Advanced Patterns And Frameworks

Zusammenfassung & Notizen

Hochschule für Technik Rapperswil
Frühjahressemester 2013

Autoren Manuel Alabor (MAL)
URL https://github.com/swissmanu/hsr-apf-2013

Inhaltsverzeichnis

Kapitel 1 **Access Control Models**

1.1. Authorization

Das Authorization Pattern beschreibt auf einfache Art und Weise die Zugriffsberechtigungen eines Subjekts auf ein bestimmtes Objekt. Es spezifiziert zudem die Art des erlaubten Zugriffes (Lesend, schreibend etc.)

Kontext

Jegliche Umgebungen in denen der Zugriff auf enthaltene Objekte kontrolliert werden muss.

Problem

In einer kontrollierten Umgebung muss sichergestellt werden, dass nur berechtigte Subjekte auf entsprechende Objekte zugreifen können. Es stellt sich also die Herausforderung, diese Information losgelöst von den eigentlichen Objekte abzulegen. Dabei soll aber eine gewisse Flexibilität bei der Definition von Berechtigungen, Objekten und Subjekten erhalten bleiben.

Des weiteren sollen diese Informationen so einfach wie möglich im Nachhinein änderbar sein.

Lösung

Strukturell fällt die Lösung zum Authorization Pattern relativ simpel aus:

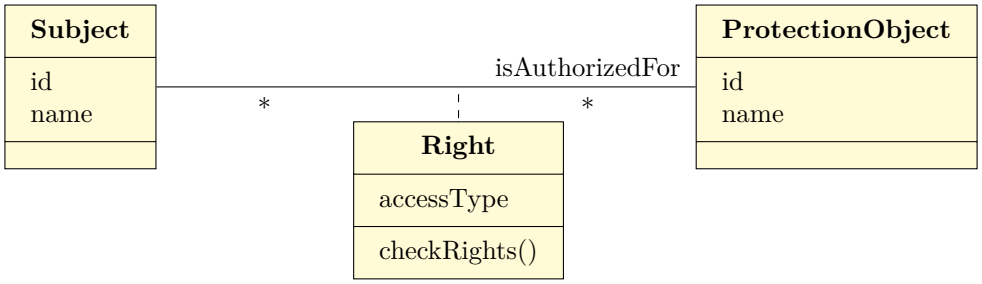


Abbildung 1.1.: Authorization Klassendiagramm

- Subject beschreibt jegliche Aspekte des zu berechtigenden Subjekts
- Das ProtectionObject ist das zu schützende Objekte
- Right enthält alle Informationen, wie Subject auf ProtectioObject zugriefen darf/-kann

Erweiterungen

Die vorgestellte Struktur kann um komplexere Aspekte erweitert werden. So kann bspw. mittels einem "Copy"-Flag eine Stellvertretung eines Subjektes durch ein anderes ermöglicht werden. Weiter ist die Verwendung eines Prädikats denkbar, welches eine Regel mit zusätzlicher "Intelligenz" austatten kann (-> "Darf nur zugreifen wenn Zeit innerhalb Arbeitszeit")

Diese Anpassungen können direkt auf dem Rights-Objekt modelliert werden.

Vor- & Nachteile

- Durch seine Offen- und Allgemeinheit kann dieses Pattern auf jegliche Umgebung appliziert werden (Filesysteme, Organistaitonsstrukturen, Zugangskontrollen etc.)
- In der beschriebenen Form sind administrative Aufgaben (Änderung der Zugriffsrechte) nicht gesondert definiert. Für bessere Sicherheit ist dies jedoch von Vorteil
- Für viele Subjekte/Objekte müssen entsprechend viele Berechtigungsregeln erfasst und auch verwaltet werden
- Viele Regeln machen die Verwaltung für einen Administrator zu einer heiklen Aufgabe (Verkettung von Berechtigungen etc.)

