

Advanced Patterns And Frameworks

# Zusammenfassung & Notizen

Hochschule für Technik Rapperswil

Frühjahressemester 2013

**Autoren** *Teilnehmer APF FS 2013*  
**URL** <http://swissmanu.github.io/hsr-apf-2013/patterns.pdf>  
**Build** 12. August 2013, 20:00

# Inhaltsverzeichnis

1.	POSA 1 . . . . .	7
2.	POSA 2 . . . . .	8
2.1.	Wrapper Facade . . . . .	8
2.2.	Interceptor . . . . .	9
2.3.	Proactor . . . . .	12
2.4.	Reactor . . . . .	16
2.5.	Asynchronous Completion Token . . . . .	20
2.6.	Acceptor Connector . . . . .	23
2.7.	Component Configurator . . . . .	26
2.8.	Active Object . . . . .	29
2.9.	Monitor Object . . . . .	31
2.10.	Scoped Locking . . . . .	34
2.11.	Strategized Locking . . . . .	35
2.12.	Thread-Safe Interface . . . . .	37
2.13.	Half-Sync/Half-Async . . . . .	38
2.14.	Leader/Followers . . . . .	40
3.	POSA 3 . . . . .	45
4.	Security Patterns . . . . .	46
4.1.	Access Control Models . . . . .	46
4.1.1.	Authorization . . . . .	46
4.1.2.	Role Based Access Control . . . . .	48
4.1.3.	Multilevel Security . . . . .	50
4.1.4.	Reference Monitor . . . . .	53
4.1.5.	Role Rights Definition . . . . .	55
4.2.	Identification & Authentication . . . . .	57
4.2.1.	I&A Requirements . . . . .	58
4.3.	System Access Control Architecture . . . . .	61
4.3.1.	Access Control Requirements . . . . .	61
4.3.2.	Single Access Point . . . . .	63
4.3.3.	Check Point . . . . .	65

4.3.4. Security Session . . . . .	66
4.4. Firewall Architectures . . . . .	69
4.4.1. Packet Filter Firewall . . . . .	69
4.4.2. Proxy Based Firewall . . . . .	72
4.4.3. Stateful Firewall . . . . .	74
4.5. Secure Internet Applications . . . . .	76
4.5.1. Information Obscurity . . . . .	76
4.5.2. Secure Channels . . . . .	78
4.5.3. Protection Reverse Proxy . . . . .	82
4.5.4. Integration Reverse Proxy . . . . .	84
4.5.5. Front Door . . . . .	87
4.6. Operating System Access Control . . . . .	89
4.6.1. Authenticator . . . . .	89
4.6.2. Controlled Process Creator . . . . .	91
4.6.3. Controlled Object Factory . . . . .	92
4.6.4. Controlled Object Monitor . . . . .	93
5. Fault Tolerance . . . . .	96
5.1. Introduction . . . . .	96
5.1.1. Fault, Error, Failure . . . . .	96
5.1.2. Coverage . . . . .	98
5.1.3. Reliability . . . . .	98
5.1.4. Availability . . . . .	98
5.1.5. Dependability . . . . .	99
5.1.6. Performance . . . . .	99
5.2. Fault Tolerant Mindset . . . . .	100
5.2.1. Design Tradeoffs . . . . .	100
5.2.2. Quality vs Fault Tolerance . . . . .	100
5.2.3. Keep it Simple (KIS) . . . . .	101
5.2.4. Incremental Additions of Reliability . . . . .	101
5.2.5. Defensive Programming . . . . .	101
5.2.6. The Role of Verification . . . . .	102
5.2.7. Fault Insertion Testing . . . . .	102
5.2.8. Fault Tolerant Design Methodology . . . . .	102
5.2.9. Fragen . . . . .	103
5.3. Introduction to the Patterns . . . . .	104
5.3.1. Shared Context for These Patterns . . . . .	104
5.3.2. Terminology . . . . .	105
5.3.3. Fragen . . . . .	105
5.4. Units of mitigation . . . . .	106
5.4.1. Escalation . . . . .	109
5.4.2. Redundancy . . . . .	110
5.4.3. Recovery Blocks . . . . .	112

5.4.4. Minimize Human Intervention . . . . .	113
5.4.5. Someone In Charge . . . . .	115
5.4.6. Maximize Human Participation . . . . .	116
5.4.7. Maintenance Interface . . . . .	117
5.4.8. Fault Observer . . . . .	117
5.4.9. Software Update . . . . .	118
5.5. Detection Patterns . . . . .	119
5.5.1. Fault correlation . . . . .	121
5.5.2. Error containment barrier . . . . .	124
5.5.3. Riding over transients . . . . .	124
5.5.4. Leaky bucket counter . . . . .	125
5.6. Error Recovery Patterns . . . . .	126
5.6.1. Restart . . . . .	129
5.6.2. Roll Back . . . . .	130
5.6.3. Roll Forward . . . . .	130
5.6.4. Return to Reference Point . . . . .	131
5.6.5. Limit Retries . . . . .	131
5.7. Error Mitigation Patterns . . . . .	132
5.7.1. Overload Toolboxes . . . . .	135
5.7.2. Deferrable Work . . . . .	136
5.7.3. Reassess Overload Decision . . . . .	137
5.7.4. Equitable Resource Allocation . . . . .	138
5.7.5. Queue For Resources . . . . .	139
5.7.6. Expansive Automatic Controls . . . . .	141
5.7.7. Protective Automatic Controls . . . . .	141
5.7.8. Marked Data . . . . .	142
5.7.9. Error Correcting Codes . . . . .	142
5.7.10. Shed Load . . . . .	143
5.7.11. Final Handling . . . . .	143
5.7.12. Share the load . . . . .	143
5.7.13. Slow it down . . . . .	144
5.7.14. Shed work at periphery . . . . .	144
5.7.15. Finish work in progress . . . . .	145
5.7.16. Fresh work before stale . . . . .	145
5.8. Fault Treatment Patterns . . . . .	146
5.8.1. Let Sleeping Dogs Lie . . . . .	147
5.8.2. Reproducible Error . . . . .	148
5.8.3. Small Patches . . . . .	149
5.8.4. Root Cause Analysis . . . . .	149
5.8.5. Revise Procedure . . . . .	151
A. Abbildungen, Tabellen & Quellcodes . . . . .	152
B. Literatur . . . . .	155

C. Glossar . . . . .	157
D. Workshops . . . . .	158

## **Einleitung**

Dieses Dokument enthält zusammengetragene Ergebnisse aus den Workshops im Rahmen des Modules “Advanced Patterns and Frameworks” an der Hochschule für Technik in Rapperswil (HSR) vom Frühjahressemester 2013.

Alle Texte sind in Originalform (Links in Spalte “Übung”) im HSR Wiki einsehbar:

- <http://wiki.ifs.hsr.ch/APF/wiki.cgi?APF&1.27>

# Kapitel 1 **POSA 1**

## Kapitel 2 **POSA 2**

### 2.1. Wrapper Facade

The Wrapper Facade design pattern encapsulates low-level functions and data structures within more concise, robust, portable, and maintainable object-oriented class interfaces.

#### Problem

System-APIs und -Libraries sind oft sehr low-level und nicht objekt-orientiert aufgebaut. Will man diese Funktionen nutzen, muss man ausserdem viele Conditions für die unterschiedlichen Plattformen (Windows, Unix, ...) einbauen.

```
1 #if defined (_WIN32)
2     EnterCriticalSection (&lock);
3 #else
4     mutex_lock (&lock);
5 #endif /* _WIN32 */
```

Quellcode 2.1: Condition für eine Plattformunterscheidung

#### Solution

Um die Wartbarkeit und Verständlichkeit des eigenen Codes zu steigern, wird eine objekt-orientierte Indirection zwischen dem eigenen Code und diesen low-level Funktionen eingeführt: Wrapper Facade-Klassen. Vorteil: Der Nutzer dieser Klassen muss keine systemabhängigen Conditions verwenden, wodurch der Code ohne Mehraufwand portabel bleibt. Er muss sich nicht mit den low-level C-Funktionen rumschlagen.

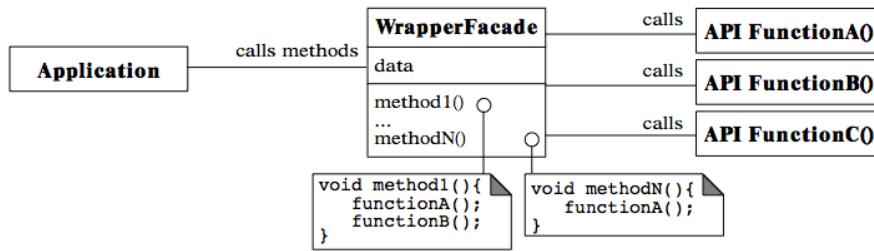


Abbildung 2.1.: Wrapper Facade Klassendiagramm

Der Entwickler der Anwendung ruft eine Methode auf der Wrapper Facade auf, welche diesen Aufruf in einen oder mehrere System-API-Aufrufe umwandelt.

## Verwendung

### 1. Java

In Java gibt es beispielsweise Swing/AWT, um GUIs zu realisieren. Diese greifen schlussendlich auf das System-API zurück. Je nach Plattform kommt dafür eine andere Implementierung zum Einsatz.

## Vorteile

- Ein einfaches, verständliches, konsistentes, robustes, objekt-orientiertes Interface für den plattformunabhängigen Zugriff auf low-level APIs. Dadurch erhöhte Wartbarkeit und Portierbarkeit.

## Nachteile

- Die Wrapper Facade stellt meistens nur den grössten gemeinsamen Nenner zur Verfügung: Also die Funktionalität, die sicher auf jeder Plattform verfügbar ist. Will man plattformspezifische Features nutzen, ist das nicht möglich. Indirektion bedeutet immer Overhead: Die Performance nimmt also ab.

## 2.2. Interceptor

The interceptor architectural pattern allows services to be added transparently to a framework and triggered automatically when certain events occur.

## Begriffe

- *Application*: Implementiert Framework
- *Interceptor*: Klasse/Objekt welche eine Schnittstelle zum Intercepten definiert
- *Concrete Interceptor*: Klasse/Objekt welche den Interceptor implementiert

- *Concrete Framework*: Instanziertes Framework (konkrete Implementation eines Frameworks)
- *Black-Box-Framework*: Framework deren Source nicht zugänglich ist / nicht modifizierbar ist

## Kontext

Frameworks welche transparent erweitert werden können.

## Problem

Frameworks können nicht voraussehen, welche Services ihre User benutzen müssen/wollen und User können insbesondere Black-Box Frameworks nicht erweitern, wenn sie nicht ursprünglich dafür gedacht waren. Frameworks müssen daher Integrationen von Dritt-Services ohne Modifikation der Architektur erlauben. Auch sollen solche Modifikationen keine existierenden Framework-Komponenten oder Applikationen des Frameworks beeinflussen.

## Lösung

Applikationen welche bestimmte Frameworks einsetzen müssen sich daher transparent beim Framework für bestimmte Services oder Events registrieren können. Durch die Implementation von bestimmten Schnittstellen, welche das Framework zur Verfügung stellt, muss die Applikation einzelne Aspekte des Frameworks kontrollieren und auf das Framework zugreifen können.

## Implementation

Das Interceptor Callback Interface ist dabei eine Implementation zu diesem Problem. Das Framework stellt dieses Interface zur Verfügung und Applikationen können concrete interceptors aufgrund dieses Interfaces implementieren. Diese Klassen registrieren sie bei einem Dispatcher für bestimmte Ereignisse und werden danach vom Framework mit context Objekten aufgerufen. Typischerweise existiert ein Dispatcher pro Interceptor.

