

Advanced Patterns And Frameworks

# Zusammenfassung & Notizen

Hochschule für Technik Rapperswil

Frühjahressemester 2013

**Autoren** Manuel Alabor (MAL)  
**URL** <http://swissmanu.github.com/hsr-apf-2013/patterns.pdf>  
**Build** 19. März 2013, 12:15

# Inhaltsverzeichnis

1.	Access Control Models . . . . .	3
1.1.	Authorization . . . . .	3
1.2.	Role Based Access Control . . . . .	5
1.3.	Multilevel Security . . . . .	7
1.4.	Reference Monitor . . . . .	10
1.5.	Role Rights Definition . . . . .	12
2.	Identification & Authentication . . . . .	15
2.1.	Einführung . . . . .	15
2.2.	I&A Requirements . . . . .	16
3.	System Access Control Architecture . . . . .	20
3.1.	Access Control Requirements . . . . .	20
3.2.	Single Access Point . . . . .	22
A.	Abbildungen, Tabellen & Quellcodes . . . . .	25
B.	Literatur . . . . .	26
C.	Glossar . . . . .	27
D.	Workshops . . . . .	28

# 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 Objekten 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:

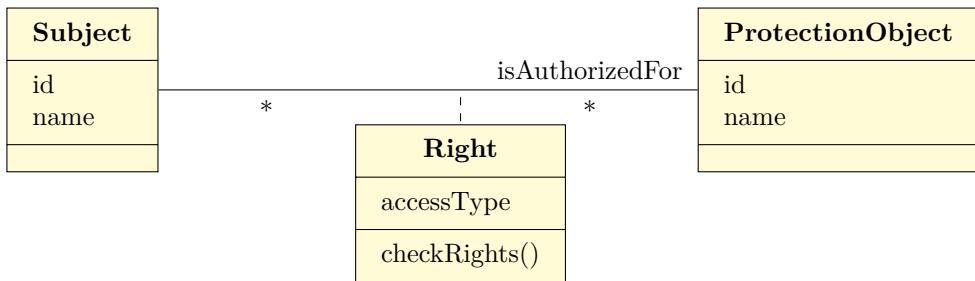


Abbildung 1.1.: Authorization Klassendiagramm

- Subject beschreibt jegliche Aspekte des zu berechtigenden Subjekts
- Das ProtectionObject ist das zu schützende Objekte
- Right enthält alle Informationen, wie Subject auf ProtectionObject zugreifen 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, Organisationsstrukturen, 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").

