference Monitor benutzt werden. Der Reference Monitor bietet kein Idiom für das Spring Security Framework an, deswegen ist eine weitere Verfeinerung notwendig. Als Verfeinerung wird das Intercepting Web Agent-Muster und das Authorization Enforcer-Muster angeboten. Der Fokus des Intercepting Web Agent-Musters liegt auf der nachträglichen Ergänzung von Authentifizierung und Autorisierung, dies ist für die vorliegende Anwendung nicht notwendig. Die Trennung der Anwendungs- von der Sicherheitslogik bietet auch das Authorization Enforcer-Muster an. Zudem wird durch dieses Muster die Authentifizierungs- und Autorisierungslogik an einer zentralisiert, was für die vorliegende Anwendung sinnvoll ist, falls zukünftig weitere Anwendungsteile geschützt werden sollen. Das Authorization Enforcer-Muster und sein Idiom für das Spring Security-Framework wird somit als Verfeinerung gewählt. Vorläufig sieht die Musterauswahl wie auf Abbildung 54 aus.

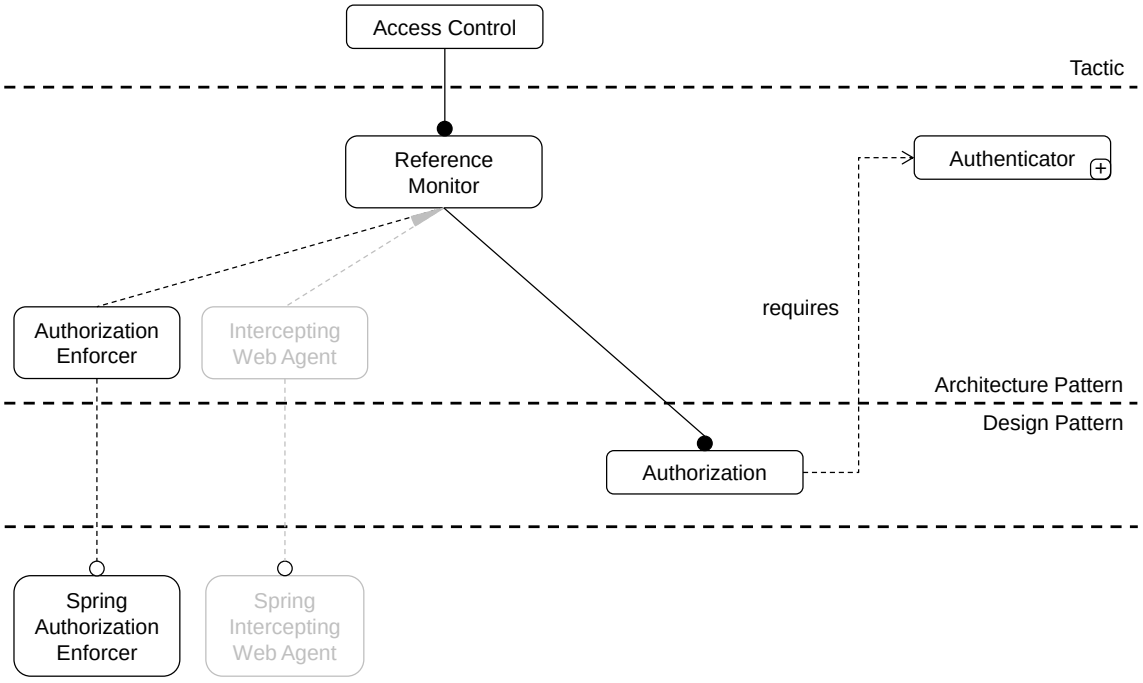

Abbildung 54: Merkmaldiagramm der Musterauswahl für die Arbeitsplatzsuche nach der Auswahl der Verfeinerungen des Reference Monitor-Musters

Neben den Verfeinerungen verpflichtet das Reference Monitor-Muster die Benutzung des Autorisierungsmusters, welches wiederum das Authenticator-Muster benötigt. Bevor in einen anderen Teilbaum gewechselt wird, soll die Auswahl des vorliegenden beendet werden. Das Autorisierungsmuster bietet kein direktes Idiom an und muss verfeinert werden. Zur Auswahl stehen die attributbasierte, rollenbasierte und identitätsbasierte Zugriffskontrolle, wobei mindestens eine ausgewählt werden muss. Die rollenbasierte Zugriffskontrolle ist für das vorliegende Problem nicht anwendbar, da die Autorisierung für einzelne Objekte einer Klasse vergeben werden soll. Die identitätsbasierte Zugriffskontrolle könnte verwendet werden, allerdings ist diese im Vergleich zum attributbasierten Ansatz aufwendiger in der Verwaltung und letzterer wäre zu bevorzugen. Mit der attributbasierten Zugriffskontrolle könnten die aufgezeigten Anforderun-

gen modelliert werden. Wenn die Kennung des Benutzers gleich der Kennung des Besitzers der Reservierung ist, so darf dieser die Reservierung anzeigen, löschen und bearbeiten. Die Wahl fällt somit auf die attributbasierte Zugriffskontrolle und deren Idiom für das Framework.

Die Alternativen im Teilbaum wurden gewählt und das Resultat zeigt Abbildung 55. Die resultierende Architektur ist auf Abbildung 56 dargestellt. Der Entwurf der Sicherheitsarchitektur für Autorisierung und deren Durchsetzung wurden beispielhaft aufgezeigt. Die Implementierung der ausgewählten Muster wurde bereits in Kapitel 4.4 vorgestellt. Der Entwurf und die Implementierung der Authentifizierung könnte analog zu dem gezeigten Vorgehen vorgenommen werden.

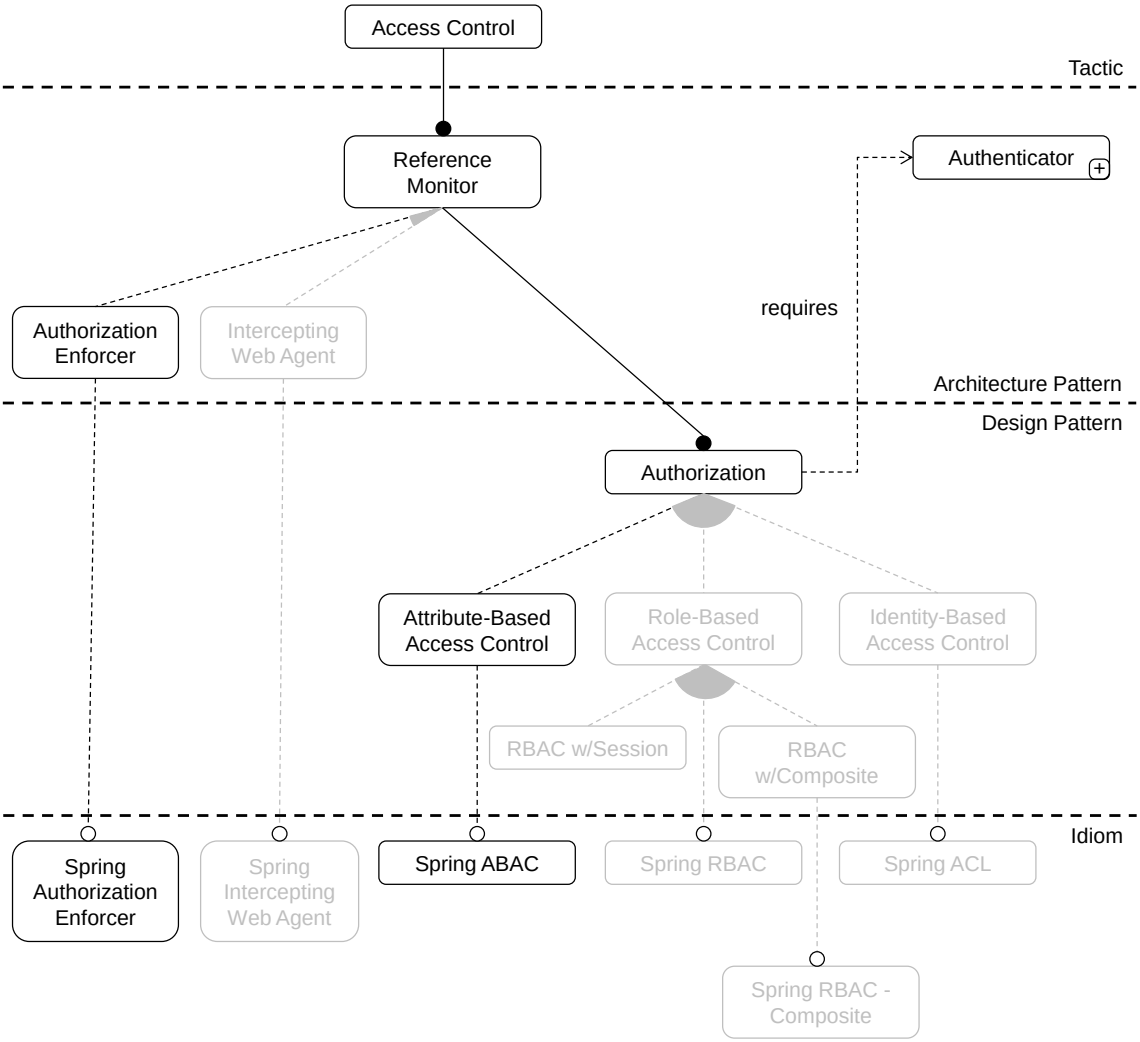

Abbildung 55: Merkmaldiagramm der Musterauswahl für die Arbeitsplatzsuche nach der Auswahl der Verfeinerungen des Autorisierungsmusters