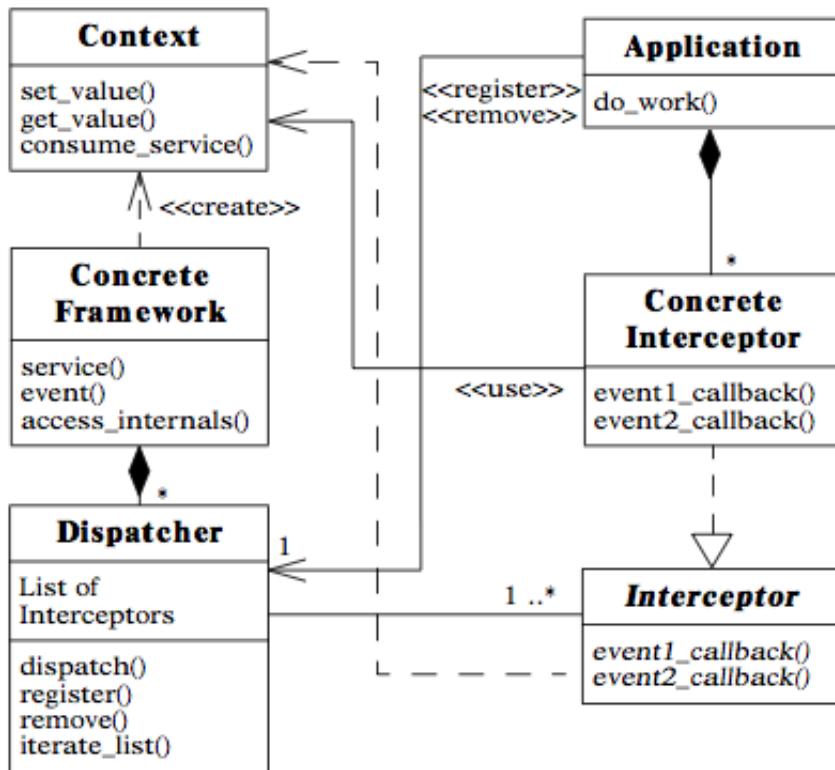


Abbildung 2.2.: Interceptor UML-Diagramm

## Varianten

- *Interceptor*  
Wird häufig auf der Server-Seite von Verteilten Systemen verwendet. Dabei wird ein Proxy (POSA1) instanziert, welcher eine Referenz auf das concrete framework bzw. den lokalen server hält. Sobald ein Client eine Anfrage macht, erhält das Proxy-Objekt diesen und führt u.U. Pre-Processing Funktionalität aus. Danach wird es dem Lokalen Server übergeben, welcher dann wiederum mittels dem Proxy dem Client antwortet.
- *Single Interceptor-per-Dispatcher*  
Diese Variante erlaubt nur die Registration einem Interceptor pro Dispatcher. Diese Restriktion kann die Implementation vereinfachen.
- *Interceptor Factory*  
Wenn das Framework die gleiche Klasse mehrere Male instanziert. Statt dass für jedes Objekt die Registration auf dem Dispatcher geschieht, werden Factories registriert, welche dann vom Framework aufgerufen werden.

- *Implicit Interceptor Registration*

Statt explizit Interceptors beim Dispatcher zu registrieren, kann das Framework auch Methoden zur Verfügung stellen, um dynamisch an spezifischen Orten oder mittels Konfigurationen (Component Configurator) diese automatisch zu registrieren.

### Verwendung

- *Component-based application servers*
- *CORBA implementation*
- *COM*
- *Web Browsers (Extensions)*
- *Reference Monitor (Security Patterns)*

### Vorteile

- Extensibility/Flexibility des Frameworks
- Separation of Concerns
- Monitoring und Kontrolle eines Frameworks unterstützen
- Layer symmetry: Durch Interceptors pro Layer können verschiedene Zusatzfunktionalitäten transparent implementiert werden
- Reusability

### Nachteile

- Architektur/Design wird komplexer
- Fehlerbehaftete Interceptors können die gesamte Applikation blockieren
- Interception cascades: Wenn ein Interceptor eine Änderung in einem Kontext-Objekt bewirkt kann das weitere Events zur Folge haben, welche weitere Interceptors triggern können

## 2.3. Proactor

The Proactor architectural pattern allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations, to achieve the performance benefits of concurrency without incurring certain of its liabilities.

## Context

An event-driven application that receives and processes multiple service requests asynchronously.

## Problem

By processing multiple requests asynchronously, the performance of event-driven applications, such as servers, can often be increased. To support asynchronous operations, the application must be able to react to completion events of the async operations, which are delivered by the operating system. Typically four aspects need to be taken into account:

- For scalability and low latency, multiple completion events should be processed simultaneously
- To maximize throughput, unnecessary context switching should be avoided.
- Extending the set of available services should require minimal effort.
- Application code should largely be shielded from the threading and synchronization mechanisms.

## Solution

Split each application service into long-duration operations and completion handlers. Long-duration operations are executed asynchronously and the completion handlers process the result of these operations as they finish. Further, decouple dispatcher and completion event demultiplexing from the application-specific processing (so the application specific code only has to provide a long-duration async operation and a completion handler).

## Structure

- *handle*: provided by operating systems to identify entities which can generate completion events (i.e. for a file or socket)
- *asynchronous operation*: application specific, async operation
- *completion handler*: application specific handler for completion of asynchronous operation
- *concrete completion handler*: the actual implementation of a completion handler
- *asynchronous operation processor*: executes async operations and queues completion events (often implemented by an OS kernel)
- *completion event queue*: filled with completion events on async operation completion

- *asynchronous event demultiplexer*: removes and returns a completion event (can block)
- *proactor*: calls demultiplexer, demultiplexes & dispatches to completion handler. Provides an event loop for the application.
- *initiator*: i.e. accepts incoming connections, and invokes async operation. Part of the application. Often also plays the role of a concrete completion handler to be able to process the result of the operation it invoked.

## Implementation

Two layers:

- Demultiplexing/dispatching infrastructure layer components: Performs generic, application-independent logic to execute async operations and demultiplexes and dispatches completion events.
- Application layer components: Defines application specific asynchronous operations and completion handlers

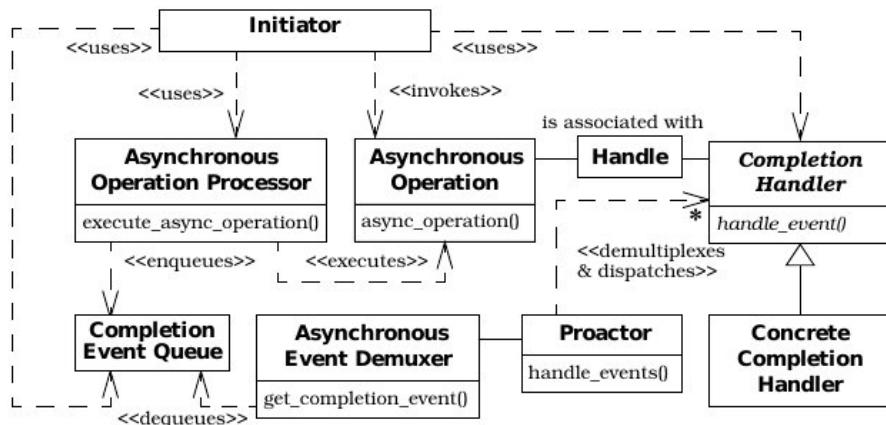


Abbildung 2.3.: proactor uml diagram

## Varianten

- *Asynchronous Completion Handlers*  
Completion handlers act as initiators, so they can also perform long-duration async operations.
- *Concurrent Asynchronous Event Demultiplexer*  
Maintain a pool of threads to which the proactor can demultiplex and dispatch completion handlers concurrently.

- *Shared Completion Handlers*

Multiple async operations can share the same completion handler. Proactor and Initiator must collaborate in this case to know which operation requires which completion handler.

- *Asynchronous Operation Processor Emulation*

Some programming environments do not export async operations to applications (i.e. the JVM). In such cases async operations can be emulated using a concurrency mechanism.

### Known uses

- Completion ports in Windows NT: OS is the async operation processor and provides the completion event queue with the completion ports.
- POSIX AIO family of async I/O: Similar to Windows NT, but preemptively async (completion can interrupt thread)
- ACE Proactor Framework
- OS device driver interrupt-handling mechanisms
- Phone call initiation via voice mail

### Benefits

- Separation of Concerns
- Portability by abstracting the async operation mechanisms and using the proactor to shield the application from non-portable completion event demultiplexing and dispatching mechanisms.
- Encapsulation of concurrency mechanisms
- Decoupling of threading from concurrency
- Performance
- Simplification of application synchronization

### Liabilities

- Restricted applicability
- Complexity of programming, debugging and testing
- Scheduling, controlling, and canceling asynchronously running operations

## 2.4. Reactor

The Reactor architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients. Das Reaktor Pattern erlaubt es, einer ereignis-gesteuerten Anwendung eine oder mehrere Client Anfragen gleichzeitig anzunehmen und auf verschiedene Serviceanbieter zu verteilen.

### Beispiel

Verteiltes Logging System: Mehrere Clients können, per TCP verbunden, ihre Lognachrichten an einen Logging Server senden, der diese entsprechend weiterverarbeitet (z.B. Schreiben in Datenbank, Console, Drucker, ...)

### Möglichkeit zur Implementation

Für jede Verbindung einen Thread. Nachteile davon sind jedoch:

- Ineffizient bzgl context switching, synchronisation und Datenbewegungen
- Braucht concurrency control im Servercode
- Nicht überall verfügbar (OS abhängig)
- Manchmal besser abgestimmt auf Anzahl CPU's statt Anzahl Verbindungen

### Problem

Event-gesteuerte Applikationen müssen viele Anfragen simultan beantworten können, auch wenn sie seriell abgearbeitet werden. Jeder Request wird durch ein indication event ausgelöst, z.B. CONNECT oder READ. Somit muss der Server zuerst die indication events an die entsprechenden Serviceimplementierungen dispatchen und demultiplexen.

Für die Implementierung muss für die Applikation folgendes gewährleistet sein, sie:

- soll nicht blockieren
- soll auf maximalen Durchsatz ausgelegt sein und somit unnötiges context switching, Datensynchronisierung oder Daten kopieren vermeiden
- soll einfach um neue Service erweitert werden können
- soll ohne komplexe multithreading und synchronisations Mechanismen auskommen

## Lösung

Synchron auf die Ankunft von indication events auf ein oder mehreren Eventquellen, z.B. sockets, warten. Für jeden Service, der eine Applikation anbietet, einen eigenen event handler anbieten, der gewisse Events von den Eventsources behandelt. Die Eventhandler registrieren sich beim Reactor, welcher einen synchronen event demultiplexer benutzt. Dieser informiert den Reaktor beim Auftreten eines Events, welcher dann synchron den event handler dispatched, um den Request abzuarbeiten.

## Struktur

### 1. *Handles*

Werden vom OS bereitgestellt und zum Identifizieren von Eventquellen gebraucht. Wenn ein indication event auftritt, wird er beim Handler in die Queue gesetzt und als „ready“ markiert.

### 2. *Synchroner event demultiplexer*

Ist eine Funktion, welche aufgerufen wird, um einen oder mehrere indication Events abzuwarten, um dann auf dem assoziierten Handle das eintreffende Event aufzurufen.

### 3. *Event handler*

Interface für concrete event handler

### 4. *Concrete Event Handler*

Implementiert spezifischen Service, welche eine Applikation bereitstellt. Ist mit einem konkreten Handler verbunden. Bsp. Logging Server: logging acceptor, welcher logging handlers erstellt und verbindet.

### 5. *Reactor*

Definiert ein Interface, bei welchem Applikationen event handler registrieren oder löschen können. Ein Reactor benutzt den synchronen event demultiplexer. Beim Auftreten eines indication events dispatched der Reactor das Event dem event handler. Im Reactor läuft ein event loop, welcher auf indication events wartet (und nicht die Applikation). Somit muss der Applikationsentwickler nur den spezifischen event handler implementieren und diesen beim Reactor registrieren. Dies wird auch Hollywood Prinzip genannt.

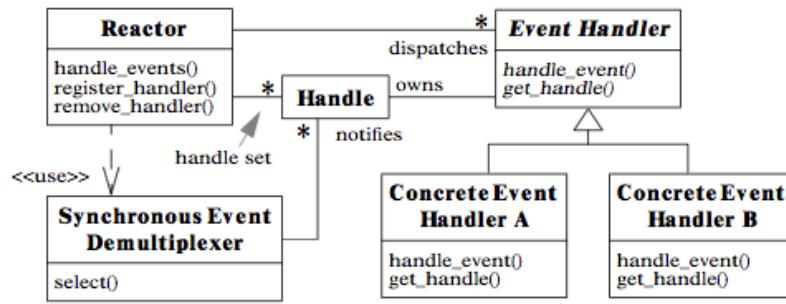


Abbildung 2.4.: Reactor UML-Diagramm

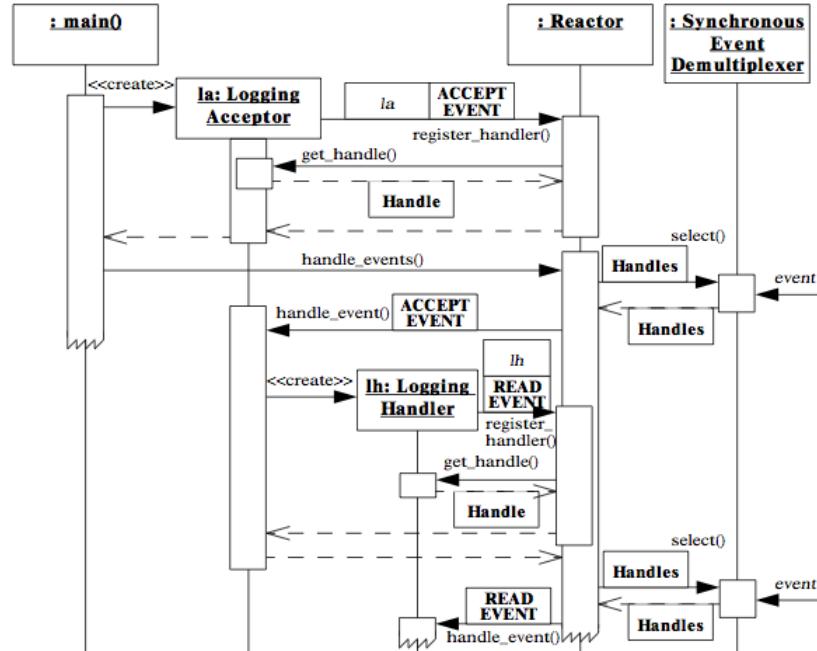


Abbildung 2.5.: Reactor Sequenzdiagramm

## Implementierung

Die Implementierung des Reactors lässt sich in zwei Layer unterteilen. Der Frameworklayer, die die applikationsunabhängige demultiplex/dispatch Infrastruktur zur Verfügung stellt und der Applikationslayer, die die konkreten event handler liefert.