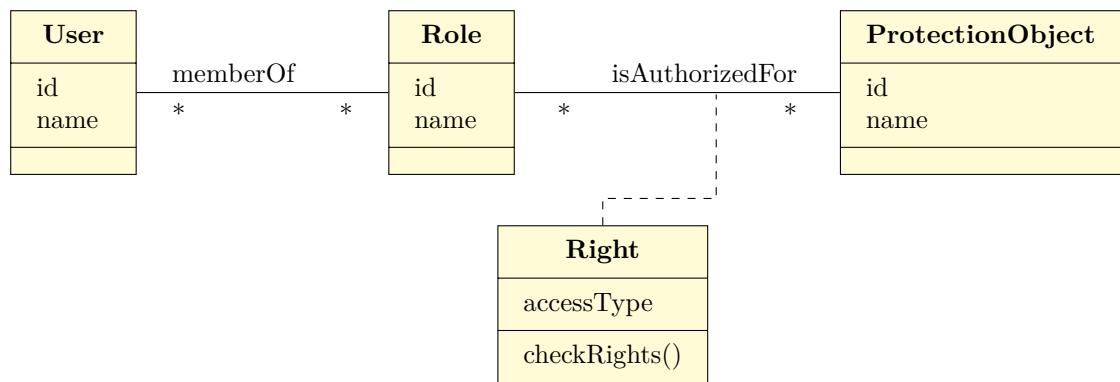


Abbildung 1.2.: Basic Role Based Access Control Klassendiagramm

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

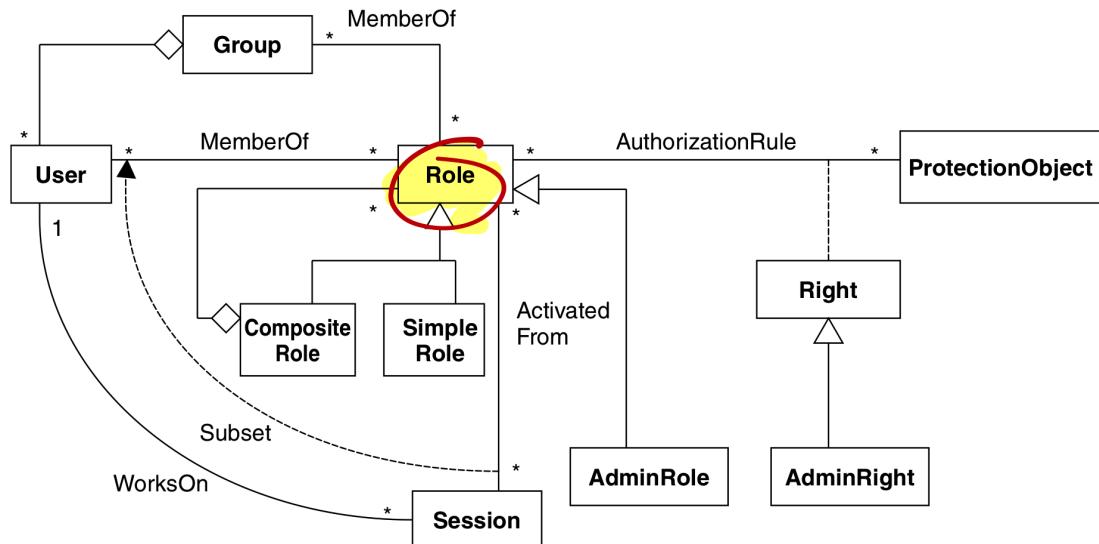


Abbildung 1.3.: RBAC mit Composite, Admins &amp; Abstract Session

### 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 Organisationsstruktur (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" zugewiesen. 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 Personaldaten zugriff oder kann nur Dateien mit einem USB-Stick austauschen.

*wackeliges Beispiel ;-)*

## 1.3. Multilevel Security

Oft sollen Informationen in verschiedene Sicherheitskategorien eingesortiert 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 Organisationsstruktur 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 können
- Die Sensitivität einer Information muss so einfach wie möglich angepasst werden können

## Lösung

Jeder Information wird ein *Sensitivity Level* zugewiesen. *Policies* definieren, welche Subjekte aus der Organisationsstruktur 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 Ressourcen tiefer eingestufter Subjekte schreiben (Informationsweitergabe!)
3. *Zugriffsmatrix*  
Matrix, welche Zugriffsberechtigungen von Subjekten auf Ressourcen festlegt

Die Korrektheit der *Policies* wiederum wird über das Biba-Modell[wikb] (der Umgehung 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

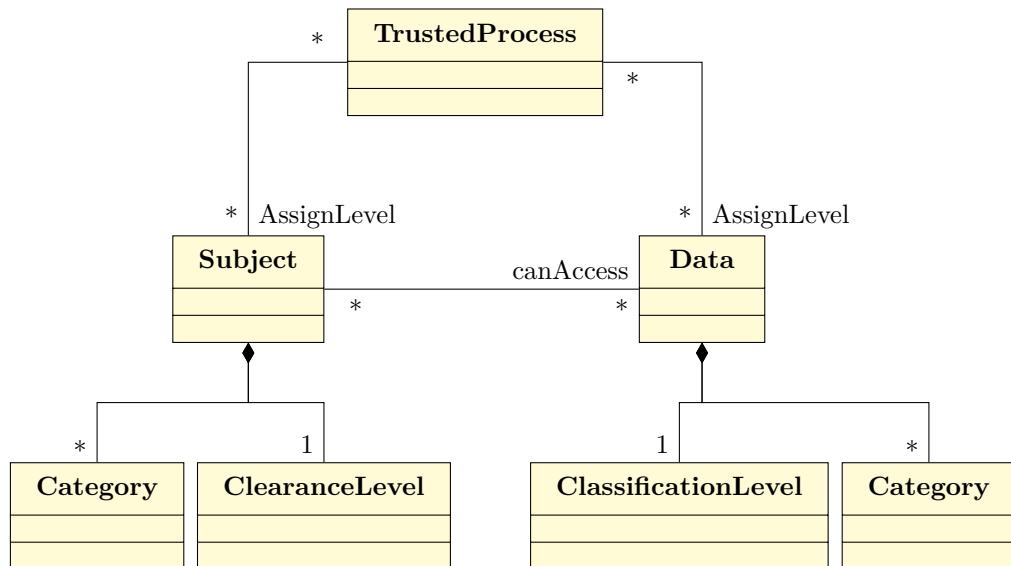


Abbildung 1.4.: Multilevel Security Klassendiagramm

## 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 kritische 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 komponiert werden: Dabei werden die *Clearance Levels* einfach auf die Gruppen statt direkt auf die Benutzer zugewiesen.

## Beispielanwendungen

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

## 1.4. Reference Monitor

*aka Policy Enforcement Point*

Das *Reference Monitor* Pattern beschreibt eine abstrakte Vor gehensweise, wie definierte Sicherheitsvorschriften um- und vor allem durchgesetzt werden können.

### Kontext

Ein IT-System, in welchem Subjekte (Benutzer als auch technische Prozesse) auf diverse Ressourcen zugreifen 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.