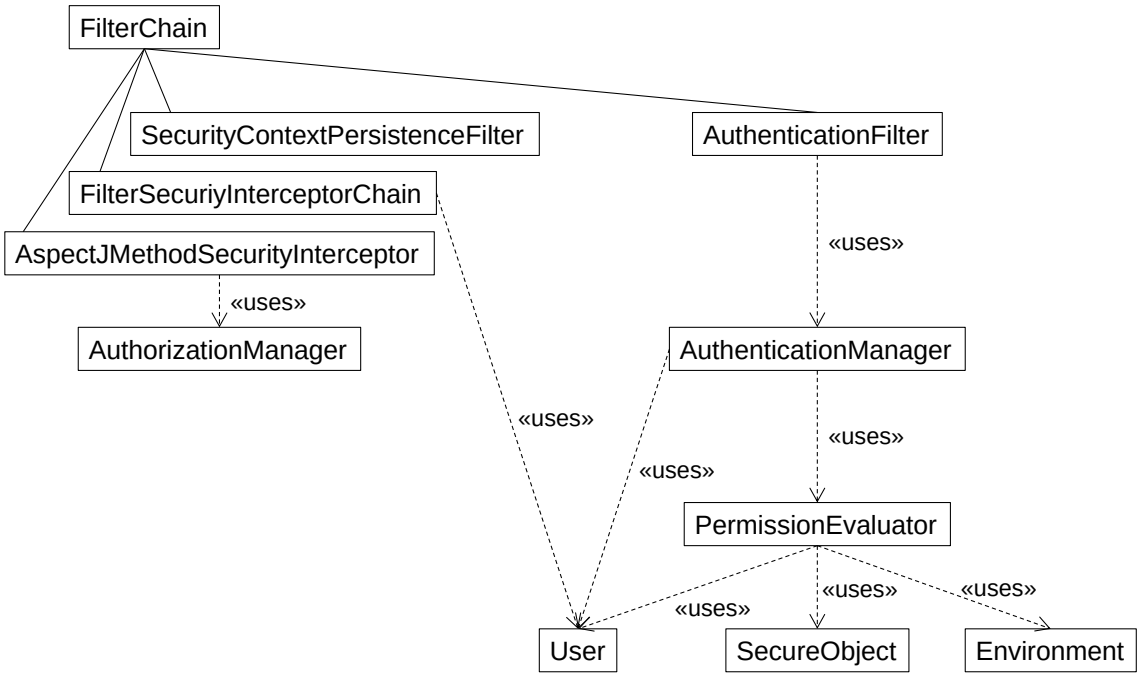

Abbildung 56: Architektur der Arbeitsplatzsuche nach der Auswahl der Verfeinerungen des Autorisierungsmusters

5.5 Zusammenfassung

In diesem Kapitel wurden die Muster aus dem vorigen Kapitel und ihre Beziehungen in einem Merkmaldiagramm dargestellt. Dabei wurde zuerst auf Mustersprachen und Merkmaldiagramme eingegangen und eine formalisierte Form der Darstellung eingeführt. Durch diese Formalisierung ist es möglich zu prüfen, ob eine Auswahl an Mustern laut einem Merkmaldiagramm umsetzbar ist. Zudem können die Merkmaldiagramme genutzt werden, um die Muster für eine Sicherheitsarchitektur einer konkreten Anwendung auszuwählen und deren Zusammenspiel für die Modellierung zu sehen. Die Muster des vorigen Kapitels wurden mit Abbildungsvorschriften in Merkmaldiagramme überführt und sollen die Grundlage für eine Mustersprache für Zugriffskontrolle bilden. Da die konkreteren Muster aus Kapitel 4.3 spezifisch für die Java Enterprise Edition sind, wurde darauf eingegangen, wie die Mustersprache um weitere Muster für andere Domänen und Technologien ergänzt werden kann.

Es wurde gezeigt, dass sich die eingeführte Interpretation der Merkmaldiagramme zur Darstellung von Musterbeziehungen respektive einer Mustersprache eignen und zur Unterstützung bei der Entwicklung der Sicherheitsarchitektur eingesetzt werden können. Im folgenden Kapitel wird das Konzept der Merkmaldiagramme eingesetzt, um beim Entwurf der Architektur einer Anwendung zu unterstützen und es wird evaluiert, ob sich die Diagramme für diesen Zweck eignen und einen Mehrwert hinsichtlich der Effizienz bieten.

6 Zusammenfassung und Ausblick

Als Abschluss dieser Arbeit wird gemäß den anfänglichen Fragestellungen eine Zusammenfassung der vorgestellten Ideen und Konzepte gegeben. Darüber hinaus werden mögliche Ansatzpunkte für weitere Arbeiten und Erweiterungsmöglichkeiten der vorgestellten Konzepte gegeben.

6.1 Zusammenfassung

An Hand der anfänglich aufgestellten Fragen soll auf die Ergebnisse dieser Arbeit eingegangen werden. Diese lauten:

1. Welche Sicherheitsmuster werden von Sicherheits-Frameworks unterstützt und wie werden diese implementiert?
2. Wie könnten Beziehungen zwischen Sicherheitsmustern modelliert werden?

Durch die Betrachtung eines konkreten Sicherheits-Frameworks wurde der Beantwortung der ersten Frage nachgegangen. Exemplarisch wurde gezeigt, wie das Spring Security Framework auf Sicherheitsmuster überprüft werden kann. Die Rollen und Abläufe der Sicherheitsmuster wurden mit Diagrammen der UML modelliert. Für das Sicherheits-Framework wurde anschließend geprüft, ob es für jede Rolle eines Musters Repräsentanten in Form von Klassen oder Konfigurationsdateien gibt und ob die modellierten Abläufe durch das Framework entsprechend den Vorgaben des Musters umgesetzt wurden. Einhergehend mit dieser Überprüfung wurde durch Quellcode und Konfigurationsdateien für einen Teil der Muster gezeigt, wie diese mit dem Framework auf bewährte Weise umgesetzt werden können.

Durch die Beantwortung der ersten Frage wurde zugleich die Lösung des Problems angegangen, ob ein Entwurf mit den Mitteln eines bestimmten Frameworks umgesetzt werden kann. Dies ist allerdings derzeit nur eingeschränkt für das Spring Security Framework und die Auswahl an in dieser Arbeit betrachteten Mustern möglich. Des Weiteren stellte sich die Frage, ob Altsysteme durch das eingesetzte Framework beachtet und weiterverwendet werden können. Die Einbeziehung von Technologien als Muster, wie LDAP-Server, in die Merkmaldiagramme kann dieses Problem lösen. Gibt es ein Idiom für das Framework für eine bestimmte Technologie, so wird diese von dem Framework auf jeden Fall unterstützt.

Zur Beantwortung der zweiten Frage wurden verschiedene Ansätze zur Darstellung von Mustern betrachtet. Schlussendlich wurde eine Darstellungsform aus dem Bereich der Softwareproduktlinienentwicklung herangezogen, die Merkmaldiagramme, und an die Bedürfnisse der Modellierung von Musterzusammenhängen angepasst. Für die entstandenen Diagramme wurde die Notation erklärt und die Erstellung mit Hilfe der Muster aus Kapitel 4 und ihrer Zusammenhänge beschrieben.

Das identifizierte Problem hinter der zweiten Frage ist, dass keine formalisierte Modellierungsmöglichkeit bekannt war, mit der einem Entwickler Musteralternativen aufgezeigt werden konnten. Durch diese Modelle sollte es nicht nur erfahrenen Entwicklern möglich sein, eine Sicherheitsarchitektur zu erstellen. Die vorgestellten Diagramme lösen dieses Problem. Sie zeigen Alternativen beim Entwurf auf und beschreiben in ihrer Gesamtheit, wie die Muster in Kombination miteinander genutzt werden können.

Durch diese Anleitung zur Benutzung der Muster entstand eine erweiterbare Mustersprache für Autorisierung und Authentifizierung. Die Benutzung der Diagramme für diesen Zweck wurde auf abstrakter Ebene beschrieben und anschließend am Beispiel der Arbeitsplatzsuche das konkrete Vorgehen demonstriert. Des Weiteren wurde beispielhaft gezeigt, wie Muster und deren Beziehungen in die Diagramme integriert werden können, um Muster aus anderen Domänen zu integrieren und eine Anpassung an die sich stetig ändernde Umwelt zu gewährleisten.

6.2 Ausblick

Zuerst soll auf Sicherheitsmuster in Frameworks als Grundlage für weitere Arbeiten eingegangen werden. Ein Ziel der Arbeit ist es, den Zusammenhang zwischen Entwurf und Implementierung von Sicherheitsmustern herzustellen und zu unterstützen. Deshalb könnten weitere Arbeiten die Erweiterung des Wissens über Muster in Sicherheits-Frameworks anstreben. Weitere Frameworks sollten auf Sicherheitsmuster, die mit ihnen umgesetzt werden können, untersucht werden.

Mit einem umfangreichen Katalog von Frameworks und deren Idiomen wäre es ebenfalls möglich, ein geeignetes Framework für eine Auswahl an Sicherheitsmustern zu finden. Jedes Framework, das ein Idiom für jedes ausgewählte Muster anbietet, wäre ein Kandidat für den Einsatz in der Implementierungsphase. Dies hat zur Folge, dass Kriterien zum Vergleich von Frameworks gefunden werden müssen, damit der Entwickler zu einer schnellen Entscheidung kommen kann. Bei diesem Vorgehen könnte beispielsweise die Kompetenzen des Entwicklerteams und die Laufzeit der abgesicherten Anwendung mitbetrachtet werden.

Ein Problem für Idiome eines Frameworks stellt die stetige Weiterentwicklung dar. Für eine neue Version eines Frameworks ist nicht abzusehen, ob die Idiome der alten Version weiterhin gelten und somit, ob die bisher unterstützen Muster weiterhin umgesetzt werden können. Es stellt sich die Frage, wie Frameworks sich weiterentwickeln und was mit Mustern in diesem Zusammenhang geschieht. Anzunehmen ist, dass ein bisher unterstütztes Muster in einer neuen Version eines Frameworks weiterhin unterstützt wird, aber das Idiom sich verändert, beispielsweise weil Schnittstellen des Frameworks sich geändert haben. Weiterhin ist es möglich, dass neue Idiome für bisher nicht unterstützte Muster entstehen, zum Beispiel weil eine neue Technologie oder ein neuer Standard in das Framework integriert wurde und der Funktionsumfang sich erweitert hat.