## Varianten

- *Thread-Safe Reactor*
  - Mit der normalen Variante ist Thread-Safety nicht nötig, da es das dispatchen der Hook Methods implizit innerhalb des Applikations-Prozesses macht
  - Allerdings kann ein Reactor auch in multi-threaded applications benutzt werden
  - Dort können mehrere Application Threads Event Handlers registrieren und entfernen
  - u.a. Deshalb muss in diesem Fall ein Thread-Safe Reactor implementiert werden
- *Concurrent Event Handlers*
  - Event Handlers laufen in dieser Variante in einem eigenen Thread
  - Damit kann der Reactor indication events concurrently dispatchen
  - Um das zu implementieren können folgende Patterns verwendet werden: Active Object, Leader/Followers, Half-Sync/Half-Async
- Concurrent Synchronous Event Demultiplexer
  - In der standard implementation wird der sync. event demultiplexer seriell gecalled.
  - wenn dieser concurrent gecalled wird, kann eine operation auf einem Handle initiiert werden ohne dass die Operation geblockt wird

## Verwendung

- libevent
- Apache Mina
- Node.js
- Vert.x
- Python Twisted
- Ruby EventMachine

## Vorteile

- klare Trennung von Framework- und Applikationslogik
- Modularität von event-gesteuerten Anwendungen durch verschiedene event handler
- Portabilität durch Trennung von Interface und Implementierung des Reactors
- einfache Parallelität durch den synchronen event demultiplexer

## Nachteile

- setzt einen event demultiplexer voraus
- Durchsatzprobleme bei lang laufenden event handler in single Threaded Applikation, dadurch, dass der event handler den Reactor blockiert
- schwierig zu testen durch die inversion of control (Don't call us, we call you)

## Prüfungsfragen

- Was ist der Unterschied zum Proactor?
  - ...

## 2.5. Asynchronous Completion Token

The Asynchronous Completion Token design pattern allows an application to demultiplex and process efficiently the responses of asynchronous operations it invokes on services.

### Kontext

Ein Event-gesteuertes System in welchem Applikationen asynchrone Operationen auf Diensten aufrufen und nach der Komplettierung die (natürlich asynchron eintreffenden) Event Responses verarbeiten.

\*

### Terminologie

- Service (Dienst) stellt eine Funktionalität zur Verfügung, welcher asynchron ange- sprochen werden kann
- Client Initiator (oder auch Applikation, Client Applikation) ruft asynchron Ope- rationen auf einem Service auf
- Ein Asynchrones Completion Token (ACT) beinhaltet Informationen über den Completion Handler des Initiators.
  - Der Initiator gibt dieses ACT dem Dienst beim asynchronen Aufruf der Ope- ration mit.
  - Der Initiator verwendet dieses ACT nach der Response der asynchronen Ope- ration um den richtigen Handler aufzurufen

## Problem

Wenn eine Applikation mehrere asynchrone Operationen ausführt, wird jeder Dienst die Antwort an die Applikation zurückgeben. Um die Komplettierungs-Events korrekt verarbeiten zu können, muss die Applikation die Events demultiplexen und somit zum richtigen Handler weiterleiten. Drei Faktoren sind dabei wichtig und müssen gelöst werden:

- Ein Dienst könnte den ursprünglichen Kontext, in welchem die Operation gestartet wurde, nicht kennen
- Um zu entscheiden wie die Applikation einen Completion Event demultiplexen und verarbeiten soll, soll so wenig Kommunikation wie möglich zwischen Applikation und Dienst ausgetauscht werden
- Sobald die Antwort eines Services bei der Client Applikation ankommt, soll so wenig Zeit wie möglich genutzt werden um den Beendigungs-Event zum Handler weiterzureichen

## Lösung

Zusätzlich zu den normalen Informationen über den Event werden auch noch Daten mitgegeben, welche dem Initiator mitteilen, wie dieser den Event weiterverarbeiten soll.

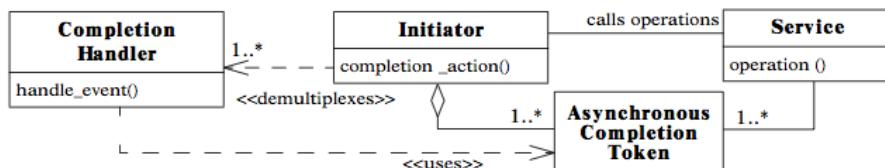


Abbildung 2.6.: Asynchronous Completion Token Klassendiagramm

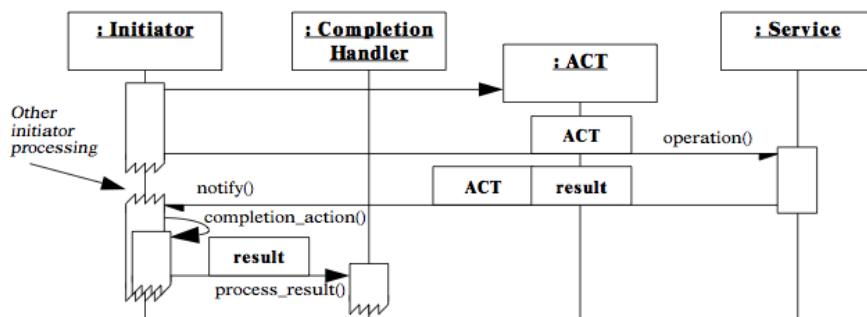


Abbildung 2.7.: Asynchronous Completion Token Sequenzdiagramm

## Varianten

- Chain of Service ACTs
  - wenn Dienste ebenfalls die Rolle eines Initiators einnehmen, um asynchrone Operationen auf weiteren Services aufzurufen
  - Chain of Services muss entscheiden, welcher Service dem ursprünglichen Initiator antwortet
  - wenn keine zusätzlichen ACTs generiert werden, kann der letzte Service direkt dem Client antworten
  - falls weitere ACTs generiert werden, muss die Sequenz korrekt eingehalten werden
- Non-Opaque ACTs
  - in manchen Implementationen ist die ACT mehr oder weniger transparent und kann verändert werden (u.a. Win32 OVERLAPPED Struktur)
- Synchronous ACTs
  - ACTs können auch für synchrone Callbacks verwendet werden
  - Somit ist das Token mehr synchron als asynchron
  - Der selbe Mechanismus zu verwenden garantiert aber eine gut strukturierte Art den Zustand eines Systems anderen Operationen/Diensten weiterzugeben

## Verwendung

- AJAX
- JSONP

## Vorteile

- Initiator muss keine komplexen Datenstrukturen unterhalten, um die Antworten des Services mit den Completion Handler zu verknüpfen. ACT kann beispielsweise ein Pointer auf einen Completion Handler sein.
- ACTs sind flexibel und müssen kein bestimmtes Interface implementieren.

## Nachteile

- Wenn Pointer als ACTs verwendet werden, kann das zu Memory Leaks führen, wenn der entsprechende Callback nie kommt.
- Initiator muss dem Service vertrauen: ACT könnte bösartig sein (gerade im Falle des Pointers)

## Prüfungsfragen

- Was muss beim Dienst beachtet werden, wenn die aufgerufene asynchrone Methode selber auch asynchron abläuft?
  - ACT muss in einer externen Datenstruktur gespeichert werden.
  - Beim synchronen Ablauf der Methode könnte es auch einfach in der Runtime gespeichert werden.

## 2.6. Acceptor Connector

The Acceptor-Connector design pattern decouples the connection and initialization of cooperating peer services in a networked system from the processing performed by the peer services after they are connected and initialized.

### Kontext

Ein Netzwerk-basiertes System, in welchem Verbindungsorientierte Protokolle zum Kommunizieren zwischen Peer Services, verbunden über Transport-Endpoints, verwendet wird.

\*

Terminologie Verbindungs-Rollen:

- *aktiv*: Dienste initialisieren aktiv Verbindungs-Anfragen
- *passiv*: Dienste akzeptieren passiv Verbindungs-Anfragen
- *aktiv/passiv*: Dienste könnten in einer Situation aktiv sein, in einer anderen passiv
- *hybrid*: Konfigurationen in welchem ein Dienst aktiv und passiv auf der gleichen Schnittstelle kombinieren.

### Problem

Bei verteilten Softwaresystemen wird oft viel Code für den Verbindungsaufbau und das Initialisieren von Services gebraucht. Dieser Code hat mit der eigentlichen Verarbeitung wenig zu tun. Damit man beispielsweise das Transportprotokoll einfach wechseln kann, will man hier eine möglichst geringe Kopplung.

### Lösung

Verbindungsauflaufbau und Initialisierung der Services wird vom Processing-Code entkoppelt. Application Services werden in Service Handlers gekapselt. Für den eigentlichen Verbindungsauflaufbau und das Initialisieren nutzt man Acceptor (wartet auf Verbindung) und Connector (baut Verbindung auf). Nach dem Verbindungsauflaufbau werden die entsprechenden Service Handlers initialisiert und ein Transport Handle wird bereitgestellt.

Die Service Handlers beinhalten dann den eigentlichen Applikationscode. Sie sind vollständig losgelöst von spezifischen Transportprotokollen etc.