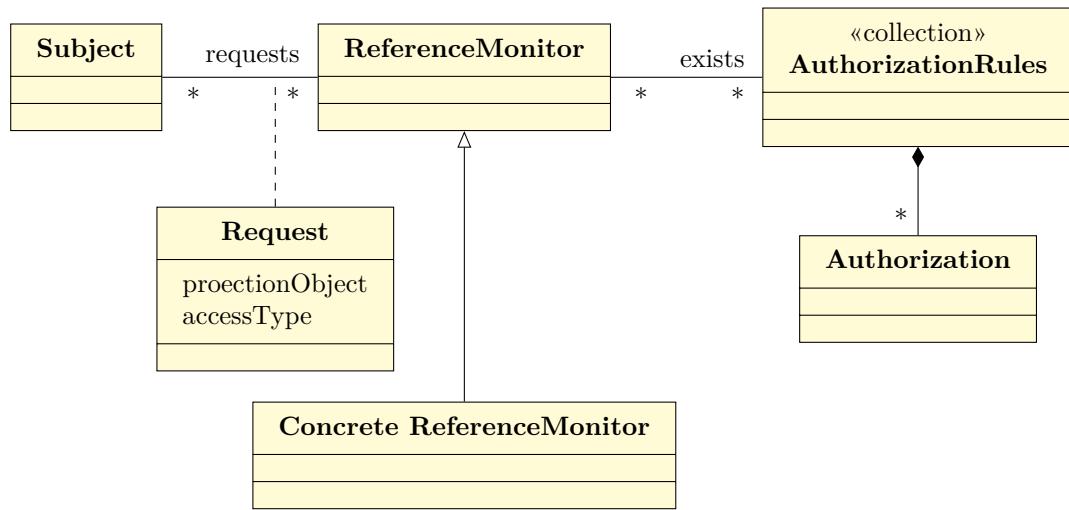


Abbildung 1.5.: Reference Monitor - Klassendiagramm

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.

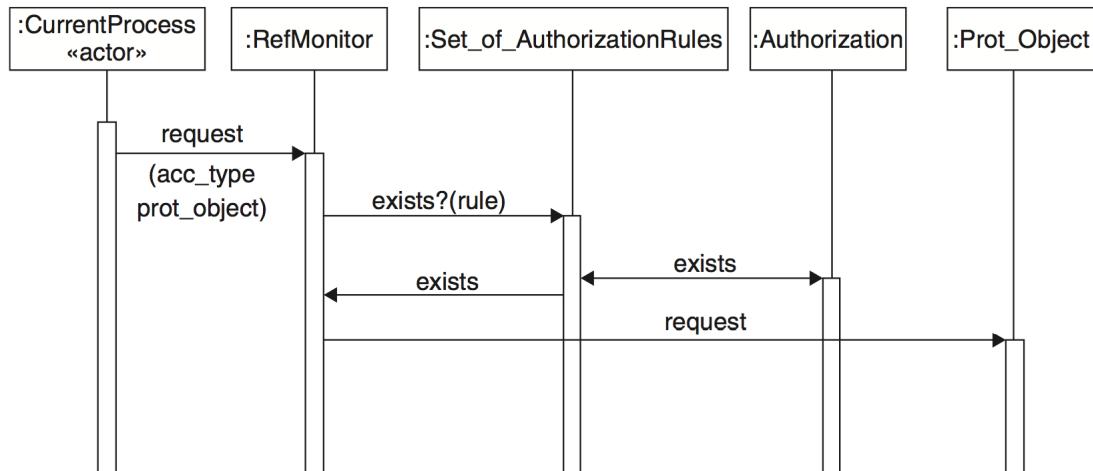


Abbildung 1.6.: Reference Monitor - Sequenzdiagramm [Sch+06]

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
- Betriebssysteme (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 detaillierte Auskunft darüber, zu welchem Zeitpunkt welcher *Actor* welches *Right* für welches explizite *Protection Object* benötigt. Ohne diese Informationen ergibt sich ein unvollständiges Gesamtbild.

## Kapitel 2 Identification & Authentication

### 2.1. Einführung

“Identification & Authentication” (I&A) fasst folgende zwei Schritte zusammen:

1. Feststellen der Identität eines Subjektes sowie Verbindung zu einer im System abgelegten ID herstellen (Identification)
2. Mittels einem Authenticator<sup>1</sup> prüfen, ob Subjekt wirklich für die ermittelte ID berechtigt ist (Authentication)

Für dieses grundlegende Schema gibt es zwei verschiedene Varianten:

1. Ein Subjekt wird mit einer eindeutigen Identität in Verbindung gebracht (Individual I&A)
2. Ein Subjekt wird lediglich auf die Zugehörigkeit zu einer Gruppe geprüft (Group I&A)  
Beispiel: Wache prüft jede Person an der Pforte, ob er einen Mitarbeiterbadge bei sich trägt.

Um I&A einsetzen zu können ist eine Reihe weiterer (aktiver und passiver) Komponenten nötig:

- *Subjektregistrierung*: Ein Subjekt muss initial registriert werden, damit es später wieder identifiziert und authentifiziert werden kann
- *Sessionmanagement*: Schlagwort Single-Sign-On
- *Gesicherte Systemkomponenten, “Using function”*: Komponenten, welche I&A aufrufen und dessen Output verwenden (z.B. Patterns aus dem Kapitel 1)

---

<sup>1</sup> Als Authenticator gilt z.B. ein Passwort, Hardwaretoken, Streichliste usw.