Eine Ausweitung des Konzepts auf den funktionalen Teil der Softwareentwicklung ist ebenfalls denkbar. Das Spring Framework bietet eine Reihe von Projekten, die auf die Unterstützung von Mustern überprüft werden könnten. Beispielsweise bietet Spring MVC eine Unterstützung für das Model-View-Controller-Muster. Bewährte Implementierungsmethoden der Muster könnten ebenfalls in diesem Bereich beim Softwareentwicklungsprozess helfen.

Wird die Modellierung von Mustersprachen mit Merkmaldiagrammen betrachtet, so ergeben sich weitere Ansatzmöglichkeiten. Es sollte untersucht werden, ob dieser Ansatz einen Mehrwert für die Softwareentwicklung, beispielsweise durch einen zeiteffizienteren oder sichereren Entwurf, bietet. Dies könnte in einer Studie festgestellt werden, bei der der konventionelle Ansatz mit dem vorgestellten Vorgehen verglichen wird.

Weitere Arbeiten könnten automatische Transformationen der Modelle anstreben. Einerseits könnte untersucht werden, inwieweit es möglich ist, die Änderung der Architektur bei der Auswahl eines Musters automatisiert vorzunehmen. Andererseits könnte der Übergang von den

Idiomen zur Implementierung formalisiert werden und automatisiert Quellcode und Konfiguration generiert werden. Das Stichwort ist die modellgetriebene Softwareentwicklung. Der Zusammenhang zwischen einer Mustersprache basierend auf Merkmaldiagrammen und der modellgetriebenen Softwareentwicklung könnte untersucht werden.

Eine Prämisse dieser Arbeit ist, dass Sicherheit im Entwicklungsprozess von Beginn an betrachtet wird. Dies lässt ebenfalls Platz für weitere Arbeiten. Kann ein bestehendes System mit Merkmaldiagrammen beschrieben werden? Wenn ja, können Merkmaldiagramme ebenfalls bei der Erweiterung oder Änderungen der Software unterstützen, beispielsweise wenn das Zugriffskontrollmodell von rollenbasierter auf attributbasierte Zugriffskontrolle geändert werden soll?

Eine weitere Einschränkung betraf die Phasen der Softwareentwicklung, es wurden ausschließlich die Entwurfs- und Implementierungsphase betrachtet und die Analysephase als Ausgangspunkt angerissen. Eventuell können Merkmaldiagramme ebenfalls in diesen früheren Phasen eingesetzt werden, um Anforderungen zu modellieren und in einen Zusammenhang zu bringen? Dies könnte einen Übergang der Phasen erleichtern, indem dieselben Konzepte genutzt und eventuell mit automatischen Transformationen gearbeitet werden kann.

Abschließend soll auf die Möglichkeit zur Entwicklung eines Werkzeugs hingewiesen werden. Das Werkzeug könnte bei der Entwicklung einer Sicherheitsarchitektur unterstützen, indem es die Merkmaldiagramme als Grundlage benutzt. Die einzelnen Entscheidungen, die getroffen werden sollen, werden durch das Werkezeug angeboten und zusätzlich werden Informationen über die Muster, wie Kontext, Problembeschreibung oder Konsequenzen, angezeigt. Die entstehende Sicherheitsarchitektur könnte durch das Werkzeug dargestellt werden. Weiter könnten die Muster, die zur Auswahl stehen, durch eine Filterfunktion eingeschränkt werden. Zum Beispiel indem nur Muster angezeigt werden, die durch ein bestimmtes Framework unterstützt werden oder für eine bestimmte Domäne spezifisch sind.

Dies soll eine Auswahl an Ansatzpunkten für weitere Arbeiten sein. Die Verbindung der vorgestellten Modellierungsform einer Mustersprache mit der modellgetriebenen Softwareentwicklung könnte eine gute Grundlage für den Entwurf von Sicherheitsarchitekturen und deren automatisierten Implementierung bieten und so zukünftige Software sicherer machen.

Anhänge

A Veröffentlichung von Aleksander Dikanski

Die Veröffentlichung wurde von Aleksander Dikanski aus den Ergebnissen der Überprüfung des Spring Security Frameworks auf Muster geschrieben. Der Artikel wurde für die SECURWARE 2012 in Rom eingereicht. Die Unterstützung bei der Erstellung der Kapitel II, IV und V war Teil dieser Arbeit, weswegen die Veröffentlichung angehängt wird.

Identification and Implementation of Authentication and Authorization Patterns in the Spring Security Framework

Aleksander Dikanski[1], Roland Steinegger[2], Sebastian Abeck[1]
Research Group Cooperation & Management (C&M)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
[1]{ a.dikanski, abeck }@kit.edu, [2]roland.steinegger@student.kit.edu

Abstract— **In the development of secure applications, security patterns are useful in the design of security functionality. Mature security products or frameworks are usually employed to implement such functionality. Yet, without deeper comprehension of these products, the implementation of security patterns is difficult, as a non-guided implementation leads to non-deterministic results. In this paper, the Spring Security framework is analyzed with the goal of identifying supported security patterns for authentication and authorization. Additionally, a best practice guide on implementing the identified patterns is presented. A real world case study is presented, in which the findings are employed to implement pattern-based security functionality in a web application.**

Keywords- security patterns, security framework, security engineering, authorization, authentication

I. INTRODUCTION

Security engineering aims for a consecutive secure software development by introducing methods, tools, and activities into a software development process [1]. As such, each phase of the software development needs to consider security aspects: in the analysis phase security requirements are identified as constraints to functional requirements, in the design phase security functionality is modeled in conjunction with the main business functionality and finally security solutions are realized in the implementation phase.

Security patterns are an agreed upon method to describe best practice solutions for common security problems [2]. When designing security functionality for an application such patterns can be instantiated in the design model to cover a certain security requirement.

The reuse of existing security functionality, i.e., in the form of security components, frameworks or products, is considered best practice as well. For one, the maturity of existing and well-maintained security functionality can usually not be achieved by implementing it completely new. Thus the quality of the security functionality of the developed application is increased. Also, as the main focus of software development lies upon the implementation of the business functionality, the reuse of existing functionality increases the efficiency of the implementation process.

Implementing security patterns using existing security functionality is complicated. For one, their built-in flexibility to support many different application contexts leads to a high complexity so that often a deep understanding of the internal workings is required. This often raises the question, if and how the required security patterns can be implemented with the selected security product. In such a case, the security functionality needs to be analyzed by security experts to determine the supported patterns.

Such an analysis is especially usefully if a model-driven approach is used in the security engineering process to automatically generate security-related artifacts from design models. The identified and supported patterns of the framework or product can be used to describe the target platform and to generate framework artifacts from design models. Such an approach is part of a reuse-based security engineering approach, which we described in earlier publications [3].

In this paper capabilities of the popular open source authentication and authorization framework Spring Security [4] are examined. The goal thereby is to identify common patterns supported by Spring Security and provide a reusable catalog of best practice advice on how to implement them in a high quality fashion. Theses informal description can be used by developers in the need to evaluate security frameworks as well as a guide to implementation. Also the can be used to describe more formal transformation rules for a model-driven approach.

The rest of this paper is structured as followed: Section 2 introduces the Spring Security framework and discusses related work. In Section 3, the relationship of the pattern-based framework description to our reuse-based security engineering approach is described. The identified security patterns and their equivalent implementation using Spring Security are covered in Section 4. In Section 5, a real-world case study is presented, which shows the security pattern implementation using Spring Security. A conclusion and outlook on future work closes the body of this paper.

II. BACKGROUND AND RELATED WORK

A. Background on Spring Security

Spring Security is an open source Java framework, providing highly flexible and adaptable authentication, authorization, and access control solutions. While it is currently developed by SpringSource, a division of VMware,

it was initially developed under the name of The Acegi Security System for Spring (Acegi Security).

The framework is built in a modular way with loosely coupled components, which are connected using dependency injection. The core classes of Spring Security and their dependencies are shown in Figure 1. Information of a user is held by an *authentication* class, which is held by a *security context* class for every authenticated user in an application. This data is loaded by the *authentication manager*, which verifies the authenticity of users using offered credentials and information from a user store.

Classes implementing functionality for intercepting calls to secured objects extend the *abstract security interceptor*, which is the central class in terms of authorization. The intercepting class gets information of the current user through the *security context* and information of the secured object is retrieved through the *security metadata source*. The latter abstracts from the type of secured object and raises variability. Access decisions are taken by the *access decision manager*, which is also called by the *abstract security interceptor*. The *access decision manager* calls voters, which decide whether access is granted or not and which can be added dynamically to the application. Thus the *access decision manager* with its voters abstracts from the access control mechanism.

Spring Security can be used for desktop applications, but the authentication interceptor has to be written by hand. The main purpose of Spring Security is to secure applications that use the Java Enterprise Edition, thus mainly web-based applications. The framework integrates with many authentication technologies and standards, e.g., Lightweight Directory Access Protocol (LDAP), Central Authentication System (CAS), OpenID and OAuth. Spring Security also provides support for basic role-based access control. Due to its flexible architecture the framework can easily be adapted to support other forms of authentication and authorization and access control.

B. Related Work

Due to the identification of security patterns, the work is based on common security pattern literature. Patterns that do not base on a specific programming language were selected from [2]. The level of abstraction varies between the introduced patterns in [2]. Some patterns are specific for operating systems or similar concrete areas and others describe the common mechanisms in security and can widely be used. Further patterns were gained from [5]. Its aim is to describe security patterns in the area of java web applications. Most patterns are based on java specifications or common web technology. Security patterns describing access for subjects are taken from [6]. Because a policy to describe authorization based on attributes is introduced in [6], the article mentions common patterns in this area. The works show example implementations of the security patterns without covering the usage of frameworks or frameworks supporting the pattern.

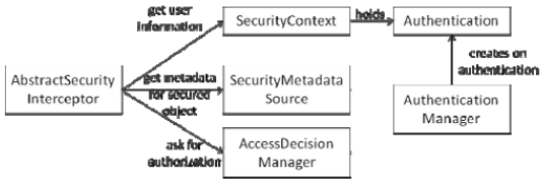

Figure 1. The main classes of the Spring Security-Framework.