Beispielanwendungen

- Dateisysteme
- Firewalls greifen teilweise auf dieses Pattern zurück, um Regeln für den analysierten Traffic zu modellieren

Mögliche Prüfungsfragen

- *Macht es Sinn, auch verbietende Regeln zu erfassen?*
  Möglich wäre dies bestimmt, im Normalfall verkompliziert dies jedoch das Sicherheitskonzept auf allen Ebenen: Die Administration wird undurchsichtiger, die Überprüfung/Durchsetzung der Regeln wird komplexer und es besteht die Möglichkeit, dass sich ein Subjekt komplett “ausschliessen” kann. (vgl. Windows Filesystem)

1.2. Role Based Access Control

Diese Pattern basiert stark auf dem Authorization Pattern und versucht dessen Nachteile durch einen zusätzlichen Abstraktionslayer auszugleichen. Das ”Role Based Access Control” Pattern definiert Berechtigungen nicht direkt auf Stufe der Subjekte, sondern versucht diese in Gruppen (Aufgabenbereiche, Kaderposition, Arbeitsort etc.) einzuteilen und anschliessend auf dieser Ebene quasi übergeordnet zu berechtigen.

Kontext

Eine Umgebung mit vielen Objekten und Subjekten. Deren Berechtigungen ändern häufig. Zudem ist damit zu rechnen dass eben so oft neue Subjekte und Objekte hinzukommen oder wieder wegfallen.

Problem

Die Rechteverwaltung in dem beschriebenen Kontext generiert einen hohen administrativen Aufwand. Um die Anzahl individueller Berechtigungen zu minimieren soll versucht werden, alle Subjekte in Gruppen einzuteilen. Die Einteilung basiert darauf, dass Subjekte mit ähnlichen Aufgaben zumeist auch ähnliche oder identische Berechtigungen benötigen. Trotzdem sollen die Berechtigungen weiterhin präzise abgebildet werden können (”Need to know”).

Lösung

Organisationen bieten normalerweise bereits mehr oder weniger wohldefinierte Gruppenstrukturen (Abteilungen, Aufgabenbereiche). Ein gutes Sicherheitskonzept sollte bestrebt sein, dass jedes Subjekt genau auf die Objekte Zugriff hat, mit welchen es täglich arbeitet (wiederum ”Need to know”).

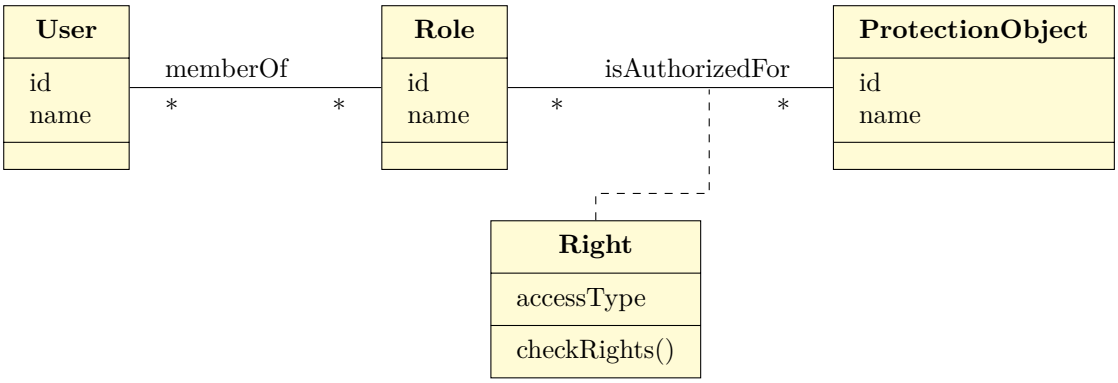


Abbildung 1.2.: Basic Role Based Access Control Klassendiagramm

Im Vergleich zum Authorization Pattern kommt lediglich ein neues Element hinzu: Die Role fasst mehrere User (Subjekte) zu einer Menge zusammen und berechtigt sie über Right für ein spezifisches ProtectionObject.