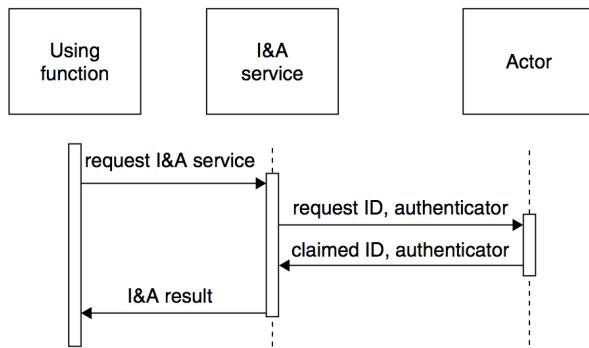


Abbildung 2.1.: Generischer Ansatz von I&A “Using functions” [Sch+06]

## Mögliche Prüfungsfragen

- *Was ist ein Authenticator?*

Nachdem ein Subjekt mit einer im System abgelegten Identität in Verbindung gebracht wurde, wird der Authenticator verwendet, um sicherzustellen, dass das Subjekt auch wirklich das Subjekt ist, für welches es sicht ausgibt.

Beispiel: Nach Eingabe des Benutzernamens wird das Passwort als Authenticator verwendet.

- *Welche grundlegenden Typen von I&A unterscheidet man?*

Individual und Group Identification & Authentication

## 2.2. I&A Requirements

Muss ein I&A Service etabliert werden, hilft das I&A Requirements Pattern mit seinen generischen Requirementsvorlagen bei der Analyse eines bestehenden oder zu konzipierenden Systems.

Dabei werden nicht nur sicherheitsrelevante Faktoren berücksichtigt. Aspekte wie Kosteneffektivität oder Benutzerzufriedenheit und -Akzeptanz fließen ebenso in die Analyse mit ein.

### Kontext

Eine Organisation oder ein Projekt konzipiert die Verwendung von I&A. Das Pattern unterstützt die Analyse jeglicher Situationen, in welchen sowohl Identification als auch Authorization notwendig ist.

### Problem

Der Natur nach können Anforderungen oftmals im Konflikt zueinander stehen. Insbesondere im Bereich der I&A können hohe Sicherheitsanforderungen nicht mit dem tiefen Projektbudget vereinbar sein.

Wie können nun aber eben diese Anforderungen auf die aktuelle Situation angepasst miteinander in Einklang gebracht werden?

### Lösung

Das I&A Requirements Pattern definiert folgende Vorgehensweise:

1. *Requirements Specification*

Generische Requirementsvorlagen im Systemdesign-Prozess aufgreifen und auf eigene Situation anpassen

2. *Prioritization Process*

Die Menge an angepassten, generischen Requirements wird nun gem. der aktuellen Situation priorisiert

### Generische Requirementsvorlagen

Anforderung	Erläuterung
Accurately Detect Imposters	Requests von unberechtigten Actors sollen als solche erkannt werden.
Accurately Recognize Legitimate Actors	Korrekte Requests an den Service sollen auch als solche erkannt werden.

Tabelle 2.1.: I&A Requirements: Funktionale Anforderungen

Die beiden funktionalen Anforderungen stehen praktisch immer in gegenseitiger Wechselwirkung: Werden mehr Requests als *Falsch* klassifiziert, erwischt man automatisch auch mehr Requests, welche eigentlich *Richtig* gewesen wären.

Anforderung	Erläuterung
Minimize Mismatch with user Characteristics	Unterschiedliche Wissenstände, Umgebungseinflüsse (Standort ...) usw. von Actors sollen zu so wenigen wie möglichen Fehlinterpretationen von Service Requests führen.
Minimize Time and Effort to Use	Bspw. Zeitaufwand für mehrmaliges eintippen des Passworts soll verhindert werden
Minimize Risks to User Safety	Beispiel: Retina Scanner muss mit Gasmaske funktionieren; dies steht im Konflikt mit der Genauigkeit der Re却esterkennung
Minimize Costs of Per-user Setup	
Minimize Changes Needed to Existing System Infrastr.	Soll der I&A Service in ein bestehendes System integriert werden, sollen die anfallenden Änderungen in der bestehenden Infrastruktur ggf. minimiert werden
Minimize Costs of Maintenance, Management & Overhead	
Protect I&A Service and Assets	Wie wichtig ist der Schutz des I&A Services und er zu schützenden Objekte?

Tabelle 2.2.: I&amp;A Requirements: Nichtfunktionale Anforderungen

### Analogie: Anlagestrategie im Finanzsektor

Es können nie alle Anforderung gleich gut abgedeckt werden. Wie bei einer Anlagestrategie (Dreieck *Liquidität, Sicherheit, Rentabilität*) müssen alle Anforderungen analysiert und auf die eigene Situation/Präferenzen zugeschnitten werden.

### Vorteile

- Eine ausführliche Domain- und Anforderungsanalyse wird gefördert.
- Die vorliegenden Requirementsvorlagen fördern die ausführliche Auseinandersetzung mit den verschiedensten Einflüssen auf I&A.
- Als angenehmen Nebeneffekt erhält man eine umfangreiche Dokumentation über die I&A Aspekte des Systems.