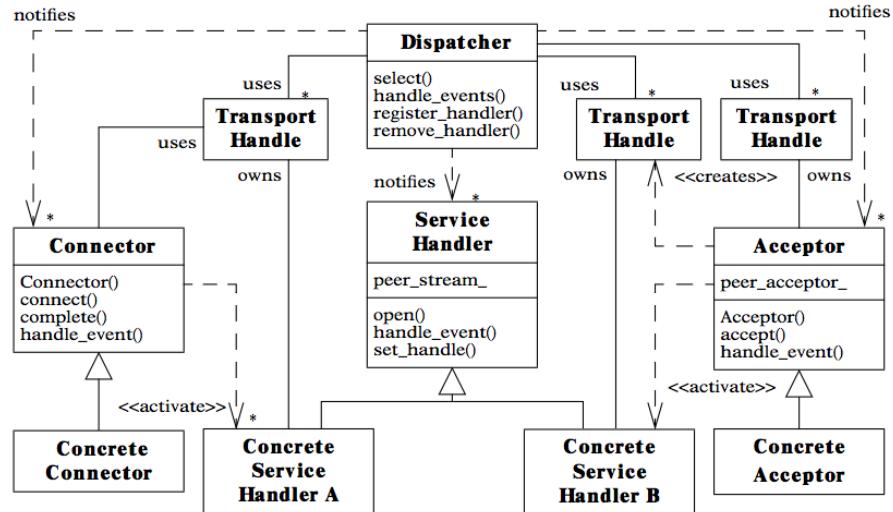


Abbildung 2.8.: Acceptor Connector Klassendiagramm

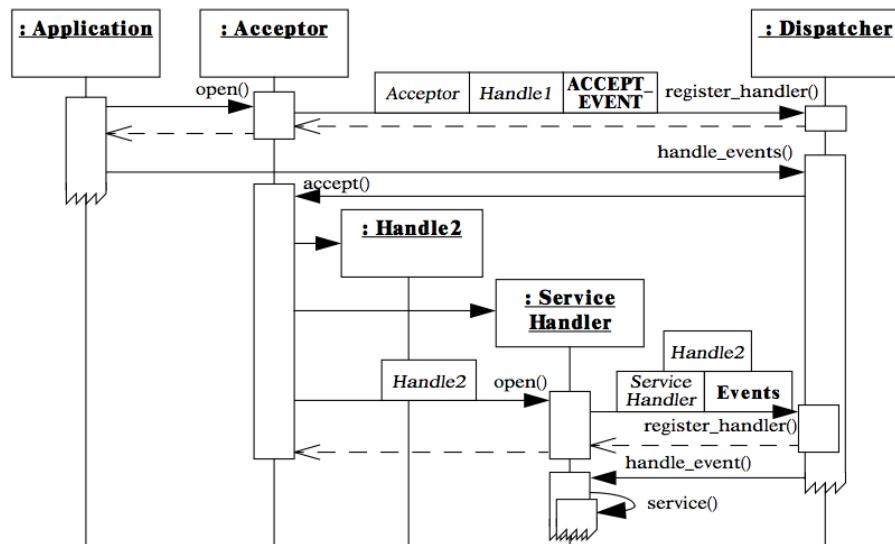


Abbildung 2.9.: Acceptor Sequenzdiagramm

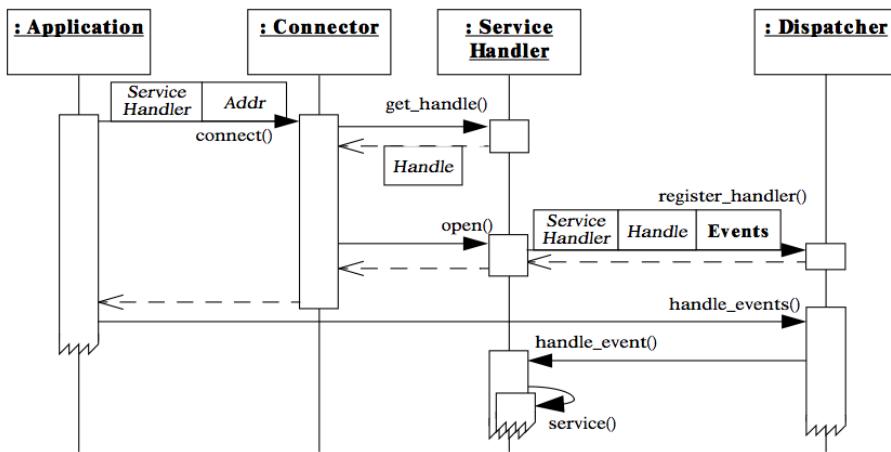


Abbildung 2.10.: Connector Sequenzdiagramm

## Verwendung

- Unix network superserver implementationen wie Inetd, Listen
- CORBA Object Request Brokers
- Web Browsers

## Vorteile

Die Geringe Kopplung führt zu:

- Austauschbarkeit, Wiederverwendbarkeit: Applikationscode nicht an spezifisches Transportprotokoll gebunden
- Portabilität: Je nach Plattform kann anderer Acceptor- und Connector-Code verwendet werden; der eigentlich Applikationscode muss nicht angepasst werden
- Robustheit: Kein low-level Gebastel im Applikationscode

## Nachteile

- Indirection: evt. Overhead
- Komplexität: für simple Applikationen ist das Pattern unter Umständen ungeeignet

## Examples

Ein Telefongespräch zwischen zwei Managern wird durch ihre jeweiligen Büroangestellten initialisiert (Acceptor / Connector). Sobald die Verbindung steht, können die Manager (Service Handlers) miteinander sprechen.

## Prüfungsfragen

- Was ist der Unterschied zwischen Acceptor und Connector?
  - ...

## 2.7. Component Configurator

The Component Configurator design pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application. Component Configurator further supports the reconfiguration of components into different application processes without having to shut down and re-start running processes.

### Kontext

Ein System/eine Applikation in welcher Komponenten so flexibel und transparent wie möglich neugestartet, suspendiert oder wieder aufgeweckt werden muss.

### Problem

Komponenten-basierte Applikationen müssen einen Mechanismus zur Verfügung stellen, um diese Komponenten in mehrere Prozesse aufteilen zu können (konfigurierbar). Die Lösung zu diesem Problem ist durch 3 Bedingungen eingeschränkt:

- Es sollte möglich sein, Implementations von Komponenten an jedem Punkt im Lifecycle der Applikation zu ersetzen
  - Modifikationen an einer Komponente dürfen nur minimale Auswirkungen auf andere Komponenten haben
  - Auch das neustarten etc. einer Komponente darf nur minimale Auswirkungen auf andere Komponenten haben
- Entwickler eines Systems wissen meist nicht im Voraus, wie eine Komponente optimalerweise in Prozesse gesplittet werden sollte
  - erste Konfigurationen von Komponenten können mit der Zeit suboptimal sein (Plattform upgrades, erhöhte Workloads etc.) und müssen neu konfiguriert werden
- Das Konfigurieren, Initialisieren und Kontrollieren von Komponenten (Administrative Tätigkeiten) müssen einfach und Komponenten-unabhängig sein

### Lösung

Komponenten-Schnittstellen von der Implementation und Applikationen von der Konfiguration der Komponenten unabhängig machen.

## Struktur

- ein Komponenten-Interface definiert eine gleichartige Schnittstelle um diese zu Konfigurieren und Administrieren
- Konkrete Komponenten implementieren dieses Interface
- Ein Komponenten-Repository verwaltet alle konfigurierten Konkreten Komponenten
- Ein *Component Configurator* verwendet das Komponenten-Repository um die neu-konfiguration von Konkreten Komponenten zu verwalten/koordinieren
- Applikationen verwenden diese Interfaces um Komponenten zu administrieren

\*

## Klassendiagramm

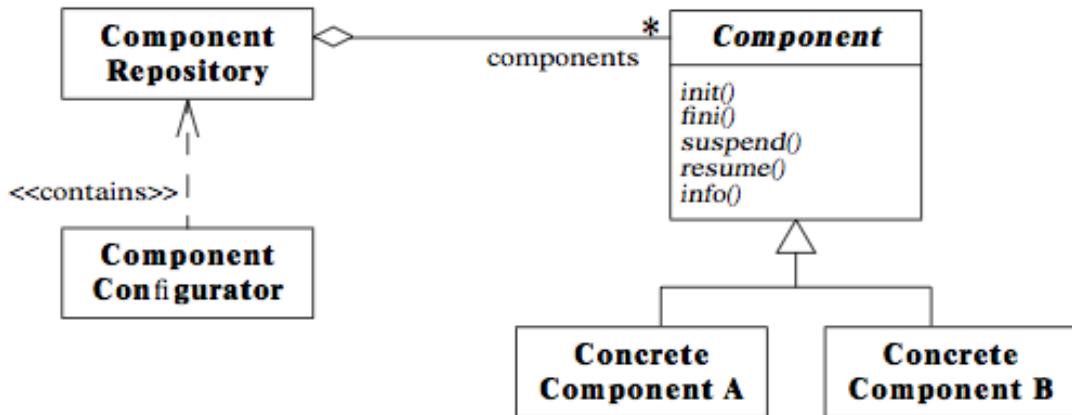


Abbildung 2.11.: component configurator classdiagram

\*

## Sequenzdiagramm

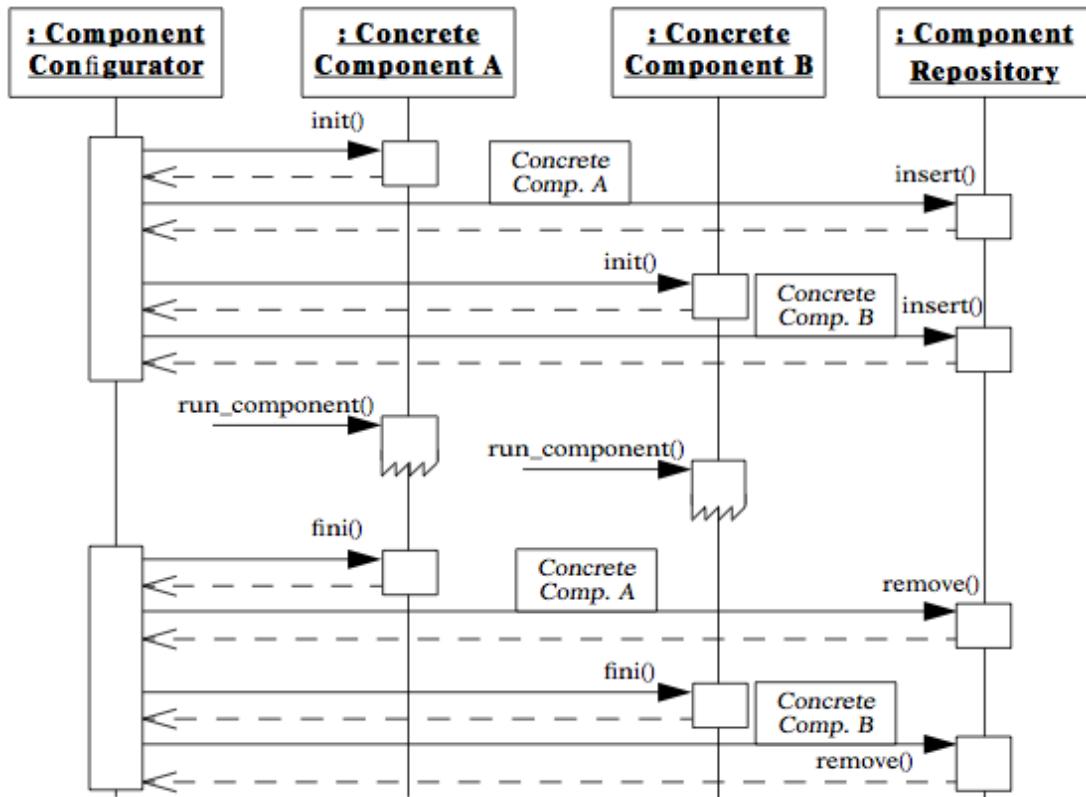


Abbildung 2.12.: component configurator sequencediagram

\*

Zustandsdiagramm einer einzelnen Konkreten Komponente

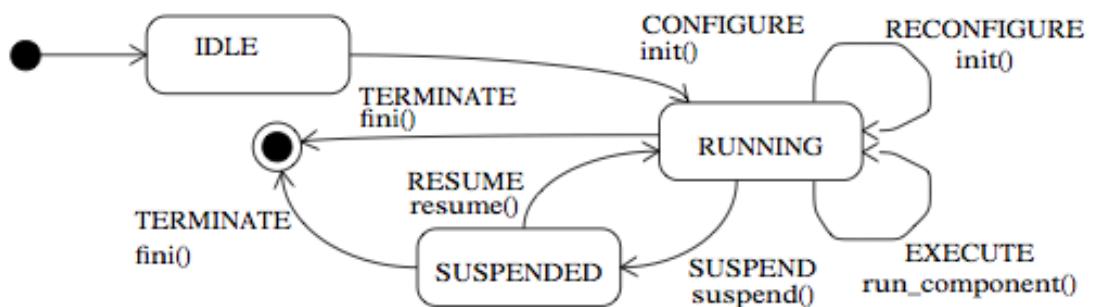


Abbildung 2.13.: component configurator statediagram

## Implementation

Das Komponente-Interface kann entweder über Inheritance oder über Message Passing implementiert werden

## Vorteile

- Uniforme Konfiguration und Administration von Komponenten
- Zentralisierte Administration
- Modularität, Testbarkeit und Wiederverwendbarkeit von Komponenten
- Dynamische Neukonfiguration
- Optimierung und Tuning während der Laufzeit möglich

## Nachteile

- Determinismus ist schwer nachzuvollziehen bis die Applikation vollständig konfiguriert ist
- Reduzierte Sicherheit und Verlässlichkeit
  - Sicherheit weil sich Betrüger als Komponenten maskieren könnten
  - Verlässlichkeit weil eine falsch konfigurierte Komponente die Ausführung der Komponente beeinträchtigen kann
    - \* Auch weil eine Komponente crashen kann und die Zustandsinformationen die sie mit anderen Komponenten teilt beschädigen kann
- Erhöhter Runtime Overhead und Infrastruktur Komplexität
- Interfaces der Komponenten könnten zu tightly coupled oder zu komplex sein, um uniform ausgeführt zu werden

## Known Uses

- Windows NT Service Control Manager
- Device Drivers

## Prüfungsfragen

### 2.8. Active Object

The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

## Problem

Ein Objekt wird von mehreren Threads/Processes benutzt, dabei soll aber eine blockierende Operation nicht alle anderen Clients blockieren.

## Lösung

Clients rufen Methoden des Objekts via Proxy auf. Die Proxy wandelt die Aufrufe in MethodRequest's um welche in eine Queue eingefügt werden welche dann von einem Scheduler abgearbeitet wird. Der Rückgabewert wird via Future-Objekt zurückgegeben.