Erweiterungen

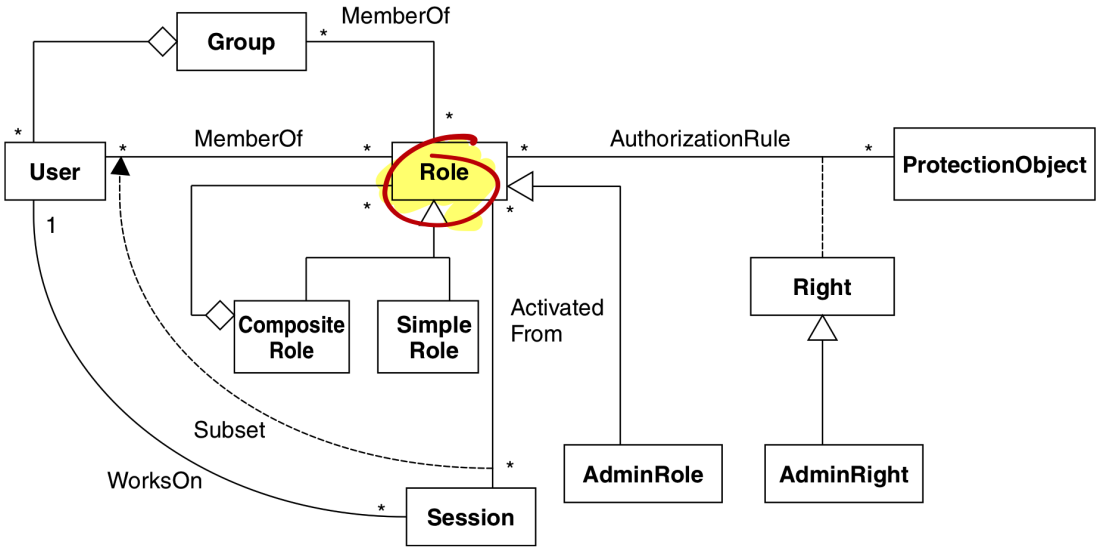


Abbildung 1.3.: RBAC mit Composite, Admins & Abstract Session

Composite Pattern

Statt einer simplen Assoziation zwischen User und Role könnte auch mit dem Composite-Pattern gearbeitet werden, um diese Abhängigkeit zu modellieren.

Administration

Wie ebenfalls bereits im Authorization-Pattern erwähnt kann auch dieses Modell zielgerichtet um Administrations-Elemente erweitert werden. Auf diese Weise kann zusätzliche Klarheit im System geschaffen werden, wer genau für was zuständig ist.

Abstract Session

Um die Möglichkeiten auf die Spitze zu treiben, sei hier auch das Abstract Session Pattern erwähnt: Die Abhängigkeit einer Session kann so direkt ins Security Modell "miteinmodelliert" werden.

Vor- & Nachteile

- Die Zusammenfassung zu Gruppen ermöglicht eine vereinfachte Administration der gesamthaft vorhandenen Berechtigungen
- Veränderungen in der realen Organistaionstruktur (Neuzugänge, Abgänge, Jobwechsel etc.) können einfacher auf das Sicherheitskonzept abgebildet werden
- Ein Subjekt kann durch mehrere Sessions verschiedene Funktionen auf einmal wahrnehmen
- Theoretisch können Gruppen wiederum in Gruppen zusammengefasst werden (Yay, even more complexity...)
- Konzeptionelle Komplexität nimmt durch die neuen Elemente wiederum zu!

Beispielanwendungen

- Windows 2000 Rights Management (Group Policies)

Mögliche Prüfungsfragen

- *Ein Subjekt hat die Rollen "Personalabteilung" und "USB Datenaustausch" zugwiesen. Wie kann verhindert werden, dass das Subjekt Personalinformationen auf einen USB-Stick speichern kann?*
  Durch die Implementierung des *Abstract Session* Patterns kann das Subjekt gezwungen werden, sich jeweils nur mit einer bestimmten Rolle am System anzumelden. So hat es jeweils entweder nur auf die Personaldatan zugriff oder kann nur Dateien mit einem USB-Stick austauschen.
  wackeliges Beispiel ;-)

1.3. Multilevel Security

Oft sollen Informationen in verschiedene Sicherheitskategorien einsortiert werden: Ein Unternehmen möchte bspw. nicht, dass der neue Praktikant auf strategisch wichtige

Informationen aus dem Verwaltungsrat-Meeting zugreifen kann. Das *Multi Level Security* Pattern beschreibt wie Informationen klassifiziert werden können.