### Nachteile

- Der Aufwand zur Umsetzung dieses Patterns kann tendenziell sehr Resourcenintensiv sein (Anforderungsanalyse, Priorisierung etc. etc.)
- Die vielen Ausprägungen der einzelnen Anforderungen können leicht in einem Over-Engineering enden. Diese Gefahr kann jedoch durch pragmatische Herangehensweise (Verwendung als Guidelines) minimiert werden

- Da eine umfangreiche Dokumentation als Resultat des Patterns entsteht, besteht natürlich auch die Gefahr, dass diese im Laufe der Zeit nicht mehr aktualisiert wird.

### Mögliche Prüfungsfragen

- *Gibt es ein I&A Patentrezept?*

Nein. Jedes System kommt mit seinen eigenen, spezifischen Anforderungen an I&A. Aus diesem Grund kann und sollte das I&A Requirements Pattern nur als Guideline/Vorlage zu eigenen spezifischen Implementierungen verwendet werden.

## Kapitel 3 **System Access Control Architecture**

### 3.1. Access Control Requirements

Das Pattern Access Control Requirements ist sehr mit dem aus I&A bekannten Pattern "I&A Requirements" zu vergleichen.

Statt Anforderungen für I&A zu definieren und zu erarbeiten, stellt "Access Control Requirements" eine Sammlung von allgemein gültigen Anforderungsschablonen zur Verfügung, welche das spezifizieren eine Massgeschneiderten Zugriffskontrolle ermöglichen.

#### Kontext

Eine Organisation oder ein Projekt konzipiert die Verwendung von Access Controls.

#### Problem

Der Natur nach können Anforderungen oftmals im Konflikt zueinander stehen. Insbesondere im Bereich von Access Control können hohe Sicherheitsanforderungen nicht mit dem tiefen Projektbudget vereinbar sein.

Wie können nun aber eben diese Anforderungen auf die aktuelle Situation angepasst miteinander in Einklang gebracht werden?

#### Lösung

Das Access Control Requirements Pattern definiert folgende Vorgehensweise:

1. *Requirements Specification*

Generische Requirementsvorlagen im Systemdesign-Prozess aufgreifen und auf eigene Situation anpassen

2. *Prioritization Process*

Die Menge an angepassten, generischen Requirements wird nun gem. der aktuellen Situation priorisiert

#### Generische Requirementsvorlagen

Folgende Anforderungen gilt es im Rahmen dieses Patterns zu analysieren und lösungsgerecht auszubalancieren:

Anforderung	Erläuterung
Deny unauthorized access	Unberechtigten Subjekten soll der Zutritt zu schützenswerten Objekten verwehrt werden
Permit authorized access	

Tabelle 3.1.: Access Control Requirements Requirements: Funktionale Anforderungen

Anforderung	Erläuterung
Limit the damage when unauthorized access is permitted	Kann ein unbefugtes Subjekt trotzdem Zugang zum System erhalten, soll der entstehende Schaden so klein wie möglich sein. Dies führt möglicherweise zu erneuten Sicherheitsprüfungen und erschwert für berechtigte Subjekte die alltägliche Nutzung des gesicherten Systems.
Limit the blockage when authorized access is denied	Wird ein grundsätzlich berechtigtes Subjekt abgewiesen, so sollen die Auswirkungen für dieses so klein wie möglich sein (Produktivität etc.)
Minimize burden of access control	Die Zugriffskontrolle soll nicht zur Bürde werden. Schlagworte wie Performance, Reaktionszeit usw. sind hier von grosser Bedeutung.
Support desired authorization policies	Meet the requirements ;-)
Make access control service flexible	Die Zugriffskontrolle soll nach Möglichkeit schnell anpassbar sein. Beispiel: Nach Terroranschlag erhöhte Sicherheitsstufe für zwei Monate, anschliessend wieder gewohntes Dispositiv.

Tabelle 3.2.: Access Control Requirements: Nichtfunktionale Anforderungen

## Vorteile

- Eine ausführliche Domain- und Anforderungsanalyse wird gefördert.
- Die vorliegenden Requirementsvorlagen fördern die ausführliche Auseinandersetzung mit den verschiedensten Einflüsse auf Access Control.
- Als angenehmen Nebeneffekt erhält man eine umfangreiche Dokumentation über den Access Control Aspekt des Systems.

## Nachteile

- Der Aufwand zur Umsetzung dieses Patterns kann tendenziell sehr Resourcenintensiv sein (Anforderungsanalyse, Priorisierung etc. etc.)

- Die vielen Ausprägungen der einzelnen Anforderungen können leicht in einem Over-Engineering enden. Diese Gefahr kann jedoch durch pragmatische Herangehensweise (Verwendung als Guidelines) minimiert werden
- Da eine umfangreiche Dokumentation als Resultat des Patterns entsteht, besteht natürlich auch die Gefahr, dass diese im Laufe der Zeit nicht mehr aktualisiert wird.

### Mögliche Prüfungsfragen

- *Gibt es ein Access Control Patentrezept?*

Nein. Jedes System kommt mit seinen eigenen, spezifischen Anforderungen an Access Control. Aus diesem Grund kann und sollte das Access Control Requirements Pattern nur als Guideline/Vorlage zu eigenen spezifischen Implementierungen verwendet werden.

## 3.2. Single Access Point

Der Single Access Point definiert einen klaren Zugangspunkt zu einem System. Die so entstehende Schnittstelle kann dazu verwendet werden, effektive Sicherheitsrichtlinien praktisch umzusetzen.