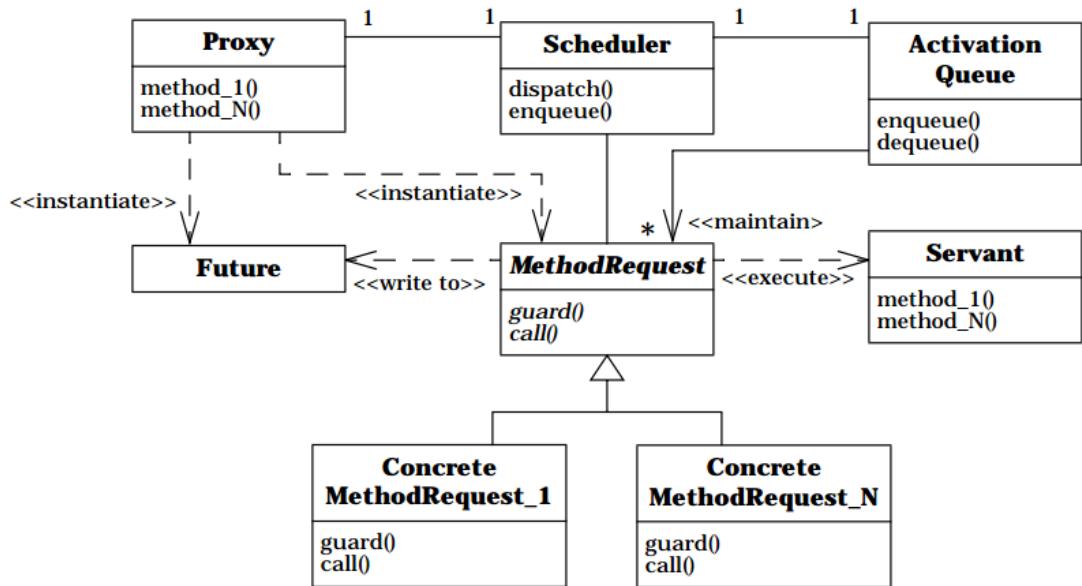


Abbildung 2.14.: active object class diagram

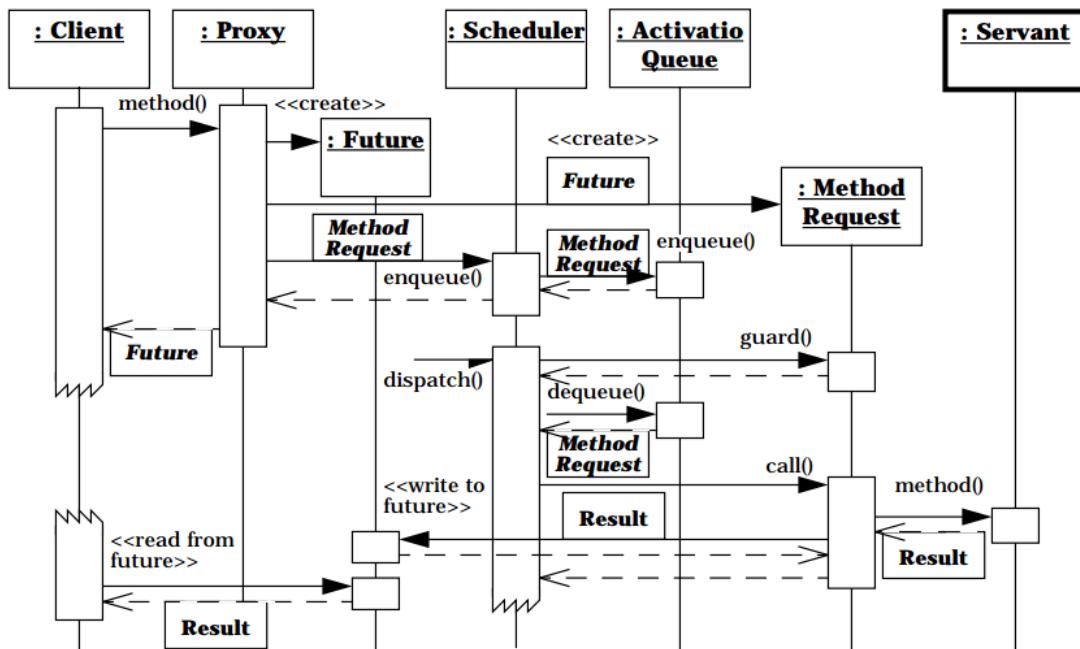


Abbildung 2.15.: active object sequence diagram

## Vorteile

- Methodenaufrufe können zum Client transparent parallelisiert werden
- ActiveObject und Proxy können z.B. auch via Netzwerk gekoppelt werden.

## Nachteile

- Erhöhte Komplexität und Implementationsaufwand
- Potentieller Performanzverlust

## Known Uses

## Prüfungsfragen

## 2.9. Monitor Object

The Monitor Object design pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to cooperatively schedule their execution sequences.

## Problem

Mehrere Threads greifen gleichzeitig auf Methoden desselben Objektes zu. Die Methoden ändern Felder des Objektes, was somit dessen Zustand ändert. Um dies zu gewährleisten, müssen folgende Probleme berücksichtigt werden:

1. Es sollten nur Interfaceaufrufe, welche die Synchronisation definieren, auf dem Objekt aufgerufen werden. Zur gleichen Zeit sollte auf dem Objekt immer nur 1 Methode ausgeführt werden.
2. Da Lowlevel Synchronisation komplex ist, soll sich das Objekt selbst um selbige kümmern.
3. Um Deadlocks vorzubeugen/zu vermeiden, müssen blockierende Threads den Lock selbstständig freigeben.
4. Bei der freiwilligen Abgabe muss das Objekt jedoch in einem stabilen Zustand sein.

## Lösung

Jedes Objekt, welches vor parallelem Zugriff geschützt werden sollte, muss einen Monitor Lock implementieren. Die Methoden, welche als synchronized deklariert sind, teilen sich einen gemeinsamen Monitor Lock. Wird dieser Monitor Lock von einer Methode beansprucht, können andere synchronized Methoden nicht zur Ausführung kommen. Die Kommunikation zwischen den Threads geschieht dann über die sogenannten Monitor Conditions (wait(), notify(), notifyAll(),...), worüber die Methoden die Ausführung selbst stoppen und wieder aufnehmen können.

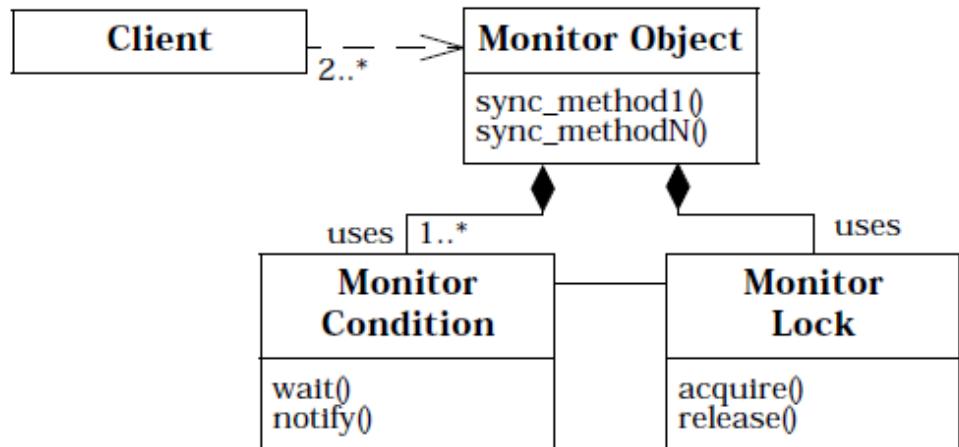


Abbildung 2.16.: UML

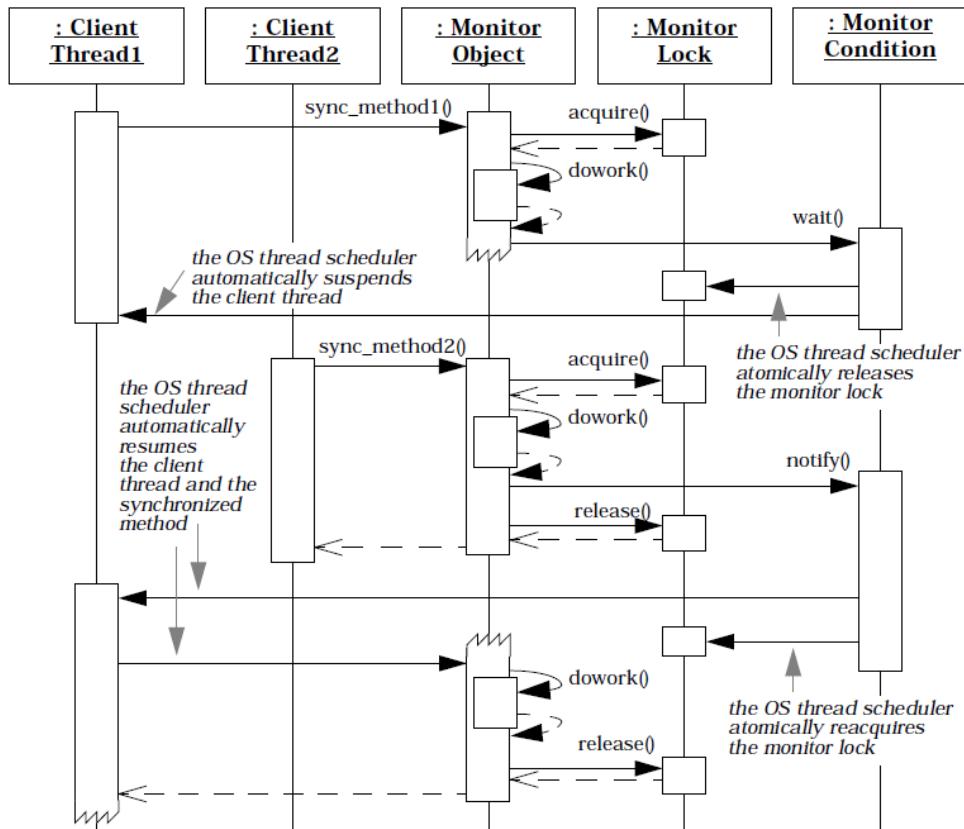


Abbildung 2.17.: SSD

## Implementation

1. Monitor Object Interface definieren
2. Von aussen zugängliche Methode definieren
3. Monitor Lock definieren (nur 1 Methode pro Objekt)
4. Monitor Condition definieren (Kommunikation zwischen Threads)

## Varianten

- Timed Synchronisation Method Invocations
  - Maximale Wartezeit kann definiert werden
- Strategized Locking

- Mehrere Monitor Locks/Condition definieren

### Vorteile

- Parallelprogrammierung (Kontrolle) wird vereinfacht
- Vereinfachung der Planung der Methodenausführungen. Mit Hilfe der Monitor Conditions (wait(..))

### Nachteile

- Skalierbarkeit wird beeinträchtigt, wenn mehrere Threads um den Monitor Lock kämpfen
- Monitor Object Synchronisation ist oft eng an die Methodenfunktionalität gebunden
- Vererbung von der Basisklasse ist nicht immer einfach, falls ein anderer Synchronisationsmechanismus zum Einsatz kommt
- Nested monitor lockout: Methoden können sich gegenseitig ausschliessen (durch Verschachtelung von Monitor Locks)

### Known Uses

- Dijkstra and Hoare-style Monitors
- Java Objects
- ACE Gateway
- Fast food restaurant

### Prüfungsfragen

- Wie kann es bei Monitor-Objects zu Deadlocks kommen? Wie viele Monitor-Objects müssen dabei mindestens involviert sein?

## 2.10. Scoped Locking

The *Scoped Locking* C++ idiom ensures that a lock is acquired when control enters a scope and released automatically when control leaves the scope, regardless of the return path from the scope

### Kontext

Shared resources welche gleichzeitig von mehreren Threads manipuliert werden

## Problem

Um eine Methode Thread-Safe zu machen wird vielfach z.B. ein Mutex innerhalb der Methode acquired. Da eine Methode aber mehrere Return-Paths haben kann (u.a. auch Exceptions, continue, break, etc.), wird dies pro Zeile Code immer wie schwieriger.

## Lösung

Erstelle eine Klasse, welche im Konstruktur ein Lock acquired und im Destruktor dieses wieder freigibt. Diese Klasse dann jeweils in Methoden/Block scopes von Kritischen Code-Teilen instantieren. Der Destruktor wird automatisch aufgerufen wenn die geschützte Methode verlassen wird. Somit ist sichergestellt, dass Locks immer wieder freigegeben werden.

## Varianten

- *Explicit Accessors:* Ein Nachteil dieser Implementation: Ein Lock wird immer nur nach dem verlassen der geschützten Methode freigegeben. Falls es gewünscht ist, explizit dieses freizugeben, müsste eine public Methode *release()* in der Guard Klasse erstellt werden. Über diese Methode kann dann explizit das Lock freigegeben werden.

## Known Uses

- Java: *synchronized* Methoden.