Es definiert hierzu *Policies* welche Subjekten *Clearances* für bestimmte *Sensitivity Levels* erteilt.

Kontext

Sicherheitskritische Informationen resp. deren Verwahrung erfordert erhöhten Aufwand im Sicherheitskonzept.

Problem

Es gibt es unterschiedlich sensitive Informationen. Ein Subjekt soll entsprechend seiner Stellung innerhalb der Organistaionsstruktur Zugriff auf kritische oder weniger kritische Informationen Zugriff erhalten.

Dabei soll ein Maximum an Flexibilität für das Verändern von Parametern bestehen:

- Ein Subjekt soll so einfach wie möglich einer anderen Stufe in der Organisation zugewiesen werden könne
- Die Sensitivität einer Information muss so einfach wie möglich angepasst werden können

Lösung

Jeder Information wird ein *Sensitivity Level* zugwiesen. *Policies* definieren, welche Subjekte aus der Organistaionstruktur auf welche *Sensitivity Levels* Zugriff erhalten.

Policies werden von *Trusted Processes* erstellt und verwaltet. Sie werden gem. dem Bell-LaPadula Sicherheitsmodell[wika] umgesetzt/überprüft:

1. *No-Read-Up*
   Niedriger eingestufte Subjekte dürfen keine Informationen höher eingestufter Subjekte lesen
2. *No-Write-Down*
   Höher eingestufte Subjekte dürfen keine Informationen in Resourcen tiefer eingestufter Subjekte schreiben (Informationsweitergabe!)
3. *Zugriffsmatrix*
   Matrix, welche Zugriffberechtigungen von Subjekten auf Resourcen festlegt

Die Korrektheit der *Policies* wiederum wird über das Biba-Modell[wikb] (der Umgehrung des Bell-LaPadula Konzepts) sichergestellt:

1. *No-Read-Down*
   Höher eingestufte Subjekte dürfen keine Informationen tiefer eingestufter Subjekte lesen

2. *No-Write-Up*
   Tiefer eingestufte Stubjekte dürfen nicht in Informationen höher eingestufter Subjekte schreiben

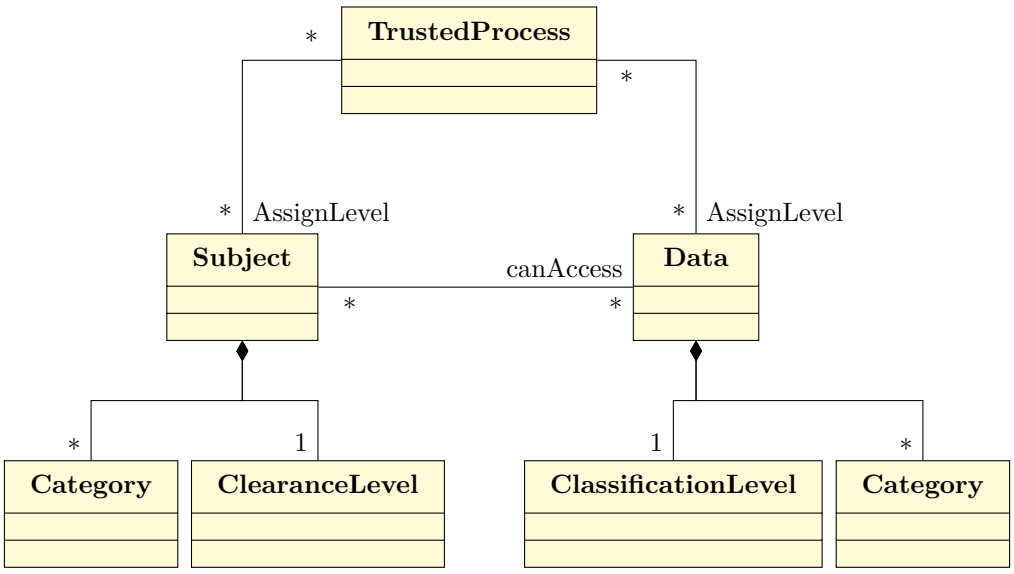


Abbildung 1.4.: Multilevel Security Klassendiagramm

Vorteile

- Welcher Benutzer welche Berechtigung erhalten soll kann relativ einfach am Organigramm einer Organisation abgeleitet werden.
- Durch die Modellierung der *Trusted Processes* trennt dieses Pattern strikt zwischen Administration und tatsächliche Umsetzung Sicherheitsregeln.