### Kontext

Subjekten soll Zugang zu einem System gewährt werden. Die Subjekte sollen bevor sie Zugang erhalten geprüft werden. Das System soll vor Beschädigung und Missbrauch geschützt werden.

### Problem

Gewährt man Subjekten Zugang zu den Komponenten eines Systems, ist deren Integrität automatisch in Gefahr.

Nun könnte man den Zugang zu jeder Komponente im System gesondert überprüfen. Dies macht im Bezug auf Performance und/oder Accessability meistens weniger Sinn (Subjekte wollen nicht mehrfach ein Passwort eingeben müssen oder sich wiederholt von einem Security-Mitarbeiter abchecken lassen müssen).

Weiter führt die wiederholte Implementierung der Sicherheitsrichtlinien unweigerlich zu höheren Kosten. Sei dies im Bereich der späteren Wartung oder bei Initialaufwänden. Erschwerend kommt im Bezug auf die Kosten hinzu, dass die meisten Komponenten im System nicht 1:1 miteinander vergleichbar sind und so evtl. nicht unbedingt gleich geschützt werden können.

## Lösung

Es wird ein Single Access Point (“ein einziger Zugangspunkt”) definiert, welcher die Sicherheitsrichtlinien umsetzen kann und welcher jegliche Subjekte, welche Zugang zum System erhalten wollen passieren müssen.

Dieser Single Access Point muss prominent platziert sein. Kann ein Subjekt ihn nicht finden, wird dieses kaum glücklich über die Sicherheitsmaßnahme sein.

Hat ein Subjekt den Single Access Point passiert, kann es sich im System frei bewegen.

Ist eine feinere Steuerung für den Zugriff auf Komponenten gewünscht, können Komponenten im System wiederum einen Single Access Point implementieren und so den Zugang zu sich selber prüfen.

Durch die Definition des Single Access Points definiert man auch eine Grenze, welche das System schützt. Es ist dabei wichtig nicht zu vergessen, dass entsprechender Aufwand nötig ist diese Grenze zu schützen/aufrecht zu erhalten (Bsp. Bau des Gitters um ein Areal, setzen der Firewall-Einstellungen etc.). Denn mit dieser Grenze steht und fällt die Sicherheitswirkung dieses Patterns.

Somit besteht die Umsetzung des Single Access Point Patterns aus folgenden Punkten:

1. Sicherheitsrichtlinien definieren
2. Single Access Point definieren (prominente Stelle etc.)
3. Effektive Prüfung der Sicherheitsrichtlinien umsetzen (Single Access Point kann auch einfach nur für Auditing/Logging verwendet werden)
4. Initialisierung des Systems (Session aufsetzen usw.)
5. Grenzen des Systems schützen (fortlaufend)

## Vorteile

- Ein einziger Zugangspunkt zum System vereinfacht die Komplexität und verbessert die User Experience
- Es muss keine wiederholte Implementierung der gleichen Sicherheitsprüfung umgesetzt werden
- Das Single Access Point Pattern kann auf verschiedensten Abstraktionsebenen umgesetzt werden
- Die interne Komplexität eines Systems kann möglicherweise vereinfacht werden, da der Sicherheitsaspekt “zentral” umgesetzt wird

## Nachteile

- Verfehlt ein Subjekt den Zugangspunkt, kann das System für ihn als nutzlos betrachtet werden

- Single Access Point <=> Single Point of Failure: Beim Ausfall des Zugangspunktes kann möglicherweise das gesamte System nicht mehr verwendet werden
- Der Zugangskontrolle muss vertraut werden können (erhöhter Aufwand für Lohn eines Wachmanns oder Schutzmassnahmen gegen Hacker etc.)
- Die Grenze des Systems ist und bleibt die schwächste Stelle im Sicherheitsdispositiv

### Reallife Beispiele

- Anmeldescreens verschiedenster Betriebssysteme
- Eingangskontrolle an einem Openair Festival  
Prominenz des Eingangs ist wichtig, da die Besucher sonst den Eingang nicht finden und vor den Absperrungen randalieren ;)
- Freizeitpark  
Nach einmaligem Bezahlen am Eingang hat man Zutritt zu allen Attraktionen (abgesehen von den Größenkontrollen bei den Achterbahnen). Ein Shuttlebus vom Parkplatz zum Eingang erleichtert es dem Besucher, den Eingang zu finden.
- Nachtclub  
Nach der Kontrolle beim Securitypersonal hat man freien Zugang zu allen Bars. Möchte man in den VIP-Bereich, ist eine weitere Kontrolle durch das Securitypersonal nötig (Eingeladen? Reserviert? Genug Bargeld? ;-) )  
Beispiel einer schlechten Systemgrenze: Der Notausgang kann auch verwendet werden, um sich Zutritt zu verschaffen

### Mögliche Prüfungsfragen

- *Nennen Sie ein Beispiel ausserhalb der IT-Welt, welche das Single Access Point Pattern umsetzen*  
Siehe “Reallife Beispiele”