## Vorteile

- Increased Robustness

## Nachteile

- Potential für Deadlocks falls es rekursiv verwendet wird
- Limitationen aufgrund von Sprach-spezifischer Semantik (wenn das C-Feature *longjmp* verwendet wird, werden keine Destruktoren aufgerufen)

## 2.11. Strategized Locking

The Strategized Locking design pattern parameterizes synchronization mechanisms that protect a component's critical sections from concurrent access

## Kontext

Eine Applikation welche effizient auf verschiedenen Concurrency-Architekturen laufen muss (z.B. Single-Threaded und Multi-Threaded)

## Problem

Unterschiedliche Applikationen könnten unterschiedliche Synchronisations-Strategien (Mutex, Reader/Writer lock, Semaphore) verwenden/erfordern. Es sollte daher möglich sein, den Synchronisations-Mechanismus ohne neuschreiben der Funktionalität auszutauschen.

## Lösung

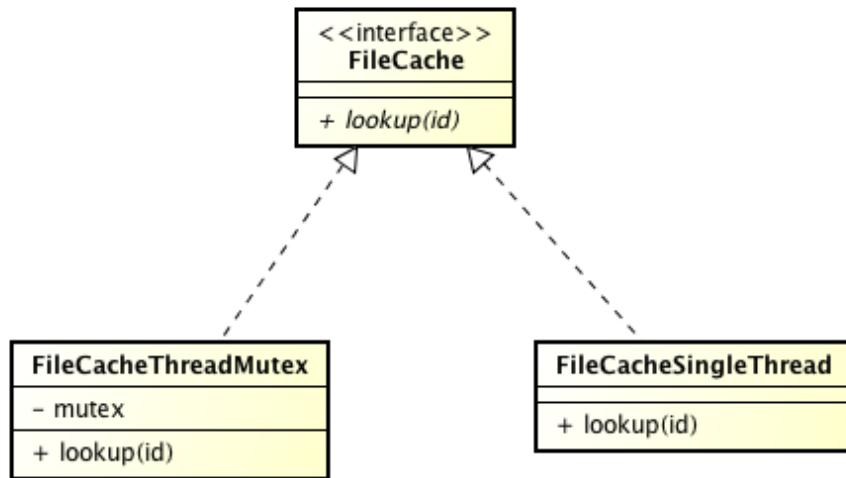


Abbildung 2.18.: Strategized Locking Class Diagram

Um z.B. eine Single-Threaded Applikation so schnell wie möglich zu machen, kann ein Null-Objekt als Lock verwendet werden.

## Vorteile

- Flexibilität und Anpassbarkeit wird erhöht
- Weniger Aufwand für die Wartung der Komponenten
- Wiederverwendbarkeit verbessert

## Nachteile

- Obtrusive Locking: Wenn templates für das Locking verwendet werden, wird die Locking Strategie dem Applikations-Code überlassen. Obwohl das Design flexibel ist, kann dies auch ein Nachteil sein, insbesondere wenn Compiler Templates nicht gut unterstützen.

- Over-Engineering: Dieses Pattern kann z.T. zuviel Flexibilität zur Verfügung stellen.

## 2.12. Thread-Safe Interface

The *Thread-Safe Interface* design pattern minimizes locking overhead and ensures that intra-component method calls do not incur 'self-deadlock' by trying to reacquire a lock that is held by the component already.

### Kontext

Komponenten in multi-threaded Applikationen, welche intra-Komponenten Methoden aufrufe benutzen.

### Problem

Multi-Threaded Komponenten haben häufig mehrere öffentliche Interface Methoden (und private Implementations-Methoden), welche den Zustand der Komponente ändern können. Um Race-Conditions zu verhindern, kann ein Lock intern für die Komponente verwendet werden, um Aufrufe solcher Methoden seriell zu machen. Falls das gemacht wird, werden folgende Punkte bei intra-komponenten-methoden Aufrufen relevant:

- Thread-Safe Komponenten müssen self-deadlock verhindern (wenn 2 Komponenten-Methoden aufgerufen werden und beide das gleiche Lock verwenden)
- Der overhead durchs Locking sollte so klein wie möglich gehalten werden

### Lösung

- Das Interface sollte das Lock machen, die eigentliche Implementation sollte erst nach dem Locken aufgerufen werden.
- Implementations-Methoden sollten nur etwas machen, wenn sie von den Interface-Methoden aufgerufen werden.

### Varianten

- Thread-Safe Facade: Kann verwendet werden, wenn der Zugriff auf ein ganzes Subsystem synchronisiert werden muss.
- Thread-Safe Wrapper Facade: Hilft den Zugriff auf eine nicht-synchronisierte Klasse zu synchronisieren

### Kown uses

- Java: `java.util.Hashtable` verwendet dies
- Security Checkpoints: Im Real-Life wird das z.B. beim Zoll verwendet: Man darf nur rein, wenn der Security Guard an der Grenze den einlass gewährt

### Vorteile

- Increased Robustness (kein self-deadlock)
- Verbesserte Performance (kein unnötiges acquiren/releasen von Locks)
- Vereinfachung der Software

### Nachteile

- Mehr Indirektionen und extra Methoden (Komplexität)
- Deadlock möglich (Thread-Safe Interface verhindert nicht einfach so die Deadlocks)
- Da Java/C++ auch class-level statt object-level access auf Methoden ermöglichen, kann es möglich sein, das falsch zu verwenden
- Overhead

### See also

- Decorator pattern

## 2.13. Half-Sync/Half-Async

The Half-Sync/Half-Async architectural pattern decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing.

### Kontext

Ein nebenläufiges (concurrent) System, welches untereinander kommunizierende synchrone und asynchrone Dienste laufen lässt.

### Problem

Asynchrones Programmieren erlaubt schnellere Programme, ist aber grundsätzlich schwieriger für den Software Entwickler. Synchrone Programme sind einfach zu verstehen, u.U aber langsamer (da Signale grundsätzlich asynchron sind). Um in einem Software-System beides zu erlauben, müssen folgende Punkte eingehalten werden:

- Entwickler, welche synchrone Software entwickeln wollen, sollen sich nicht um asynchrone Programmteile kümmern müssen. Dasselbe gilt für Entwickler, welche Asynchrone Programme entwickeln.
- Asynchrone und synchrone Services sollen untereinander kommunizieren können, ohne die Performance anderer Programmteile zu beeinträchtigen.

## Lösung

Das System in zwei Layer aufteilen (sync und async) und dazwischen ein Queue-Layer setzen, damit sie untereinander kommunizieren können.

## Struktur

- synchrones Service Layer* betreibt High-Level Dienste. Dienste im Sync. Service Layer laufen in separaten Threads/Prozessen, welche blockieren können
- asynchrones Service Layer* betreibt Low-Level Dienste, welche normalerweise aus einer oder mehreren Externen Event Sourcen entspringen. Dienste in diesem Layer dürfen nicht blockieren, sonst geht die Performance den Bach runter
- queueing Layer* stellt den Mechanismus für die Kommunikation von Diensten in den einzelnen Layers zur Verfügung. Dieser Layer ist zuständig, dass die Dienste benachrichtigt werden, wenn Nachrichten in der Message Queue sind
- external Event Sources* generieren Events, welche vom Async Service Layer empfange und verarbeitet werden. Das sind z.B. Network Interfaces, Disk Controller etc.

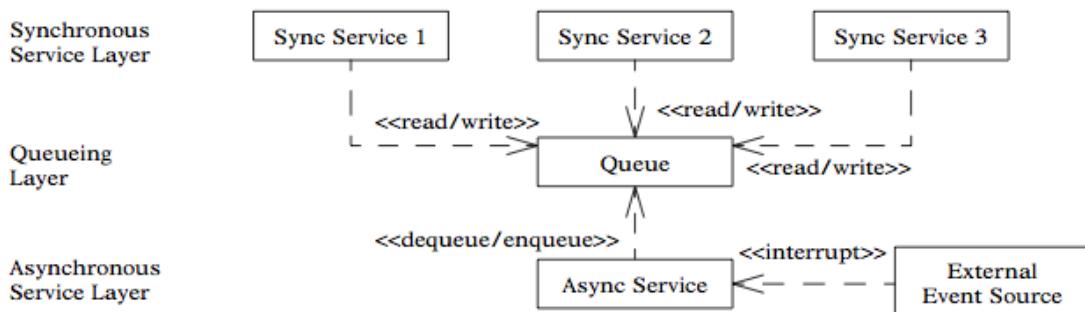


Abbildung 2.19.: Half-Sync/Half-Async Klassendiagramm

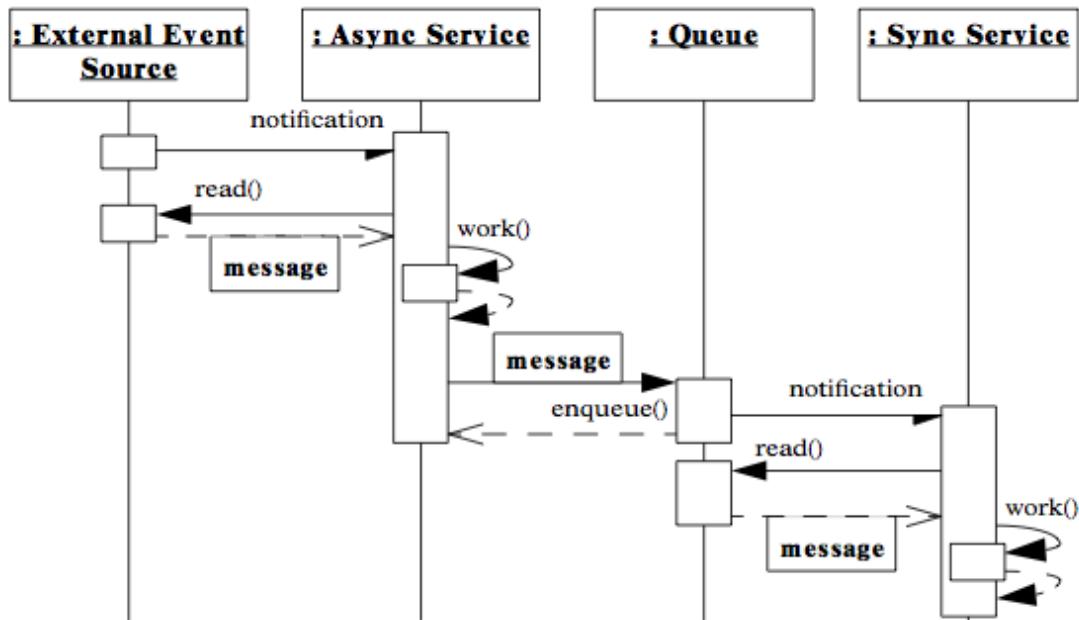


Abbildung 2.20.: Half-Sync/Half-Async Sequencediagram

- Vereinfachung und Performance: Synchrone Prozesse können einfacher implementiert werden, Asynchrone Prozesse können mit einer hohen Performance entwickelt werden
- Separation of Concerns: Synchronizations Policies in jedem Layer sind decoupled
- Zentralisierung von Inter-Layer Kommunikation

### Nachteile

- Boundary-Crossing Penalty: Für die Kontextwechsel, die notwendige Synchronisation und das Datenkopieren für die inter-layer Kommunikation (letzteres kann durch gemeinsam genutzte Speicherbereiche verhindert werden)
- Komplexität

## 2.14. Leader/Followers

The Leader/Followers architectural pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources.

## Kontext

Eine eventbasierte Applikation, in welcher mehrere Service Requests von mehreren Event-Quellen gleichzeitig auftreten und effizient von mehreren Threads abgearbeitet werden müssen.

## Problem

Multithreading wird oft verwendet, um mehrere Events nebenläufig zu verarbeiten. Es ist jedoch schwierig, das auch performant zu implementieren. Folgende Punkte müssen eingehalten werden:

- Das Demultiplexen von Event-Quelle nach Worker-Thread muss effizient gestaltet werden. Ein Thread pro Event-Quelle skaliert oft nicht effizient.
- Der Overhead, welcher durch Nebenläufigkeit generiert wird, wie Context-Switching, Synchronisation, Cache-Kohärenz-Management, sollte minimiert werden. Insbesondere Nebenläufigkeitsmodelle, welche bei Übergabe von Daten dynamisch Speicher allozieren, wirken sich negativ auf die Performance aus.
- Race-Conditions müssen verhindert werden, wenn mehrere Threads auf die gleichen Event-Quellen zugreifen.

## Lösung

Einen Thread-Pool-Mechanismus erstellen, welcher sich Event-Quellen effizient teilt, indem ein Thread nach dem anderen je einen Event synchron an einen Applikations-Service ausliefert. Es wartet immer nur ein Thread (der Leader), auf einen Event. Die anderen Threads (Follower), reihen sich in eine Queue ein und warten. Sobald der Leader einen Event detektiert, wählt er zuerst den nächsten Leader aus den Followern aus, und liefert den Event dann als Processing-Thread einem Event-Handler aus, welcher das applikations-spezifische Event-Handling im Processing-Thread durchführt. Sobald der Thread mit dem Processing fertig ist, prüft er, ob es noch einen Leader gibt. Falls nicht, wird er der Leader. Andernfalls legt er sich wieder schlafen und wartet, bis er zum neuen Leader ernannt wird.