Besides the need of pattern knowledge, information on the Spring Security framework, its inner relations and concepts is needed. This knowledge on the framework and its usage can be found in [7], where concepts of the Spring Security are described and several examples and best practices on solving security issues are shown. The examples are based on the version 3.0 and thus features like using permission for authorization [8] are not included. Security patterns supported by Spring Security are not introduced in [7], although the implementation of patterns, e.g., *Identity-Based Access Control*, is described. Another source for key concepts and best practices in using Spring Security 3.1 is [8]. Changes from version 3.0 to 3.1 and new concepts in solving security problems with Spring Security are mentioned. Thus [8] enhances [7] with new features and practical experiences.

III. REUSE-BASED SECURITY ENGINEERING

The pattern-based identification and description of security functionality in existing frameworks is part of a reuse-oriented security engineering approach, we presented in earlier works [3]. We argue for reuse of existing security functionality as well as knowledge throughout the phases of development process to increase the quality of the implemented software artifacts as well as to increase development efficiency.

For one, the reuse of knowledge about possible threats and attacks against information resources, as well as appropriate countermeasures, is feasible in the analysis of security requirements of an application.

The topic discussed in this paper covers the design and implementation phase of the engineering process. In the design phase existing security knowledge should be used to determine possible solutions for security problems. Security patterns offer a proven method for describing such best practice solutions and can be integrated with common design patterns [2]. The implementation of security solutions should be based on existing security functionality, such as provided by products, frameworks or components, as these are more mature and field tested, than a new implementation. Also, they usually offer support for existing security standards and technologies.

Yet to support the security engineering process, there is a need for knowledge of the frameworks used for securing the software product. During the design phase knowledge about patterns that are supported by a framework is needed, in order to avoid incompatibilities between design and implementation. When implementing the design it is beneficial to know how to implement a pattern with a framework and this knowledge is either needed for a model-

driven approach. This leads to the need of pattern identification in security frameworks.

IV. Authentication and Authorization Pattern Identification

The following section describes the process of how we identified patterns in the Spring Security framework as well as the reasons that led to the patterns to search for. The main part will be about our results in identifying patterns in the framework and how these patterns can be implemented; the patterns are separated in first authentication and second authorization patterns. Thereby a distinction is made between patterns describing authentication as well as authorization policies and architectural patterns, describing components using and evaluating the policies.

The patterns were identified manually by using knowledge on implementing applications with Spring Security and reading the source code of the framework. Another source of information is the reference documentation and a book about using the Spring Security framework [7].

Choosing the patterns to identify in the framework, we tried to cover several areas within authentication and authorization; among these are patterns on specification of policies, on enforcing and combining authentication and authorization. Another reason in favor for the selection is their publicity; we tried to choose common known patterns and therefore selected them from [2] and [5].

C. Authentication Patterns Description

The patterns described in this chapter are supporting decisions in the software development process concerning authentication.

1) Authentication Policy Patterns

In [5] several mechanisms to authenticate a subject are specified. The mechanisms are based on what the subject knows, owns or is. We abstract these into an *Authentication Information* pattern, which defines, that the subject has to deliver some sort of information to authenticate itself. Further specification of the type of information is needed. According to the three types mentioned above, a concretization of this pattern is that the subject has to deliver knowledge, e.g., a username and password. Further concretizations are handing in information the subject owns, e.g., from a smartcard, or the subject commits information it has, e.g., finger prints. Lastly the fourth specialization of the *Authentication Information* pattern is the combination of any two or more of these three concretizations.

2) Authentication Architectural Patterns

Authentication Information needs to be stored for comparison with user input. This gap is closed by the *User Store* pattern [5]. It defines, that user information is stored in some kind of repository. Depending on the type of authentication mechanism different concretization of the *User Store* are required. Thus, this pattern only defines, that there has to be a user store offering information, but abstracts from the concrete implementation. A concrete *User Store* can be for example a LDAP directory or a database, containing usernames and passwords, or a file system, storing certificates.

Enforcing the authentication needs specification of the required components in the software architecture and their interplay. The *Authentication Enforcer* pattern [5] describes these components and their interaction in a web-based application. The pattern abstracts from the authentication mechanism used to enhance reuse. Another aim of the pattern is to centralize authentication functionality and therefore to reduce redundancy and errors being spread over the application. The main component is the eponymous *Authentication Enforcer*. Authentication requests of the client are sent to this component. The *Authentication Enforcer* takes the information offered by the clients from the request context and compares it to data from the user store. On successful verification, the *Authentication Enforcer* creates a subject containing information gained from the user store on the subject.

B. Authentication Patterns Identification

This section shows how the previously introduced patterns can be implemented using Spring Security. Some code and configuration snippets as well as the relation of the classes offered by the Spring Security framework are presented.

The main interface for the *Authentication Information* pattern is the Spring Security *Authentication* interface, as its implementation offers information depending on the authentication mechanism. The *Authentication* interface is closely coupled to the *Authentication Provider* that loads the user information.

This leads to the *User Store* pattern. Accessing storages with the Spring Security framework is achieved through different implementations of the *Authentication Provider* interface. Each implementation represents a different *User Store* and uses varying *Authentication* concretizations, e.g., the *OpenID Authentication Provider* offers OpenID authentication by creating an *OpenID Authentication Token* that implements the *Authentication* interface. The *Authentication Manager* uses the *Authentication Provider* to verify authenticity of users. An *Authentication Manager* and its *Authentication Providers* can be configured using XML. An example configuration is shown in Figure 2. The default authentication manager is used and the custom authentication provider class can be inserted.

In Spring Security, the *Authentication Enforcer* is implemented using the filter chain mechanism introduced by the Java Servlet Specification [9]. There is a filter for rendering a login page, for processing the user input and verifying authenticity as well as for persisting the security session of the client. The *Default Login Page Generating Filt*er is executed if the login URL of the application is called and renders a login page to the client. When the client sends the rendered login form, the *Username Password Authentication Filter* tries to authenticate the client using the configured *Authentication Manager*. Another example is the *Basic Authentication Filter*, which gets the username and password from the request according to RFC 1945 [10] and verifies authenticity. There are also filters for CAS or

```
<authentication-manager>
  <authentication-provider ref=”customAuthProvider/>
</authentication-manager>
```

Figure 2. Defining an authentication manager in Spring Security.

```
<bean id="filterChainProxy" class="... .FilterChainProxy">
 <sec:filter-chain-map ...>
  <sec:filter-chain pattern="/application/**" filters="
      securityContextPersistenceFilter,
      basicAuthenticationFilter, ..." />
 </sec:filter-chain-map>
</bean>
```

Figure 3. Defining the filter chain in Spring Security.

OpenID authentication, because they depend on an external user store and therefore must be treated differently.

Due to the statelessness of HTTP, the *Security Context Persistence Filter* is needed, which saves the security context including the authentication in the HTTP session before responding to a request and recovers the security context at the beginning of the next request. Writing an own filter for supporting, e.g., biometric authentication is possible, too. How the filter chain can be defined is shown in Figure 3. In the example the filter chain proxy is configured, which calls the several filters for the specified path. For each filter specified in the filter chain there must be a Java class with the same name. The filter chain and authentication provider offers flexibility in adding new authentication mechanisms and user stores needed to support the *Authentication Enforcer* pattern.

C. Authorization Patterns Description

This section introduces patterns that can be used to describe or enforce authorization. Because there is a close relationship between authentication and authorization, some architectural patterns require authentication or even offer it.

1) Authorization Policy Patterns

There has to be a way to define access control, i.e., who is allowed to do a specific action in a system. The Authorization pattern [2] addresses this problem. It is used to define access control for resources at a high level of abstraction. A subject is assigned a right for a resource. High level of abstraction means that subject, right and resource are not specified concretely. The protected resource can be of any type, e.g., a physical desk in an office or an entry of a person in a database. The same applies for the subject, in that it is closely coupled to the resource in its abstraction. The subject can be a real person, a server accessing a web resource. At last the right of a subject is not specifically described. The right can be evaluated by a complex expression or it is directly specified like Person P is allowed to access Resource R in any way.

Role-Based Access Control (*RBAC*), described as a pattern in [2], is a specialization of the *Authorization* pattern, which separates the *subject* of the authorization pattern into two parts. Instead of directly setting rights for a subject, a *subject* gets assigned a *role*, which has a *right* to access a *resource*. Thus, it is possible to combine subjects with the same rights among a system reducing the complexity of rights management.

Another concretization of the *Authorization* pattern is *Attribute-Based Access Control* (*ABAC*, [6]). In contrast to RBAC, in ABAC the calculation of the *right* is described in more detail. It defines, that a right can be defined through expressions on *attributes* of the *subject*, *resource* or *environment*. Therefore the *right* must be calculated at runtime and gets a dynamic facet.

The last policy and specialization of the *Authorization* pattern examined, is the direct interpretation of the Authorization structure, called *Identity-Based Access Control* (*IBAC,* [2]). Due to the structure, the concrete *subject* gets directly a *right* to access a *resource* in a specific way. Thus a fine-grained definition of access control is established.

2) Authorization Architectural Patterns

Besides defining authorization policies, there are patterns describing their enforcement, i.e., access control. An abstract example for enforcement of access control is the *Policy Enforcement Point* (*PEP*) pattern, also known as *Reference Monitor* [2], [11]. The *PEP* defines components and flows needed to support controlling access to a resource in an abstract way. Requests to a *protected resource* shall be intercepted by the *PEP*. According to *authorization rules*, which consist of *authorization* items, access is granted or denied. Thus the pattern covers situations when a guard secures a server room verifying the authorization of the people as well as a protection proxy.

Another concretization of the PEP is the *Authorization Enforcer* pattern [5]. The purpose of the pattern is to control access in a Java EE application. Due to this circumstance, there are several variations of the pattern using different Java specifications. Requests from the *client* are intercepted and redirected to the *Authorization Enforcer*, which uses the *authentication provider* to set the *permissions* to the already loaded *subject*. Thus the pattern needs an authenticated *subject*, e.g., set by the *Authentication Enforcer* pattern. With the *permissions* of the *subject* the *authorization enforcer* decides whether the access is granted or rejected.