Nachteile

- Bei der Umsetzung dieses Patterns sollte darauf geachtet werden, dass normierte Bezeichnungen für die entsprechenden Sensitivity und Clearance Levels verwendet wird (–> Glossar)
- Der *Trusted Process* ist eine kiritsche Stelle im System.
  "Aber wer wird über die Wächter selbst wachen?"
- Daten als auch Benutzer müssen optimalerweise in hierarchische Berechtigungstrukturen eingeteilt werden können. Dementsprechend kann dieses Pattern nur schwer auf alltägliche Systeme übertragen werden. (vgl. Militär)

- Nur weil ein Subjekt mit einer hohen Sicherheitsklassifizierung ausgestattet wurde, muss dies nicht bedeuten, dass keine Informationen nach Aussen getragen werden. Beispiel: Banker telefoniert im Zug lautstark und gibt sensible Kundeninformationen preis.

Erweiterungen

Das Rollenkonzept von 1.2 Role Based Access Control kann mit diesem Pattern problemlos kompiniert werden: Dabei werden die *Clearance Levels* einfach auf die Gruppen statt direkt auf die Benutzer zugwiesen.

Beispielanwendungen

- Militäreisches IT-System
- Datenbanksysteme (bspw. Oracle)
- Betriebssysteme (bspw. HP Virtual Vault: HP Unix Abkömmling, properitär)

1.4. Reference Monitor

aka Policy Enforcement Point

Das *Reference Monitor* Pattern beschreibt eine abstrakte Vorghensweise, wie definierte Sicherheitsvorschriften um- und vorallem durchgesetzt werden können.

Kontext

Ein IT-System, in welchem Subjekte (Benutzer als auch technische Prozesse) auf diverse Resourcen zugriefen möchten.

Problem

Die vorangegangenen Patterns beschrieben bis anhin lediglich, *wie* Sicherheitsrichtlinien modelliert und definiert werden können. Regeln nur zu definieren kommt einem weglassen dieser gleich. Wir benötigen also eine Möglichkeit, die aufgestellten Regeln auch effektiv durchzusetzen und zu überwachen.

Beim definieren eines möglichen Mechanismus soll darauf geachtet werden, dass dieser so abstrakt wie möglich und dadurch auf verschiedenste Architekturen sowie auf alle Ebenen eines Systems appliziert werden kann.

Lösung

Folgendes Klassendiagramm zeigt den Ansatz des abstrakten *Reference Monitors*, inkl. einer konkreten Implementierung dessen. Die Collection aus *Authorization Rules* ist konkret mit einer ACL vergleichbar.