## Anhang A **Abbildungen, Tabellen & Quellcodes**

### Abbildungsverzeichnis

1.1. Authorization Klassendiagramm . . . . .	4
1.2. Basic Role Based Access Control Klassendiagramm . . . . .	6
1.3. RBAC mit Composite, Admins & Abstract Session . . . . .	6
1.4. Multilevel Security Klassendiagramm . . . . .	9
1.5. Reference Monitor - Klassendiagramm . . . . .	11
1.6. Reference Monitor - Sequenzdiagramm [Sch+06] . . . . .	11
2.1. Generischer Ansatz von I&A “Using functions” [Sch+06] . . . . .	16

### Tabellenverzeichnis

2.1. I&A Requirements: Funktionale Anforderungen . . . . .	17
2.2. I&A Requirements: Nichtfunktionale Anforderungen . . . . .	18
3.1. Access Control Requirements Requirements: Funktionale Anforderungen .	21
3.2. Access Control Requirements: Nichtfunktionale Anforderungen . . . . .	21

### 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*

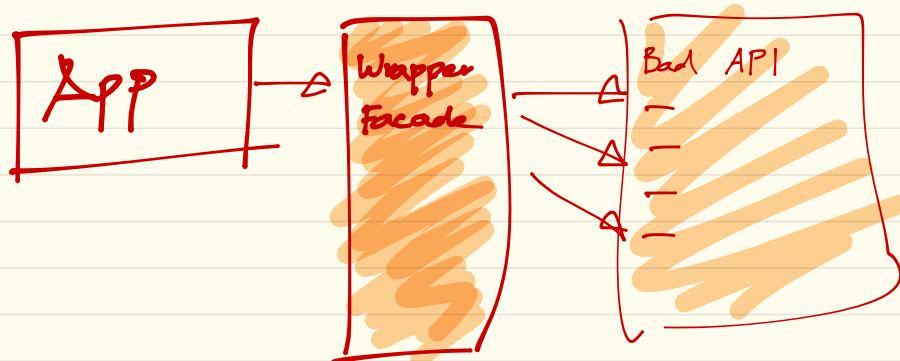
Access Control List; eine Liste mit Zugriffsregeln für eine bestimmte Resource. 10

### *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. Sonnenlad:

- Error-Handling ist wichtig  $\Rightarrow$  Auch hier wrappen!

$\hookrightarrow$  Falls nötig Domain-spezifische Errors

- Wenn nicht anwendbar?

- Wrapper vom Wrapper vom Wrapper

- Stärke:

Async vs. Sync (Async ist schneller, da Bufferkopieren entl. gespart werden kann)

# Fault Tolerant 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 innerhalb gegebener Zeit wieder erholen kann: Mean Time To Failure } Mean Time Between Failure  
Mean Time To Recover

$$\hookrightarrow \text{Reliability: } e^{-\frac{t}{MTTF}}$$

- FIT:  $\frac{\# \text{ Failures}}{1 \cdot 10^3 \text{ h}}$  ⇒ Failures in Time

⇒ Stichwort: Server-Zuverlässigkeit

**Fault Silent:** Bei Fehler übernimmt automatisch andere Komponente

**Fault Consistency:** Man muss herausfinden welche Systemkomponenten fehlerhaft sind

**Malicious Failure:** Man kann nicht einfach herausfinden welche Systeme fehlerhaft sind ⇒ Byzantinische Generäle zur Abstimmung

# Architekturpatterns Fault Tolerance

12.03.2013

"Allgemeingültige" Patterns für gesamte Architekturen

## Units of Mitigation

Problem: Fehler soll nicht gesamtes System beeinträchtigen  
⇒ Beschränkung auf "Unit", bspw. try-catch-Block  
⇒ Unitgröße ist essentiell (zu gross: sinnlos,  
zu klein: Code-Aufwand)

Lösung: Aufteilen in Units, jede Unit enthält Logik für eigene Fehler

Beispiele für Units: → Funktionsgruppen

- try-catch, CPU-Cores, Threads, Layers, Interfaces

Fehler bleiben Unit-spezifisch ⇒ Erkennung & Behebung bleiben intern

Was passiert solange eine Unit mit Fehlerbehandlung beschäftigt ist?

- Abhilfe durch Redundanz (AKW-Kühlsystem)
- Queering

## Correcting Audits

Begriffe: statische Daten: User ID, dynamische: Wechselkurs

Problem: Defekte Daten sollen so früh wie möglich erkannt und korrigiert werden. Werden solche Daten gefunden, wird geprüft, wie weit sich der Fehler evtl. schon ausgebreitet hat.

Lösung: Finden: Strukturelle Fehler; Zusammenhänge (Vorsch. Umrechnungen des selben Wertes), macht der Wert Sinn?