The *Intercepting Web Agent (IWA)* pattern [5] helps in separating application logic from authorization and authentication logic. It can also be used to add access control and authentication after the development of an application. The name already suggests that the patterns operational area is web application development. *Client* requests are intercepted by the eponymous *IWA*. Either the *client* authenticates itself and its authentication information is persisted through a cookie or the *client* tries to access a *resource*. If accessing a *resource*, the *IWA* loads the previously persisted information of the *subject*. The request is forwarded by the *IWA*, if the *subject* is authorized.

D. Implementation of Authorization Patterns

The following section, the implementation of the authorization patterns with the Spring Security framework is shown.

1) Policy Pattern

First the *Authorization* pattern is examined. Due to the voter mechanism used for access decisions, the framework can be enhanced to support several access control patterns. The sections about ABAC, RBAC, and IBAC show different voters supported by the framework and indicate the flexibility. By implementing an *access decision voter* it is possible to access external frameworks or software and to gain extra information needed for the decision or to ask for the decision from external software.

RBAC raises the need for defining *roles* of *users*. In Spring Security *roles* are called (*granted) authorities* [7]. *Authorities* can be assigned to *users*, e.g., via configuration or the *authorities* are loaded from the *User Store* [7]. A documented best practice is the arrangement of *authorities* into hierarchies [8]. *Roles* are assigned to *users* and *rights* are assigned to *roles*. Thus a hierarchy is built and *users* are assigned several *rights* through their *role*.

Next step is to give a *role* a *right* to access a *resource*. Spring Security supports the protection of methods [7] and URLs [7]. In the configuration or annotation the corresponding *right* is used, e.g., the method to modify *resource* A is annotated with "@RolesAllowed-('RIGHT_MODIFY_A')". Thus only *users* with a *role,* having the *right* to modify resource A, are allowed to access the method. When implementing a web application based on the REST (Representational State Transfer) paradigm introduced by [12], the approach of protecting URLs is preferred. Otherwise method security and the use of annotations according to the Java Specification Request (JSR) 250 should be used. Thus the flexibility in changing the security framework is saved.

ABAC is not directly supported by Spring Security, but can be easily enhanced as shown next. Spring Security offers the Spring Expression Language (SpEL, [7]) to describe access control. Instead of annotating a right to methods or to a URL, expressions can be used. When evaluating to True, access is granted. In SpEL expressions, *attributes* of the *subject* or the *resource* can be used and compared, e.g., "authentication.id=#resource.ownerId", which evaluates to True, if the users owns the resource.

These expressions can be combined with "and" and "or". In general, the SpEL fulfills the requirements of the application. When using more complex ABAC expressions, SpEL in combination with *permission evaluators* can be used. The expression "hasPermission" can be used [7]. The *access decision voter* processing expressions calls *permission evaluators* for each expression. Implementing a *permission evaluator* closes the gap between the needs of ABAC and the Spring Security access control implementation. The implementation of the *permission evaluator* interface can access any *attribute* of the *subject*, *resource* and *environment*.

Last authorization policy pattern to discuss is IBAC. Spring Security offers the use of *Access Control Lists*, which are a common approach to implement IBAC [2]. In [7], the set up of a database holding the *access control lists* and the configuration of Spring Security to use the database is shown. For each *resource* an *access control entry* can be added to the database, giving specific *permissions* to a *subject*. Built-in permissions are read, write, create, delete and administer. These permissions can be enhanced or replaced [7]. Besides *access control list* and its *entries* the protected URLs have to be configured or methods have to be annotated. This is made with the already known "hasPermission" expression of SpEL [7].

2) Architectural Pattern

The previous section showed the definition of authorization policies with Spring Security and merely parts of their enforcement. The framework uses a concrete PEP each for URLs and for method access control. The PEP has to handle all requests on a *protected object*. A filter (*filter security interceptor*) is used to intercept requests on URLs and to control access on the URL. The filter implements the *abstract security interceptor*. Thus requests on URLs are handled as described in Section II.A.

Requests on methods are intercepted using the Spring Aspect-Oriented Programming (AOP) [13] feature. The Spring *annotation security aspect* enhances security annotated methods. The advice of the aspect redirects method calls to the *aspect method security interceptor*. This *interceptor* is an implementation of the *abstract security interceptor* interface, too.

Thus requests to URLs and methods are intercepted by the Spring Security framework and processed to enforce access control. The *authorization rules* are described by the *authorization policy* that is used. Method annotation and expressions in configuration for URLs describe the concrete *authorization* for a *resource*. The PEP pattern is used with Spring Security, if the *authorization* pattern is set up and the *filter chain* is configured or method security is activated [7].

The *Authorization Enforcer* pattern is the concretization of the PEP for Java EE applications. Thus, the mentioned protection of methods and URLs is an implementation of the pattern. The Spring Security *authentication manager* takes the role of the *authentication provider* and the several authentication filters as well as the *authorization manager* represent the *authorization enforcer* role. Thus the *Authorization Enforcer* pattern can be implemented by using Spring Security access control.

The *Intercepting Web Agent* pattern cannot be applied to the method protection because the pattern defines application execution after access control. Thus the implementation of the pattern is applied through configuration of the *authentication enforcer* pattern, the *authorization* pattern and a configured URL protection.

V. Implementing Case Study

The knowledge described in the previous sections in combination with, e.g., use cases, misuse cases and component diagrams has been applied to the software development in a scientific environment. Security engineering was part of the whole development process. Spring Security was used to protect the application using the security pattern knowledge.

A. Scenario Description

The application is part of the KITCampusGuide, which is a navigation tool supporting students, teachers and staff in

finding and navigating to points of interest (POI). A POI can be any kind of landmark, like a canteen, an auditorium or offices. Due to restricted areas on the campus and several other requirements the access on POIs has to be controlled. Because users should create private POIs that can only be seen and modified by themselves, ABAC is used for access control definition.

B. *Development of the POI Manager*

Management of POIs is the most relevant to security. Thus a POI Management component was developed using Spring Security. The requirements were modeled with UML use cases. Misuse cases were used to enhance the use cases and to find security issues [14]. When designing the architecture of the application knowledge of supported patterns by the target framework could be used. Thus the architecture was not enhanced with Spring Security components, but the designed architecture used components mentioned by the supported security patterns. During the implementation phase the ABAC expressions were implemented with concretizations of the permission evaluator interface. Spring Security also supported testing the access control configured for the application.

C. *Problems and Experiences*

Finding the level of abstraction needed for the application is an important issue during design phase. In the case study the whole development process was traversed by a single person and the application size was manageable. But as the size of the application grows, this could lead to problems. A hierarchy of patterns indicated in the previous chapters would close the gap between a high level of abstraction and a level close to implementation. This could help in concretizing the design step by step.

VI. Conclusion and Outlook

In this paper the open source security framework Spring Security was examined in its support for common security patterns for authentication and authorization. Patterns for role-based and attribute-based access control as well as for username/password-based authentication were identified and appropriate best-practice implementation templates for Spring Security were provided. These templates can be used as a reference to implement the mentioned patterns in different projects. Additionally, the benefits of a pattern-based security framework description for a model-driven approach were discussed and its role in a reuse-based security engineering process was briefly explained.

In continuation of this work, the possible security design and implementation decisions need to be captured in flexible variation models to provide a decision support. Also the relationships between the patterns will be determined and specified to identify mandatory or optional dependencies between the design and implementation patterns. In future research, we focus on completing the different parts of our reuse-based security engineering process.

References

[1] R. J. Anderson, Security Engineering, 2nd ed. Indianapolis, Ind.: Wiley, 2008, p. 1040.

[2] M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, Security Patterns. Chichester, England: John Wiley & Sons Ltd, 2005, p. 565.

[3] A. Dikanski and S. Abeck, “Towards a Reuse-oriented Security Engineering for Web-based Applications and Services,” to be published in the Proceedings of The Seventh International Conference on Internet and Web Applications and Services (ICIW 2012).

[4] “Spring Security.” SpringSource Community, p. Apache License, Apr.-2008.

[5] C. Steel, R. Nagappan, and R. Lai, Core Security Patterns, 1st ed. Upper Saddle River, N. J.: Prentice Hall International, 2005, p. 1088.

[6] E. Yuan, J. Tong, B. Inc, and V. McLean, “Attributed Based Access Control (ABAC) for Web Services,” 2005 IEEE International Conference on Web Services, 2005.

[7] P. Mularien, Spring Security 3. Birmingham: Packt Publishing, 2010, p. 420.

[8] M. Wiesner, “Introduction to Spring Security 3 /3.1,” SpringOne 2GX. Chicago, Oct.-2010.

[9] Sun Microsystems, “Java Servlet 2.3 Specifications,” Palo Alto, 53, Sep. 2001.

[10] T. Berners-Lee, R. Fielding, U. C. Irvine, and H. Frystyk, “Hypertext Transfer Protocol (HTTP 1.0),” 1945, May 1996.

[11] T. E. Fægri and S. O. Hallenstein, “A Software Product Line Reference Architecture for Security,” in Software Product Lines, no. 8, Berlin, Heidelberg: Springer, 2006, pp. 276–326.

[12] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Irvine: University of California, Irvine, 2010.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in Lecture Notes in Computer Science, vol. 1241, no. 10, M. Akşit and S. Matsuoka, Eds. Berlin/Heidelberg: Springer-Verlag, 1997, pp. 220–242.

[14] G. Sindre and A. L. Opdahl, “Eliciting Security Requirements with Misuse Cases,” Requirements Engineering, vol. 10, no. 1, pp. 34–44, 2005.