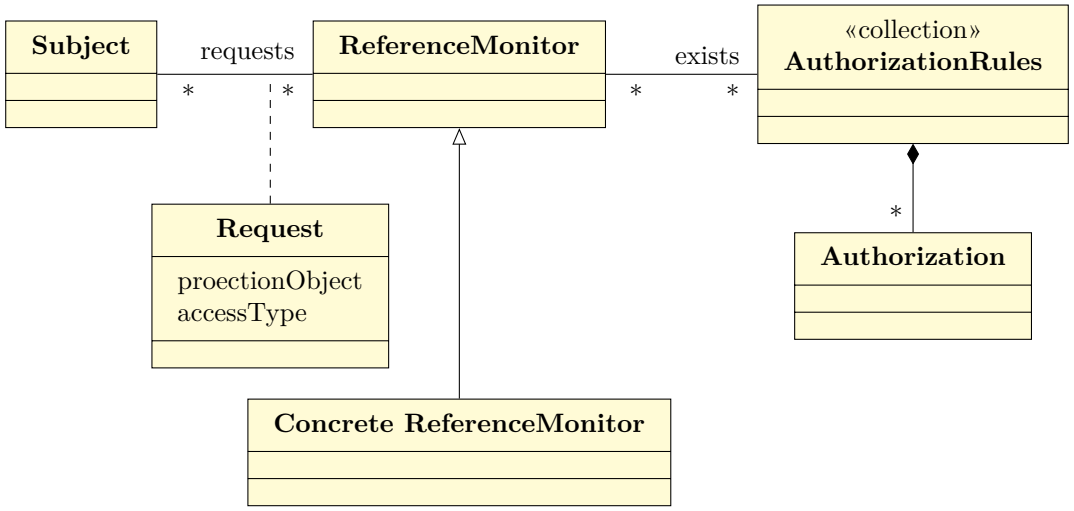


Abbildung 1.5.: Reference Monitor - Klassendiagramm

Die effektive Überprüfung, ob ein Subjekt für den Zugriff berechtigt ist, ist denkbar einfach: Jeder Zugriff auf eine Resource (ein Protection Object) wird durch den Reference Monitor geführt. Dieser prüft, ob eine entsprechende Zugriffsregel vorhanden ist und gewährt ggf. den Zugriff.

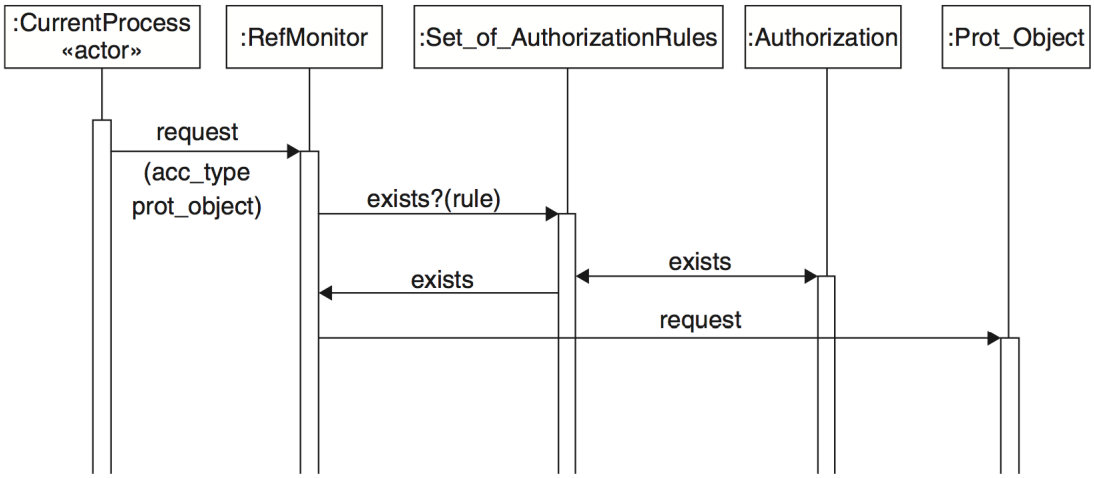


Abbildung 1.6.: Reference Monitor - Sequenzdiagramm [Sch+06]

Dieses Vorgehen leitet vom *Interceptor* Pattern ab, und findet an vielen anderen Orten Verwendung (JEE Servlet Filter usw.)

Vor- & Nachteile

- Wenn sichergestellt werden kann, dass alle *Requests* überprüft werden können, so ist eine maximale Befriedigung der Sicherheitsanforderungen gewährt.
- Jede Resource benötigt ihre eigene Implementierung eines *Reference Monitors*; Ein *Request* auf eine Datei muss evtl. anders behandelt werden als ein *Request* auf eine spezifische Datenbanktabelle.
- Die Prüfung vieler *Requests* kann bei hoher Systemlast zum Performancerisiko führen. Dementsprechend sollte die Logik zur Sicherheitsprüfung auch so einfach/-schlank wie möglich gehalten werden.

Beispielanwendungen

- Datenbanksysteme
- Betriebsysteme (bspw. Windows 2000 ff. verwendet eine ACL für NTFS Berechtigungen)

1.5. Role Rights Definition

Beim Definieren von Sicherheitsrichtlinien spielt das *Least Privilege* oder auch das *Need to know* Prinzip eine fundamentale Rolle: Jedes Subjekt soll gerade so viele Berechtigungen erhalten, damit es seine Aufgaben ungehindert erledigen kann.