## Struktur

- *Handles* werden vom Betriebssystem zur Identifikation der Event-Quellen zur Verfügung gestellt (i.e. ein file handle).
- Ein *Handle Set* erlaubt auf mehreren Handles auf einen Event zu warten.
- Ein *Event Handler* spezifiziert das Interface für den applikations-spezifischen Code.
- Der *Thread pool* manages die Threads welche eine Protokoll zur Koordination implementieren (Leader/Follower-Detection). Die Threads warten auf einen Event

von einem Handle aus ihrem Handle Set. Sobald ein Event auftritt, wählt der Thread den nächsten Leader aus und liefert den Event dann einem Event Handler aus.

### Klassendiagramm

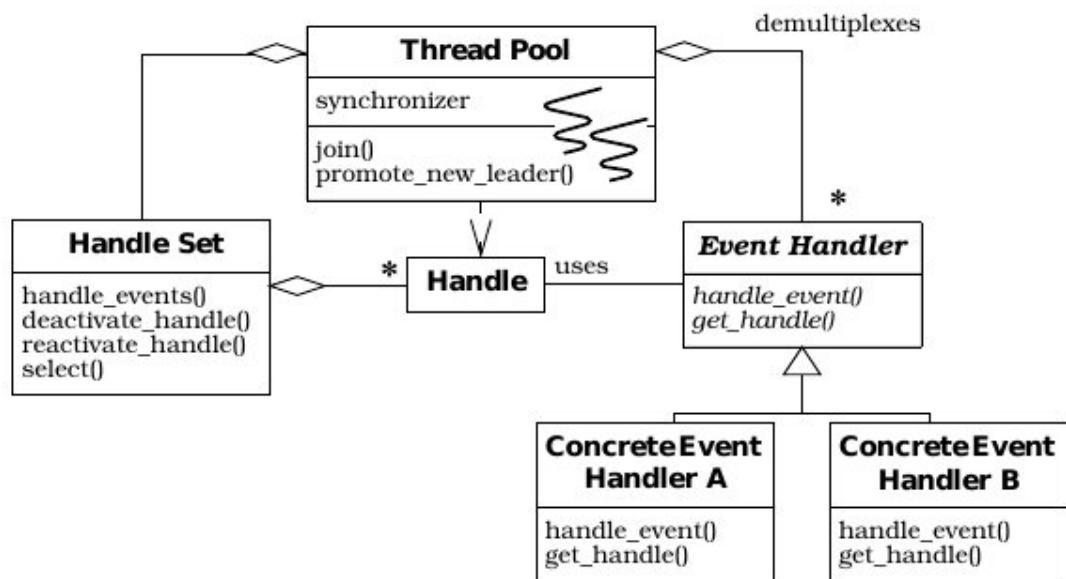


Abbildung 2.21.: Leader/Follower Structure

## Sequenzdiagramm

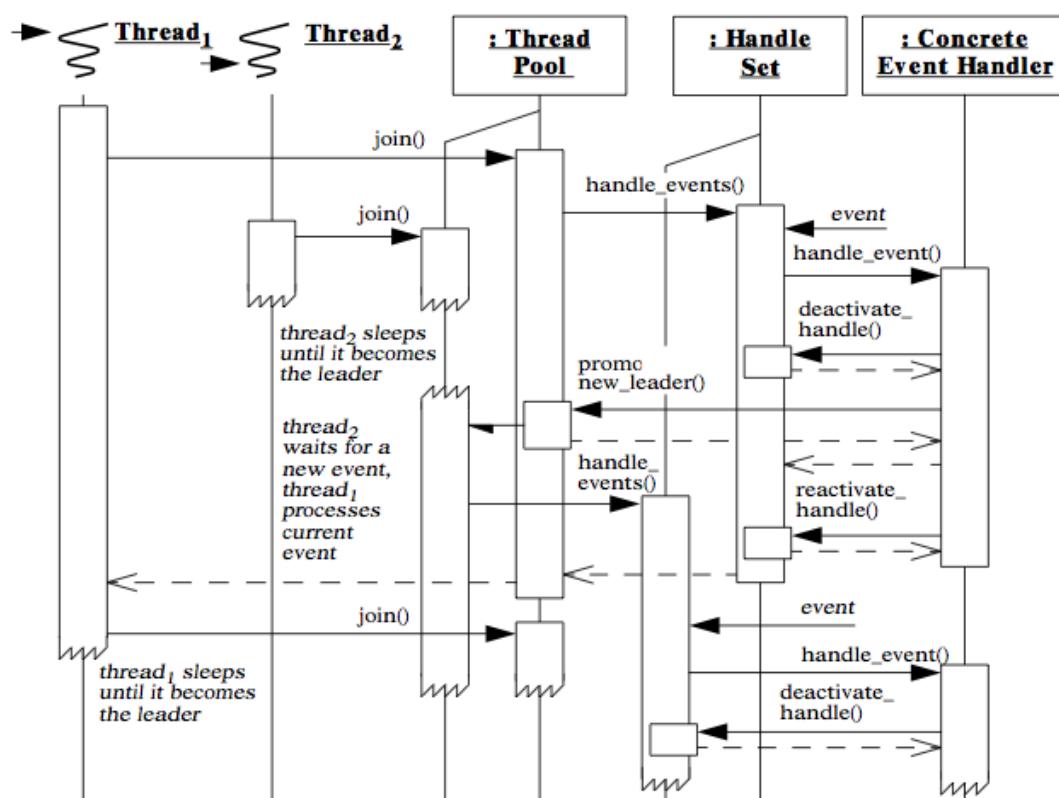


Abbildung 2.22.: Leader/Followers Sequence Diagram

## Zustandsdiagramm

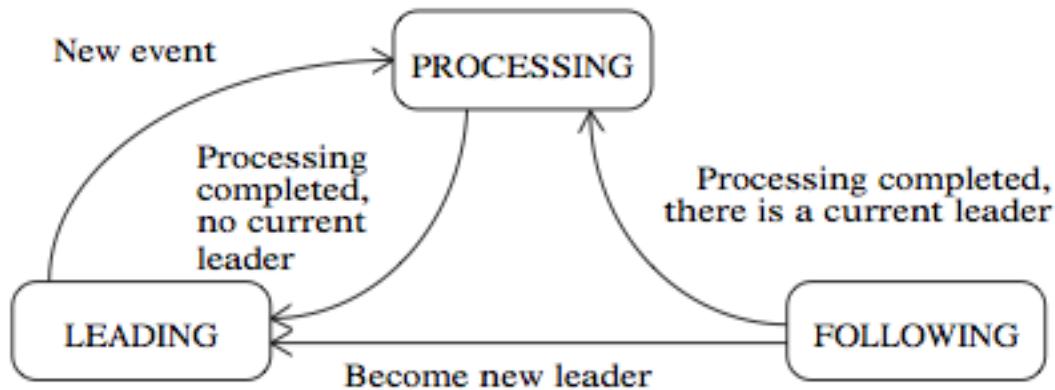


Abbildung 2.23.: Leader/Followers State Diagram

## Kapitel 3 **POSA 3**

## Kapitel 4 **Security Patterns**

### 4.1. Access Control Models

#### 4.1.1. 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:

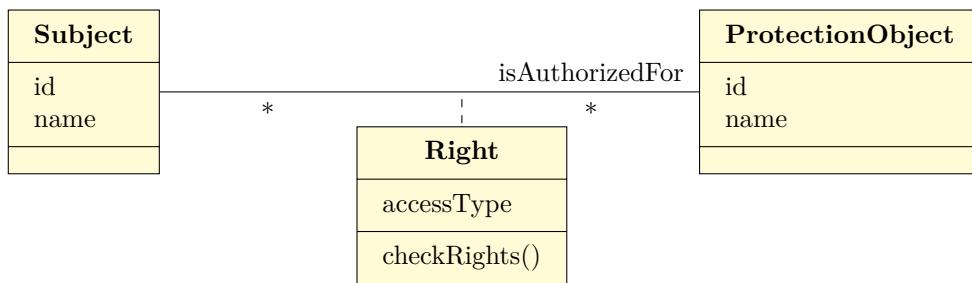


Abbildung 4.1.: Authorization Klassendiagramm

- 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)

### 4.1.2. 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 strebt sein, dass jedes Subjekt genau auf die Objekte Zugriff hat, mit welchen es täglich arbeitet (wiederum "Need to know").

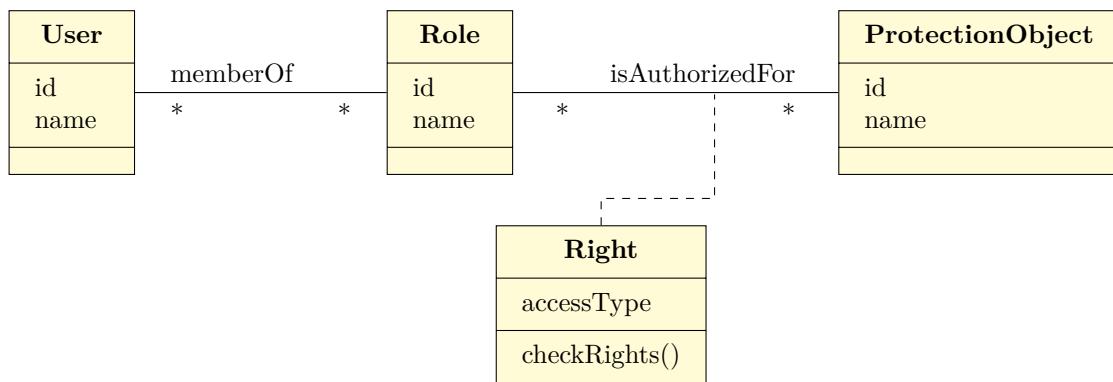


Abbildung 4.2.: Basic Role Based Access Control Klassendiagramm

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

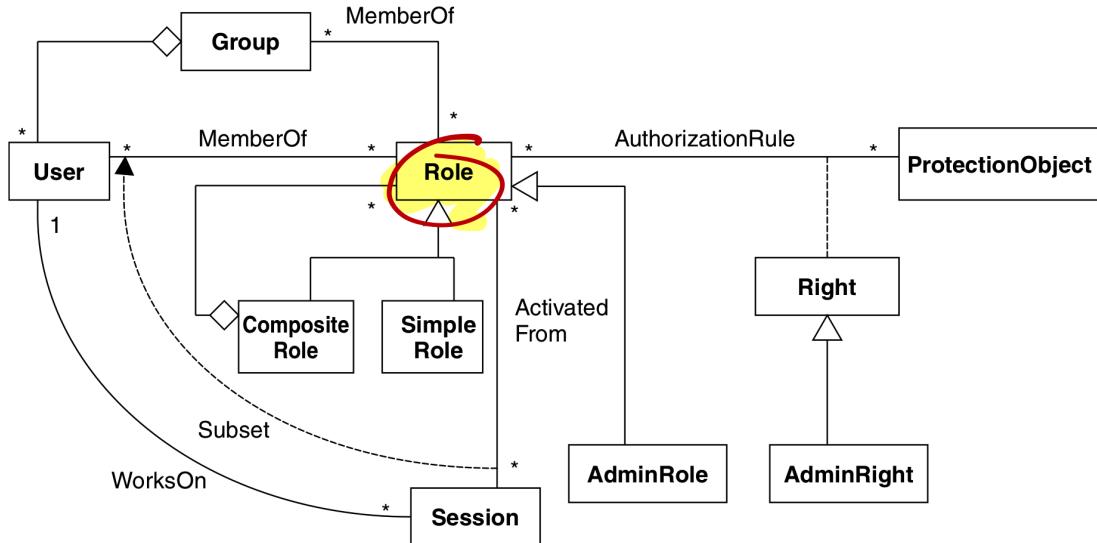


Abbildung 4.3.: RBAC mit Composite, Admins &amp; Abstract Session

### 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 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" 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 ;-)*

### 4.1.3. 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 Resourcen tiefer eingestufter Subjekte schreiben (Informationsweitergabe!)