B Abkürzungen und Glossar

Abkürzung oder Begriff	Langbezeichnung und/oder Begriffserklärung
C&M	Cooperation & Management Name der an der Universität Karlsruhe (TH) angesiedelten Forschungsgruppe.
Framework	Ein Framework ist selbst noch kein fertiges Programm, sondern stellt den Rahmen, innerhalb dessen ein Entwickler eine Anwendung erstellt, zur Verfügung, wobei u. a. durch die in dem Framework verwendeten Entwurfsmuster auch die Struktur der Anwendung beeinflusst wird.
Arbeitsplatz	Mit Arbeitsplatz ist im Rahmen dieser Arbeit ein Ort gemeint, an dem eine Person Lernen oder Schreiben oder in anderer Form Arbeiten kann, beispielsweise ein bestuhlter Tisch in einer Bibliothek oder ein Gruppenarbeitsraum. Der Begriff wird im Folgenden nicht als Arbeitsplatz, der von einem Unternehmen für zukünftige Mitarbeiter angeboten wird, verstanden.
POI	Point of Interest Ein besonderer Ort, beispielsweise ein Restaurant oder ein Platz in einer Stadt. Im Rahmen des KITCampusGuide, ein Ort zu dem im navigiert werden kann.
KITCampusGuide	Software zur Navigation auf dem Campus des Karlsruher Institut für Technologie.
Authentifizieren	Das Bestätigen der Identität eines Subjekts durch einen Berechtigungsnachweise (engl. Credential).
Credential	Ein Berechtigungsnachweis, beispielsweise im Internet oft ein Passwort oder eine Smartcard.
IoT	Internet der Dinge (engl. Internet of Things). Zusammenfassender Begriff für die Vernetzung verschiedener elektronischer Objekte in einem großen Netzwerk..
IOSB	Institut für Optronik, Systemtechnik und Bildauswertung
PEP	Policy Enforcement Point Sicherheitsentwurfsmuster zur Durchsetzung von Richtlinien, sogenannter Policies.
Dependency Injection	Konzept in der Softwaretechnik, um eine lose Kopplung zwischen Komponenten zu erreichen.
RBAC	Rollenbasierte Zugriffskontrolle (engl. Role-Based Access Control) Zugriffskontrollmodell, bei dem Benutzer über ihre Rolle Zugriffsberechtigungen auf Ressourcen erhält.
ABAC	Attributbasierte Zugriffskontrolle (engl. Attribute-Based Access Con-

Abkürzung oder Begriff	Langbezeichnung und/oder Begriffserklärung
	trol)
	Zugriffskontrollmodell, bei dem Ausdrücke über Attribute von Benutzern, geschützten Objekten und Umwelt zur Beschreibung der Zugriffsrechte eines Benutzers eingesetzt werden.
JSR	Java Specification Request
	Anfrage zur Standardisierung im Java Umfeld.
ACL	Zugriffskontrolllisten (engl. Access Control List)
	Liste von Subjekten und ihren Rechten für eine bestimmte Ressource oder Liste von Ressourcen und Berechtigungen eines Benutzers.
http	Hypertext Transfer Protocol
	Grundlegendes Übertragungsprotokoll im World Wide Web. Angesiedelt auf Anwendungsschicht nach dem ISO/OSI-Schichtenmodell.
RFC	Request for Comments
	RFCs sind technische und/oder organisatorische Internetstandards.
AOP	Aspektorientierte Programmierung
	Konzept der Softwaretechnik, um effizient und übersichtlich Querschnittsfunktionalität implementieren zu können.
JAAS	Java Authentication and Authorization Service
	Nicht mehr unterstützter Java Standard, der die Implementierung von Authentifizierung und Autorisierung in Java beschreibt.
JCP	Java Community Process
	Der JCP beschreibt den Lebenszyklus von Java Standards (JSRs).
Java EE	Java Enterprise Edition
	Eine Sammlung von Java Standards zur Entwicklung von Unternehmensanwendungen.
XML	eXtensible Markup Language
	Markup Sprache zur Darstellung von Baumstrukturen in Textdateien.
Separation of Concern	Prinzip der Softwaretechnik, nach dem Elemente, wie Komponenten oder Klassen, gemäß einem bestimmten Konzept getrennt werden soll. Zum Beispiel die Einteilung von Klassen in authentifizierungs- oder autorisierungsspezifische.
API	Programmierschnittstelle (engl. Application Programming Interface)
	Beschreibt eine Schnittstelle an Hand der Datenformate und Methoden, die angeboten warden.
POJO	Plain Old Java Object

Abkürzung oder Begriff	Langbezeichnung und/oder Begriffserklärung
	Eine Java Klasse, die keine umfangreiche Funktionalität anbietet, sondern nur aus Feldern und deren Zugriffsmethoden besteht.
Subjekt, Subject	Im Bereich von Authentifizierung und Autorisierung ein Element, dass Anfragen auf Ressourcen stellen kann. Dies kann beispielsweise ein konkreter Benutzer einer Anwendung oder ein Dienst, der auf einen anderen Dienst zugreift, sein.

C Abbildungen

Abbildung 1: Zusammenhang zwischen Sicherheitsanforderung Zugriffskontrolle und Sicherheitsmustern. 5
Abbildung 2: Zusammenhang zwischen Sicherheitsmuster Rollenbasierter Zugriffskontrolle und verschiedenen Implementierungen des Musters. 5
Abbildung 3: Der Zusammenhang zwischen Spring Security und verschiedenen Implementierungsmöglichkeiten zur Rollenspeicherung. 6
Abbildung 4: Diagrammelemente des Kompositionsstrukturdiagramms. Beschreibung der Rollen und Rollenbeziehungen eines Musters (links) und der Instanz eines Musters mit den Rollen der Komponenten (rechts) 12
Abbildung 5: Grundlegende Elemente eines Sequenzdiagramms 13
Abbildung 6: Elemente und Beispiel eines Merkmaldiagramms 16
Abbildung 7: Beispiel für Dependency Injection: Der sortierbaren Listenansicht wird die konkrete Implementierung des Sortieralgorithmus injiziert 17
Abbildung 8: Übersicht über die Module des Spring Frameworks 26
Abbildung 9: Die Hauptkomponenten des Spring Security Frameworks 28
Abbildung 10: Mustersystem für Zugriffskontrolle nach [PF+04:5] 32
Abbildung 11: Beziehungen der Mustersprache für Identitätsmanagement. Quelle: [DF+07] 34
Abbildung 12: Konzeptionelles Modell der Referenzarchitektur. Quelle: [FH06:281] 36
Abbildung 13: Ausschnitt der Sicherheitsarchitektursprache aus [FH06:303] 37
Abbildung 14: Zusammenhang von Zugriffskontrollmodellen als Musterdiagramm. Quelle: [FP+08:41] 39
Abbildung 15: Kompositionsstrukturdiagramm des RBAC-Musters 44
Abbildung 16: Instanziierung des RBAC-Musters durch das Spring Framework 47
Abbildung 17: Rolle erweitert durch Composite-Entwurfsmuster 47
Abbildung 18: Administratorrolle und -recht als Erweiterung der RBAC 48
Abbildung 19: Erweiterung der RBAC durch Gruppen zur Strukturierung der Benutzer 49
Abbildung 20: Sitzungen erweitern das RBAC-Muster und ermöglichen das Einschränken der Rollen eines Benutzers zur Laufzeit 49
Abbildung 21: Struktur der attributbasierten Zugriffskontrolle 50
Abbildung 22: Instanziierung des ABAC-Musters durch Spring Security 53
Abbildung 23: Struktur der Identitätsbasierten Zugriffskontrolle-Musters 54
Abbildung 24: Instanziierung des IBAC-Entwurfsmusters durch Spring Security 55
Abbildung 25: Struktur des Intercepting Web Agent-Musters 57
Abbildung 26: Ablauf des Intercepting Web Agent-Musters 57
Abbildung 27: Instanziierung des Intercepting Web Agent-Musters durch Spring Security 60
Abbildung 28: Struktur des Authorization Enforcer-Musters 61
Abbildung 29: Ablauf des Authorization Enforcer-Musters 61
Abbildung 30: Instanziierung des Authorization Enforcer-Musters durch das Spring Framework 63
Abbildung 31: Struktur des Reference Monitor-Musters 64
Abbildung 32: Darstellung der Anfrageverarbeitung beim Reference Monitor-Muster 64
Abbildung 33: Struktur des Authentication Enforcer-Musters 66
Abbildung 34: Instanziierung des Authentication Enforcer-Musters durch das Spring Framework 67

Abbildung 35: Abstraktionsebenen dargestellt durch die Baumschichten der Merkmaldiagramme 75
Abbildung 36: Verfeinerung eines Musters als Merkmaldiagramm 75
Abbildung 37: Optionale und verpflichtende Beziehungen als Merkmaldiagramm 76
Abbildung 38: Schließt Aus- und Benötigt-Beziehung als Merkmaldiagramm 77
Abbildung 39: Zusammensetzung eines Musters aus anderen als Merkmaldiagramm 77
Abbildung 40: Oder-Beziehung als Merkmaldiagramm 77
Abbildung 41: Exklusiv-Oder-Beziehung als Merkmaldiagramm 78
Abbildung 42 Verweis auf Spezifikation der Zusammenhänge eines Musters an einer anderen Stelle 78
Abbildung 43: Verfeinerungen des Autorisierungsmusters mit Merkmaldiagrammen 80
Abbildung 44: Verfeinerungen des User Store-Musters 80
Abbildung 45: Verfeinerungen des Authentication Information-Musters 80
Abbildung 46: Verfeinerungen des Reference Monitor-Musters (links) und die des Authenticator-Musters (rechts) als Merkmaldiagramm 81
Abbildung 47: Benötigt-Beziehung zwischen Authentication Information- und User Store-Muster 82
Abbildung 48: Mustersprache für Autorisierung und Authentifizierung als Merkmaldiagramm beginnend bei der Wurzel 85
Abbildung 49: Fortsetzung der Mustersprache für Autorisierung und Authentifizierung als Merkmaldiagramm 86
Abbildung 50: Genauere Definition der Muster Benutzername und Passwortspeicher sowie Zertifikatspeicher 86
Abbildung 51: Erweiterung des Autorisierung-Musters um Muster aus [PF+04] 87
Abbildung 52: Ergänzung des Authenticator-Musters um Sicherheitsarchitektursprache aus [FH06:303] 88
Abbildung 53: Ergänzung des Autorisierungsmusters um Sicherheitsarchitektursprache aus [FH06:303] 89
Abbildung 54: Merkmaldiagramm der Musterauswahl für die Arbeitsplatzsuche nach der Auswahl der Verfeinerungen des Reference Monitor-Musters 90
Abbildung 55: Merkmaldiagramm der Musterauswahl für die Arbeitsplatzsuche nach der Auswahl der Verfeinerungen des Autorisierungsmusters 91
Abbildung 56: Architektur der Arbeitsplatzsuche nach der Auswahl der Verfeinerungen des Autorisierungsmusters 92

D Literaturanalysen

Patterns and Pattern Diagrams for Access Control

[FP+08] Eduardo B. Fernandez, Günther Pernul, Maria M. Larrondo-Petrie: ”Patterns and Pattern Diagrams for Access Control”

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) Die Autoren meinen, dass die Menge an Varianten des einfachen Zugriffskontrollmodells Entwickler verwirrt, weswegen in der Praxis nur einfach Modelle eingesetzt werden.