Das *Role Rights Definition* Pattern beschreibt einen systematischen Ansatz, wie aus vorhandenen *Requirements Engineering* Artefakten *Need to Know*-konforme Sicherheitsregeln gewonnen werden können

Kontext

Eine relativ komplexe Ansammlung von Rollen soll mit passenden Berechtigungen ausgestattet werden.

Problem

Role Based Access Control wird in vielen Systemen als grundlegendes Sicherheitkonzept verwendet. Wie im Abschnitt 1.2 erwähnt ist die Definition von Berechtigungskonzepten bei umfangreichen System (und grosser Anzahl an Aufgabenbereichen) mit beträchtlichem Aufwand verbunden.

Zudem überlässt *Role Based Access Control* es komplett dem Implementator, aufgrund von welchen Informationen Gruppen resp. deren Berechtigungen zusammengestellt werden.

Wie können wir *Role Based Access Control* mit Sicherheitsrichtlinien füttern, welche folgende Punkte befriedigen?

- Rollen sollen Aufgabenbereichen in der Organisationsstruktur entsprechen

- Rechte sollen so erteilt werden, dass sie dem *Need to know* Prinzip genügen
- Weiterhin soll die Anpassung bestehender Rollen und Rechten so einfach wie möglich bleiben
- Die Definition von Rechten und Rollen soll unabhängig von einer effektiven Implementierung des Systems bleiben

Lösung

Die Idee ist denkbar einfach: Ein (hoffentlich bestehendes) Use Case Model und die damit verbundenen Sequenzdiagramme werden dazu verwendet, alle von *Role Based Access Controls* benötigten Elemente zu erfassen:

- Ein *Actor* entspricht einer *Role*
- Jegliche *Objects* entsprechen einem potentiellen *ProtectionObject*
- Jede *Operation* welche ein *Actor* auf einem *Object* ausführt, ist ein potentielles *Right* einer *Role*
- Eine *Use Case Exception* bestimmt das Verhalten im Falle einer Verletzung einer Sicherheitsrichtlinie

Vorteile

- Sicherheitsrichtlinien können, bei entsprechendem Projektvorgehen, bereits sehr früh definiert und erkannt werden.
- Wird ein "*model driven*"-Ansatz für die Softwareentwicklung gewählt, können Sicherheitsrichtlinien im optimalsten fall "einfach" aus den bestehenden Requirements Artefakten generiert werden
- *Role Rights Definition* erstellt "perfekte" Sicherheitsrichtlinien für *RBAC*
- Sind alle Use Cases modelliert, und das System kann auf diese Weise komplett abgebildet werden, so ist ein Maximum an Sicherheit garantiert
- Verändert sich die Funktionalität (sprich die Use Cases) des Systems (neuer Release etc.), so können auch die damit verbundenen Änderungen im Sicherheitskonzept problemlos abgebildet werden.
- *Role Rights Definition* bleibt komplett implementationsneutral

Nachteile

- Ohne ausführliches, durchgehendes und kompetentes Requirements Engineering hat dieses Pattern so gut wie keinen Nutzen

Mögliche Prüfungsfragen

- *Für welches Pattern ist der "Output" von Role Rights Definition bestens geeignet? Warum?*
  Role Rights Definition analysiert Use Cases und extrahiert daraus aufgaben- und funktionsbezogene Zugriffsberechtigungen für alle vorhandenen *Actors*.
  Diese Regeln entsprechen dem *Need to know* Prinzip: Jeder *Actor* kann genau das tun/sehen, was er zu Ausübung seiner Aufgaben tun/sehen können muss.
  Damit sind eben diese Regeln optimal für die Verwendung im *RBAC* Pattern geeignet.

- *Warum reicht es nicht aus, lediglich das Use Case Model zur Gewinnung von Roles und Rights zu analysieren?*
  Die Sequenzdiagramme geben detailierte Auskunft darüber, zu welchem Zeitpunkt welcher *Actor* welches *Right* für welches explizite *Protection Object* benötigt. Ohne diese Informationen ergibt sich ein unvollständiges Gesamtbild.