⇒ Datendesign für einfache Prüfung anlegen

Korrigieren: Direkt vom Programm

Repeat Finden, um Korrekturen zu prüfen

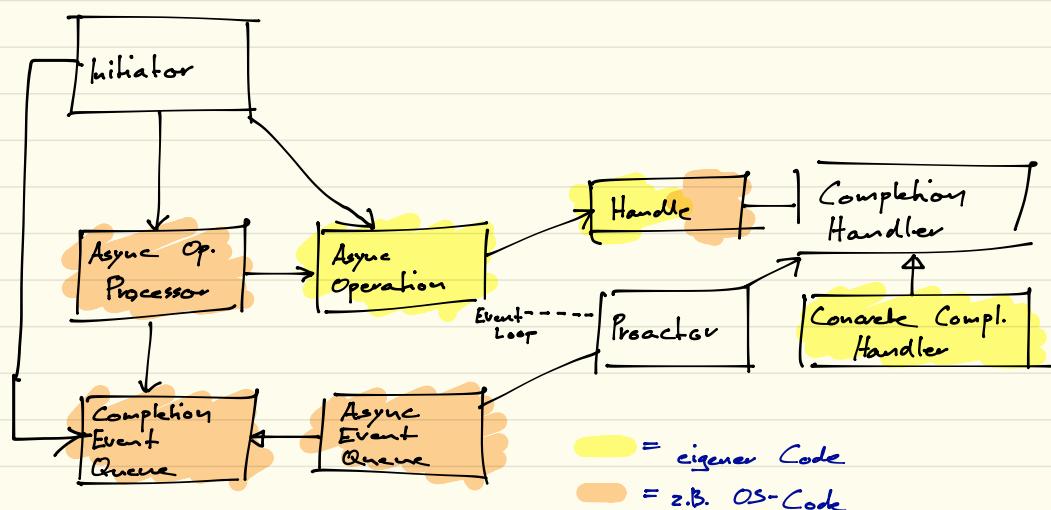
### Escalation

Problem: Was passiert, wenn Fehler immer wieder auftritt?

Lösung: Fehler als externe Instanz nach aussen weitergeben

- bspw. Operator
- bspw. Fehlerbehandlungsdienst.

### Proactor



⇒ NS Operation Queue? Warum "Proactor"? ⇒ Pattern arbeitet selbstständig eine Queue ab.

Vergleich Reactor: Reactor "antwortet sofort", Proactor entscheidet selber wann er welche gequeckten Operations ausführen soll  
⇒ ermöglicht z.B. Priorisierung der Operations

Knackpunkte: Asynchrone Entwicklung vs. sequentiell

Vorteile:

- Parallelisierung I/O & Completion Handler
- Tendenziell robuster, da entkoppelt (Async hält)

(⇒ Warum Async I/O schneller? OS-näher)

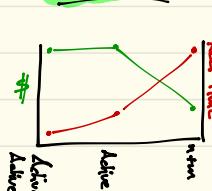
## Fault Tolerant Systems: Redundancy

Redundanz sagt nicht, dass "Not-System" identisch sein muss... 13.03.2013

### Typen

- ① Räumliche Redundanz (z.B. Hardware aufteilen etc)
- ② Zeitliche Red. (Berechnungen wiederholt ausführen)
- ③ Informatisch (Daten mehrmals abgelegt zum Vergleichen)  
↳ z.B. Punkte speichern, dann Distanz berechnen

### Räuml. Red.



• Aktiv-Aktiv Alle Redundanzen immer aktiv (Load Balancing...)

• Aktiv-Passiv Redundanz ist passiv, bis sie gebraucht wird (Notstrom)

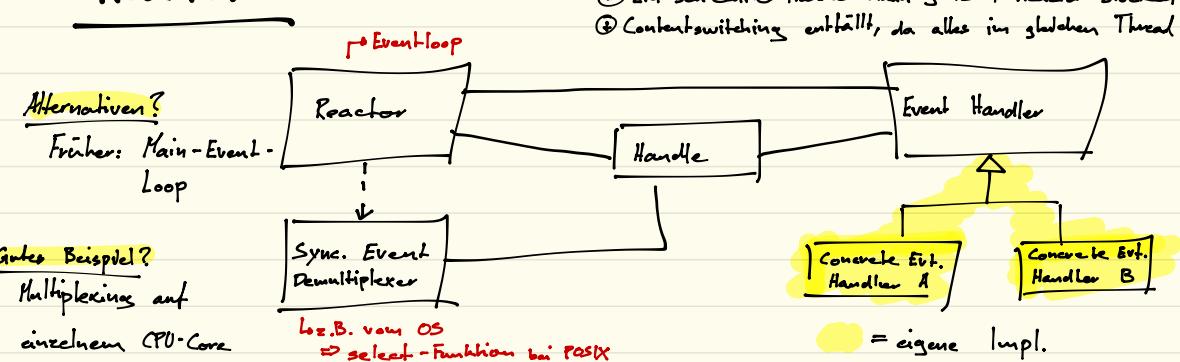
• n+m 5 aktiv, 3 passiv  $\Rightarrow$  Abwärmen, optimieren

Recovery Blocks Gleiche Aufgaben von verschiedenen Implementierungen ausführen lassen, Ergebnis vergleichen & bestes wählen  
 $\Rightarrow$  n-Version-Programming

Beispiel: Verschiedene Sort-Algos



## Reactor



Command Processor vs Reactor  
 Reactor looppt über registrierte Events, und führt entsprechende Handler aus  
 Command Processor führt lediglich Code aus, ohne "Events"

Was tun gegen Freeze?  
 Evtl. auslagern in eigene Threads