(I2) Ein neuer Diagrammtyp für Musterbeziehungen wird vorgestellt und an Beispielen gezeigt.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) Die Autoren verweisen auf keine ähnlichen Lösungen zur Darstellung von Musterbeziehungen. Nach eigener Recherche gibt es diese derzeit auch nicht. Einzige

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) Auf Grund der großen Anzahl an Varianten für verschiedene Muster, soll in dem Artikel nur eine Auswahl üblicher Muster dargestellt werden.

(P2) Es wird angenommen, dass die Anzahl der Varianten zur Durchsetzung der Zugriffskontrolle viel geringer ist als die Zahl der Zugriffskontrollmodelle.

Lösungen

Was sind die eigenen Lösungen?

(L1) Zur Lösung wird das Musterdiagramm eingeführt, mit dem Musterzusammenhänge modelliert und Musterhierarchien dargestellt werden können.

(L2) Die Benutzung der Diagramme durch einen Entwickler wird auf an einem einfachen Beispiel gezeigt (ein Absatz).

(L3) In dem Artikel werden zudem zwei Beispiele zur Erzeugung neuer Muster vorgestellt.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) Es wird kein Nachweis gegeben.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) Die genaue Semantik hinter den Diagrammen wird nicht erklärt und es wird auch nicht geklärt, ob es diese gibt. Die Beschriftung der Beziehungen sieht recht willkürlich gewählt aus.

Somit ist die Frage ungelöst, ob die Beschriftung frei gewählt werden kann oder einer bestimmten Semantik gefolgt werden muss?

(O2) Die Diagramme ähneln der Darstellung aus [FP+08] ohne Beschriftung der Kanten. Das dortige Diagramm wurde nicht benannt. Die Frage, ob dort bereits ein Musterdiagramm eingesetzt wurde stellt sich. Falls dem so ist, gelten die Nachteile des dortigen Diagrammtyps. (Unübersichtlichkeit, fehlende Semantik)

(O3) Die Benutzung des Diagramms wird nur an einem kurzen Beispiel gezeigt, weswegen die Frage aufkommt, ob die Diagramme einem Entwickler ausreichend gegenüber der Situation ohne Diagramme helfen?

A Software Product Line Reference Architecture for Security

[FH06] T E Fægri und S Hallsteinsen: A Software Product Line Reference Architecture for Security

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) Die Frage, ob Wissen über Sicherheitsaspekte einer Architektur in einer Referenzarchitektur zusammengefasst werden kann, wird beantwortet.

(I2) Auf der ersten Frage aufbauen wird überprüft, ob eine solche Referenzarchitektur beim Design einer Sicherheitsarchitektur für Software Produktlinien nützlich ist.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) Die Autoren haben die Speicherung von Wissen unter anderem über Muster als ein Mittel zur effektiveren Entwicklung von Software identifiziert und sie wollen dies mit einer Referenzarchitektur bereitstellen.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) Beim Entscheidungsmodell ist nicht angestrebt, dass alle Variationen zur Lösung eines Szenarios dargestellt werden. Im Vordergrund steht das Zusammentragen von Informationen in diesem Bereich.

Lösungen

Was sind die eigenen Lösungen?

(L1) In der Arbeit wird eine Referenzarchitektur vorgestellt, die aus einer abstrakten Architektur, die in Zusammenhang mit Sicherheitsaspekten, wie der Anfälligkeit des Systems für bestimmte Angriffe, und ausgewählten Bedrohungsszenarien gebracht wird, besteht. Die abstrakte Architektur enthält Gegenmaßnahmen für Bedrohungen, aus denen sich konkrete Lösungen aus drei Bereichen (Erkennung, Verhinderung und Erholung) ergeben.

(L2) Der Weg zur Architektur wird über ein Entscheidungsmodell aufgezeigt. In diesem werden verschiedene Bedrohungsszenarien vorgestellt und Lösungsvarianten genannt. Die Varianten geben Sicherheitsmuster an, die umgesetzt werden sollen und die Architektur formen.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) Die Verfasser arbeiteten mit drei unterschiedlichen Unternehmen zusammen und benutzten die erarbeitete Referenzarchitektur zur Identifikation der Anforderungen durch die Szenarien und zur Unterstützung im Entwurf der geeigneten System Architektur.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) Die Referenzarchitektur bietet Unterstützung auf einer sehr abstrakten Ebene und es ist erforderlich, dass die angegebenen Muster weiter verfeinert werden, um zu einer Implementierung zu gelangen. Beispielsweise wird das Muster Authentifizierung angegeben, wobei nicht auf den konkreten Authentifizierungsmechanismus eingegangen wird und auch nicht angegeben ist, wo die relevanten Daten abgespeichert werden sollen.

(O2) Ein Zusammenhang zwischen den Mustern ist nur dahingehend gegeben, dass durch eine Variation zur Lösung eines Szenarios mehrere Muster angegeben sein können. Allerdings ergibt sich daraus nicht, ob sich Muster gegenseitig ausschließen können oder ob es weitere Verbindungen gibt.

(O3) Die Variationen im Entscheidungsmodell lassen vermuten, dass dies die einzigen Möglichkeiten zur Lösung des Problems im Szenario seien, allerdings gibt es oft mehrere Lösungsmöglichkeiten.

A pattern system for access control

[PF+04] Torsten Priebe, Eduardo B. Fernandez, Jens I. Mehlau, Günther Pernul: „A pattern system for access control“

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) Es wird ein Mustersystem vorgestellt, mit dem Autorisierung und Zugriffskontrollmodelle beschrieben werden können

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) Um ein System entwerfen zu können, benötigen unerfahrene Entwickler gute Musterkollektionen

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) Muster sind wichtiger Bestandteil des Entwurfs einer Architektur.

(P2) Da die Autoren UML zur Darstellung von Musterbeziehungen benutzen, wird vorausgesetzt, dass die UML hierfür geeignet ist.

(P3) Da die Autoren ein Mustersystem statt einer Mustersprache verwenden, wird implizit angenommen, dass die Menge von Mustern nicht vollständig sein könnte.

Lösungen

Was sind die eigenen Lösungen?

(L1) Die Autoren führen Muster ein und verwenden verbreitete Muster zur Erstellung eines Mustersystems. Zudem identifizieren sie einige Muster als Grundlegend für die Zugriffskontrolle.

(L2) Die Zusammenhänge der Muster werden mit einem UML Diagramm modelliert.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) Der Artikel nennt Beispiele, in denen die Muster des Mustersystems eingesetzt werden und eingesetzt werden sollen.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) Inwieweit das Mustersystem vollständig ist und an welchen Stellen Lücken auftreten, wird durch die Autoren nicht diskutiert. Unter anderem die Frage nach der Durchsetzung der Zugriffskontrolle und ob dies Teil einer vollständigen Sprache wäre, wird nicht betrachtet.

(O2) Auf eine geeignete Darstellung der Musterzusammenhänge wird nicht eingegangen, dies ist allerdings auch kein zentraler Punkt des Artikels.

Mapping Feature Models to the Architecture

[SPR04] Periklis Sochos, Ilka Philippow, Matthias Riebisch: Mapping Feature Models to the Architecture

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) Die zentrale Frage der Arbeit ist, wie Feature Models auf die Architektur abgebildet werden können und wie eine lose Kopplung zwischen den Komponenten, die eine Eigenschaft umsetzen, erreicht werden kann.

(I2) Die Methoden FODA (Feature-Oriented Domain Analysis), FeatuRSEB (Featured Reuse-Driven Software-Engineering Business), HyperFeatuRSEB (Hyper Featured RSEB), GenVoca und FORM (Feature-Oriented Reuse Method), die zur Erstellung von Feature Models genutzt werden können, werden kurz erläutert.

(I3) Zur Beantwortung der ersten Frage, werden die Ansätze „Aspektorientiertes Programmieren“ und „GenVoca“ bewertet, da diese eine mögliche Lösung des Problems darstellen.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) Es wird bemängelt, dass kein direkter Zusammenhang von Feature Model zu Architektur besteht. Die Feature Models, die durch verschiedene Anforderungserfassungstechniken wie Feature-Oriented Domain Analysis entstehen, sind zu abstrakt und

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) Das vorgestellte Verfahren „Feature-Architecture Mapping“ erfordert eine Erweiterung des Feature Models um eine „interacts“-Beziehung.

(P2) Ein Mangel an Werkzeugen zur Unterstützung der Entwicklung mit der angegebenen Technik und Erweiterung von Feature Models wird angemerkt.

Lösungen

Was sind die eigenen Lösungen?

(L1) Feature-Architecture Mapping wird als Methode eingeführt. Es beinhaltet eine Feature Model-Transformation durch die das Ausgangsmodell logisch eingeteilt, Interaktionen zwischen den Eigenschaften eingeführt und eine Einteilung in Eigenschaften, die der Kunde des Produkts haben will, erreicht werden soll.

(L2) Das Zusammenführen der Eigenschaften wird durch einen Plug-In-Mechanismus erreicht. Durch diese wird eine ausgewählte Eigenschaft, falls es Alternativen gibt, für die konkrete Software ausgewählt.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) Es werden keine Nachweise erbracht, allerdings wird angemerkt, dass das Verfahren derzeit bei der Entwicklung von Anwendungen für Smartphones eingesetzt wird.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) Aus Sicht der Anwendung des Verfahrens ist unklar, ob und wie es umgekehrt werden kann, um beispielsweise aus einer Architektur ein Feature Model zu erzeugen.

(O2) Aus Sicht der Arbeit wären eine Formalisierung der Transformation und eine Untersuchung des Anwendungsbereichs der Technik erstrebenswert.