Anhang A **Abbildungen, Tabellen & Quellcodes**

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Anhang B Literatur

[Sch+06] Markus Schumacher u. a. *Security Patterns - Integrating Security and Systems Engineering*. 1. Aufl. John Wiley & Sons, Ltd, 2006. ISBN: 978-0-470-85884-4.

[wika] wikipedia.org. *Bell-LaPadula-Sicherheitsmodell*. URL: `http://de.wikipedia.org/wiki/Bell-LaPadula-Sicherheitsmodell` (besucht am 03. 03. 2013).

[wikb] wikipedia.org. *Biba-Modell*. URL: `http://de.wikipedia.org/wiki/Biba-Modell` (besucht am 03. 03. 2013).

Anhang C Glossar

ACL

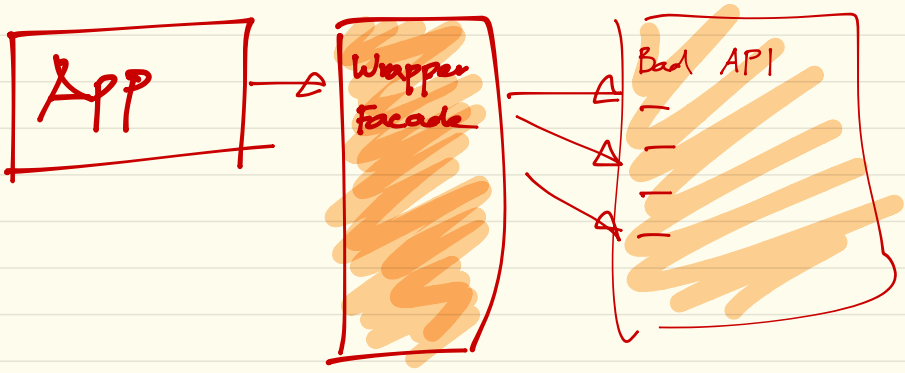
Access Control List; eine Liste mit Zugriffsregeln für eine bestimmte Resource. 10

RBAC

Role Based Access Control; Siehe Abschnitt 1.2. 13, 14

Anhang D **Workshops**

Wrapper Facade



- Kann auch mehr Logik enthalten
- Typensicherheit
- Legacy-Code verpacken für "Heute"
- Nachteile:
  Performance kann zum Problem werden
- Vorteile: Bessere API's
- Knackpunkte:
  Wrapper soll semantisch korrekt bleiben (zusammen was zusammen gehört

Anmerkungen P. Sommerlad:
- Error-Handling ist wichtig ⇒ Auch hier wrappen!
  ↳ Falls nötig Domain-spezifische Errors
- Wann nicht anwendbar?
  - Wrapper vom Wrapper vom Wrapper
- Stichworte:
  Async vs. Sync (Async ist schneller, da Buffer kopieren evtl. gespart werden kann)

Fault Tolerance Systems

Introduction: Zusammenhang Fault, Error & Failure
Fault: Bug, Ursache
Error: Zustand
Failure Effektives Problem
↳ Zu vermeidendes Problem

- Failure definieren sich im Normalfall durch Abweichung von der Spec

- Unterschiedliche Faults können zu gleichen Errors/Failures führen
- Coverage: Wahrscheinlichkeit dass sich ein System innert gegebener Zeit wieder erholen kann: Mean Time To Failure + Mean Time To Recover } Mean Time Between Failure

↳ Relaibility: $e^{-\frac{1}{MTTF}}$

- FIT: $\frac{\#\text{Failures}}{1 \cdot 10^9 h}$ ⇒ Failures in Time

⇒ Stichwort: Server-Zuverlässigkeit

Fail Silent: Bei Fehler übernimmt automatisch andere Komponente
Fail Consistency: Man muss herausfinden welche Systemkomponenten fehlerhaft sind
Malicious Failure: Man kann nicht einfach herausfinden welche Systeme fehlerhaft sind ⇒ Byzantinische Generäle zur Abstimmung