Design Pattern Recovery in Architectures for Supporting Product Line Development and Application

[PSR03] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch: Design Pattern Recovery in Architectures for Supporting Product Line Development and Application

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

(I1) Vorstellung und Bewertung existierender Algorithmen zur Mustersuche

(I2) Einführung eines Verfahrens zum automatisierten Suchen nach Entwurfsmustern in einer Architektur angelehnt an die menschliche Vorgehensweise

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

(D1) Bestehende Mustersuchalgorithmen erreichen keine guten Werte hinsichtlich der Anzahl korrekt gefundener Muster und Fehlklassifikationen.

(D2) Andere Algorithmen sind meist nur in der Lage einen Teil der bekannten Entwurfsmuster zu finden.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

(P1) Die Entwurfsmuster aus „Design Patterns – Elements of Reusable Object-Oriented Software“ von Gamma, Helm, Johnson und Vlissides werden als Grundlage der Suche verwendet und keine darüber hinausgehenden Muster betrachtet.

(P2) Es wird angenommen, dass das Suchen ähnlich der menschlichen Vorgehensweise am erfolgversprechendsten ist und dass das Suchen nach minimalen Schlüsselstrukturen der Muster der menschlichen Vorgehensweise am nächsten kommt.

Lösungen

Was sind die eigenen Lösungen?

(L1) Die Suche nach einer minimalen Schlüsselstruktur wurde erweitert um positive und negative Kriterien, die entweder dafür oder dagegen sprechen, dass eine Struktur ein Muster ist oder nicht.

(L2) Die Entwurfsmustern von Gamma et al. wurden untersucht und häufig auftretende Strukturen als Suchkriterien entnommen:

- Abstrakte und konkrete Klassen und Vererbungsbeziehungen
- Attribute (Sichtbarkeit, Typ, Name)
- Methoden (Sichtbarkeit, Polymorphie, Rückgabewert, Parameter, Abstraktion, Name)
- Konstruktoren (Sichtbarkeit, Name, Parameter)

- Beziehungen (Assoziation, Komposition, Aggregation, Delegation, Objektgenerierung, Methodenaufrufe, Unterschiedliche Benutzung, Vorlagennutzung)

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

(N1) Das Verfahren wurde an studentischen Projekten getestet. Dabei konnten nicht alle Muster in einbezogen werden, da das verwendete Framework einige der Suchkriterien nicht unterstützte. Angegeben wird in der Arbeit, dass das Programm die Muster Singleton und Interpreter ohne Fehlklassifikation erkannte.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben bzw. stellen sich dem Leser?

(O1) Da die Implementierung nicht vollständig war und nicht jedes Entwurfsmuster gesucht werden konnte, stellt sich die Frage, ob das Vorgehen für komplexere Muster gute Ergebnisse liefert.

(O2) Es wäre interessant zu wissen, wie die Entscheidungen für die relevanten Suchkriterien aus dem Buch von Gama et al. getroffen wurden und ob wie Kriterien für Sicherheitsentwurfsmuster gefunden werden könnten.

E Literatur

[AIS77] Christopher Alexander, Sara Ishikawa, Murray Silverstein: „A pattern language“, Oxford University Press Inc, 1977

[Ale79] Christopher Alexander: „The timeless way of building“,Oxford University Press Inc, 1979

[Bat05] Don Batory: „Feature Models, Grammars, and Propositional Formulas”, SPLC'05 Proceedings of the 9th international conference on Software Product Lines, University of Texas, Springer-Verlag, Seiten 7-20, 2005

[BC87] Kent Beck, Ward Cunningham: „Using Pattern Languages for Object-Oriented Programs“, Object-Oriented Programming, Systems, Languages & Applications, URL http://c2.com/doc/oopsla87.html (abgerufen am 5. Juni 2012), 1987

[BK+04] Günter Böckle, Peter Knauber, Klaus Pohl, Klaus Schmid: „Software Produktlinien”, dpunkt.verlag, Mai 2004

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: „Pattern-orientierte Software-Architektur – Ein Pattern-System“, Addison-Wesley, Januar 1998 (englisches Original 1996)

[BHS07] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt: „Pattern-oriented software architecture – On Patterns and Pattern Languages“, Wiley, 2007

[BSI11] Bundesamt für Sicherheit in der Informationstechnik: „BSI-Lagebericht IT-Sicherheit 2011“, URL https://www.bsi.bund.de/cln_165/ContentBSI/Publikationen/Lageberichte/bsi-lageberichte.html (abgerufen am 24. Februar 2012), 2011

[DF+07] Nelly Delessy, Eduardo B. Fernandez, Maria M. Larrondo-Petrie: „A Pattern Language for Identity Management”, Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), Florida Atlantic University, März 2009

[DZP07] Jing Dong, Yajing Zhao, Tu Peng: „Architecture and Design Pattern Discovery Techniques - A Review”, Proceedings of the 2007 International Conference on Software Engineering Research & Practice, Las Vegas Nevada, CSREA Press, Seiten 621-627, Juni 2007

[FH06] T E Fægri, S Hallsteinsen: „A Software Product Line Reference Architecture for Security”, Software Product Lines, 10th International Conference, Baltimore, Maryland, Seiten 275-324, August 2006

[FP01] Eduardo B. Fernandez, Rouyi Pan: „A pattern language for security models”, Florida Atlantic University, 2001

[Field00] Roy Thomas Fielding: „Architectural Styles and the Design of Network-Based Software Architectures“,University of California, Irvine, 2000

[Fow04] Martin Fowler: „Inversion of Control Containers and the Dependecy Injection pattern”, URL http://martinfowler.com/articles/injection.html (abgerufen am 18.06.2012)

[FP+08] Eduardo B. Fernandez, Günther Pernul, Maria M. Larrondo-Petrie: „Patterns and Pattern Diagrams for Access Control", Springer, September 2008

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, November 2004

[IETF11] Internet Engineering Task Force: „HTTP State Management Mechansism", http://tools.ietf.org/html/rfc6265 (abgerufen am 24. Februar 2012), April 2011

[IETF99] Internet Engineering Task Force: „Hypertext Transfer Protocol – HTTP/1.1", http://tools.ietf.org/html/rfc2616 (abgerufen am 11. Juni 2012), Juni 1999

[JCP02] Java Community Process: „JSR-000053 Java Servlet 2.3 and JavaServer Pages 1.2 Specifications", http://www.jcp.org/en/jsr/detail?id=53 (abgerufen am 18. Juni 2012), Juni 2002

[JCP09] Java Community Process: „JSR 311: JAX-RS: The Java API for RESTful Web Services", http://jcp.org/en/jsr/detail?id=311 (abgerufen am 18. Juni 2012), November 2009

[KL+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin: „Aspect-oriented programming", Proceedings of the European Conference on Object-Oriented Programming, Seiten 220-242, Juni 1997

[Mul10] Peter Mularien: „Spring Security 3: Secure your web applications against malicious intruders with this easy to follow practical guide", Packt Publishing, Mai 2010

[NWG96] Network Working Group: „Hypertext Transfer Protocol -- HTTP/1.0", http://www.ietf.org/rfc/rfc1945.txt (abgerufen am 24. Februar 2012), Mai 1996

[Ora11a] Oracle Technology Network: „JavaBeans FAQ: General Questions", http://www.oracle.com/technetwork/java/javase/faq-135947.html#Q1 (abgerufen am 31. Oktober 2011)

[Ora11b] Oracle Technology Network: „Java™ Authentication and Authorization Service (JAAS) Reference Guide", http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html (abgerufen am 14. März 2012)

[Ora11c] Oracle Technology Network: „Java EE 6 Technologies", http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html (abgerufen am 11.06.2012)

[PF+04] Torsten Priebe, Eduardo B. Fernandez, Jens I. Mehlau, Günther Pernul: „A pattern system for access control", PROCS OF THE 18TH. ANNUAL IFIP WG, Kluwer, Seiten 25-28, 2004

[PSR03] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch: „Design Pattern Recovery in Architectures for Supporting Product Line Development and Application", „Modelling Variability for Object-Oriented Product Lines", BoD GmbH, -Seiten 42-57, Juli 2003

[RQ+07] Chris Rupp, Stefans Queins, Barbara Zengler: „UML 2 – Glasklar", Hanser, 2007

[SH+06] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux: „Feature Diagrams: A Survey and a Formal Semantics", 14th IEEE International Conference , Seiten 139-148, September 2006

[SNL06] Christopher Steel, Ramesh Nagappan, Ray Lai: „Core Security Patterns – Best Practices and Strategies for J2EE, Web Services, and Identity Management", Prentice Hall, Oktober 2005

[SO05] G. Sindre and A. L. Opdahl: „Eliciting Security Requirements with Misuse Cases", Springer-Verlag New York, Seiten 34-44, Januar 2005

[SPR04] Periklis Sochos, Ilka Philippow, Matthias Riebisch: „Mapping Feature Models to the Architecture", Springer, Seiten 138-152, 2004

[SprS11a] Spring Source: „Spring Framework", URL http://www.springsource.org/ (abgerufen am 31. Oktober 2011)

[SprS11c] Spring Source: „Spring Security", URL http://www.springsource.org/spring-security (abgerufen am 31. Oktober 2011)

[SprS11e] Spring Source: „Spring Security Reference Documentation", http://static.springsource.org/spring-security/site/docs/3.1.x/reference/springsecurity.html (abgerufen am 13. März 2012)

[SF+06] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad: „Security pattern – Integrating Security and Systems Engineering", John Wiley & Sons, Dezember 2005

[Sun04] Sun Microsystems Laboratories: „Sun's XACML Implementation", http://sunxacml.sourceforge.net (abgerufen am 26. Februar 2012)

[Wi11] Mike Wiesner: „Introduction to Spring Security 3/3.1", URL http://www.infoq.com/presentations/Spring-Security-3 (abgerufen am 25.Januar 2011)

[YT05] Eric Yuan, Jin Tong: „Attributed Based Access Control (ABAC) for Web Services", 2005 IEEE International Conference on Web Services, Orlando, Florida, Juli 2005

[Zim95] Walter Zimmer: „Relationships between design patterns", ACM Press/Addison-Wesley Publishing Co. New York, Seiten 345-364, 1995