### 3. Zugriffsmatrix

Matrix, welche Zugriffsberechtigungen von Subjekten auf Resourcen festlegt

Die Korrektheit der *Policies* wiederum wird über das Biba-Modell[wikb] (der Umgebung 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 Stabjekte dürfen nicht in Informationen höher eingestufter Subjekte schreiben

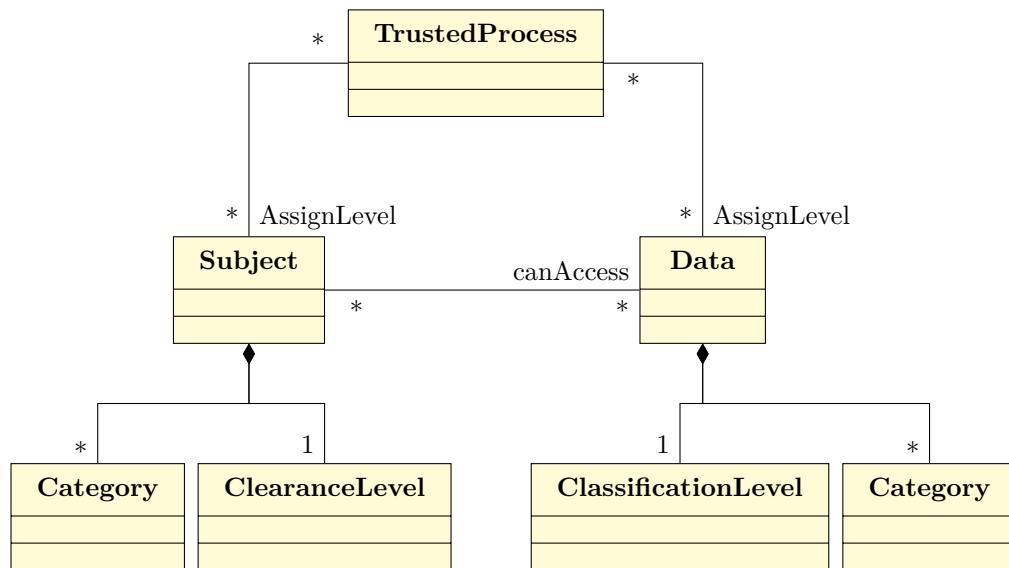


Abbildung 4.4.: Multilevel Security Klassendiagramm

## 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 tatsächlichen 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 Berechtigungsknoturen 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 4.1.2 Role Based Access Control kann mit diesem Pattern problemlos kompiliert 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)

### 4.1.4. Reference Monitor

*aka Policy Enforcement Point*

Das *Reference Monitor* Pattern beschreibt eine abstrakte Vorgehensweise, 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.

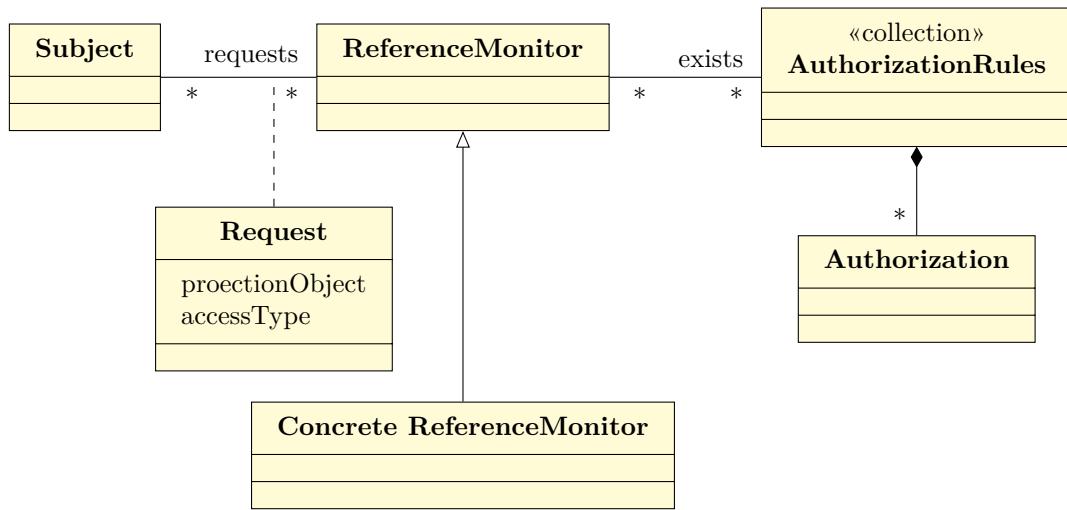


Abbildung 4.5.: Reference Monitor - Klassendiagramm

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.

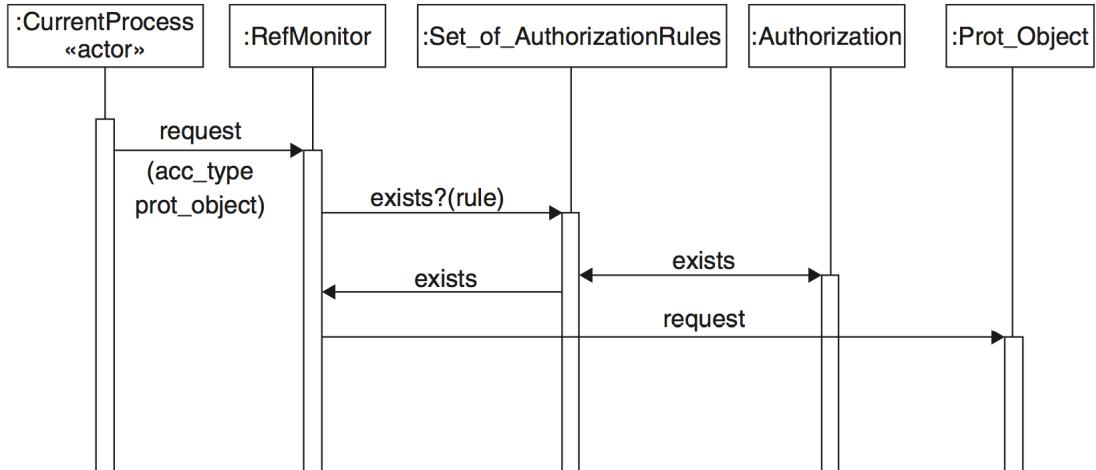


Abbildung 4.6.: Reference Monitor - Sequenzdiagramm [Sch+06]

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)

### 4.1.5. 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 4.1.2 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.

## 4.2. Identification & Authentication

### 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

---

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

- *Sessionmanagement:* Schlagwort Single-Sign-On
- *Gesicherte Systemkomponenten, "Using function":* Komponenten, welche I&A aufrufen und dessen Output verwenden (z.B. Patterns aus dem Kapitel 4.1)

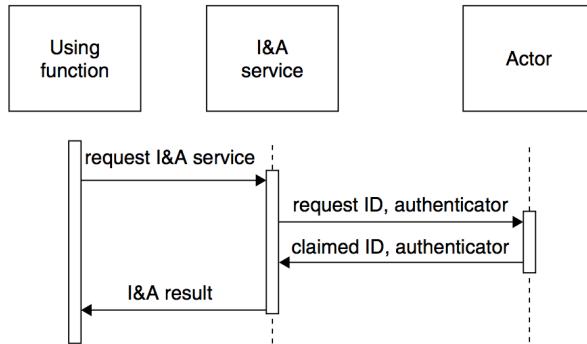


Abbildung 4.7.: Generischer Ansatz von I&A “Using functions” [Sch+06]

### 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

#### 4.2.1. 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 4.1.: 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 Wissensstände, 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 4.2.: I&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.

## 4.3. System Access Control Architecture

### 4.3.1. 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 4.3.: 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 4.4.: Access Control Requirements: Nichtfunktionale Anforderungen

## Vorteile

- Eine ausführliche Domain- und Anforderungsanalyse wird gefördert.
- Die vorliegenden Requirementsvorlagen fördern die ausführliche Auseinandersetzung mit den verschiedensten Einflüssen 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.

### 4.3.2. 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 verschiedenen 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"

#### 4.3.3. Check Point

Kontext

Problem

Lösung

Vorteile

- 

Nachteile

-

## Mögliche Prüfungsfragen

- ?

### 4.3.4. Security Session

Wurde ein Subjekt einmal identifiziert und authentifiziert, sollen die dadurch erlangten Informationen wenn möglich nicht erneut abgefragt resp. erfragt werden müssen.

Mit der *Security Session* werden Informationen zur Identität und dem Aufenthalt eines Subjektes in einem System generalisiert gespeichert. Zudem werden diese Informationen entsprechenden Systemkomponenten zugänglich gemacht.

#### Kontext

Subjekt spezifische Informationen sollen zwischen den Komponenten eines gesicherten Systems ausgetauscht werden können.

#### Problem

Subjekte haben im seltensten Fall Zugriff auf ein komplettes System, welches sie mit anderen Subjekten teilen.

Oft wird unter Verwendung von *Identification & Authentication* Patterns die Identität eines Subjektes festgestellt. Mittels den kennengelernten *Access Control Models* wird anschliessend sichergestellt, dass jedes Subjekt nur auf Funktionen oder Ressourcen zugriff hat, für welche es auch berechtigt ist.

Beim *Single Access Point* und *Check Point* wurde aufgezeigt, dass die zentralisierte Identifizierung und Authentifizierung für ein gut entworfenes System viele Vorteile mit sich bringt: Jede Systemkomponente kann sich fortan auf ihre Kernkompetenzen fokussieren und muss sich nicht noch um sicherheitsrelevante Aspekte kümmern.

Oftmals sollen Systemkomponenten in einem globalen Kontext übergreifend Informationen (bspw. den Namen eines Subjektes) ablegen und austauschen können. Wie kann nun aber sichergestellt werden, dass sich auf ein spezifisches Subjekt bezogene Informationen nicht mit denen anderer Subjekte vermischen?

Weiter sollen die Aktivitäten eines Subjektes innerhalb des Systems "als Ganzes" verfolgt werden können: Befindet sich ein Subjekt bereits im System? War es für eine gewisse Zeit inaktiv oder war es aktiv im System? Hat es das System verlassen?

#### Lösung

Es wird ein *Session* Objekt eingeführt. Das Session Objekt enthält zum einen sicherheitsrelevante Informationen (quasi seinen "Ausweis" während dem Aufenthalt im System) und bietet den Systemkomponenten zusätzlich die Möglichkeit, beliebige Informationen zu einem Subjekt abzuspeichern.

Das Session Objekt wird nach erfolgreichem Anmelden im System (optimalerweise bspw. am Check Point) initialisiert. Meistens wird es da mit gewissen Standardwerten

befüllt: Zugriffsberechtigungen um wiederholte Abfragen in der Datenbank zu vermeiden, Benutzerprofil usw.

Im Hintergrund kann ein *Manager* verwendet werden, um alle aktuellen Sessions zu überwachen. Er kann z.B. sicherstellen dass inaktive Sessions nach einer gewissen Zeit sich automatisch wieder am System frisch anmelden müssen.

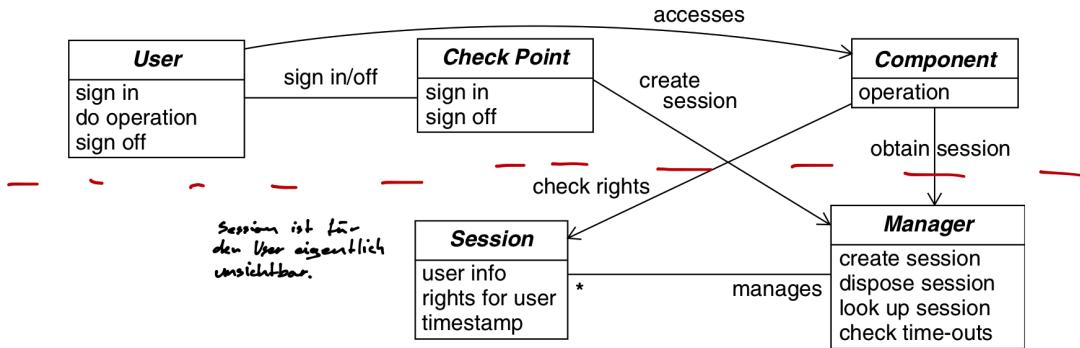


Abbildung 4.8.: Security Session: Schematischer Aufbau [Sch+06]

Damit ein Subjekte wiederkehrend mit seinem Session Objekt in Verbindung gebracht werden kann, wird eine Session ID nach aussen publiziert. Dabei ist zu beachten, dass diese für Außenstehende keine Rückschlüsse auf tatsächliche Inhalte des Session Objektes zulassen.

Das Sequenzdiagramm in Abbildung 4.9 zeigt ein Session Objekt von seiner Erstellung bis hin zu der Zerstörung sobald das Subjekt das System verlässt.

### Implementierung

1. Session Objekt einführen (klar definierte Schnittstelle zur Speicherung von Informationen (Key/Value Pairs, ...))
2. Einführung eines Session Managers zur Verwaltung der Session Objekte (Zugriff auf Session Objekte mittels Session ID's usw.)
3. Session Timeouts und die nötig werdenden Aktionen (erneut Anmelden usw.) definieren
4. Dem Subjekt ermöglichen, sich an einer Session an- und abzumelden (you don't say ;)

### Vorteile

- Klar definierter und zentraler Standort für jegliche Informationen zu einem Subjekt welches sich im System befindet

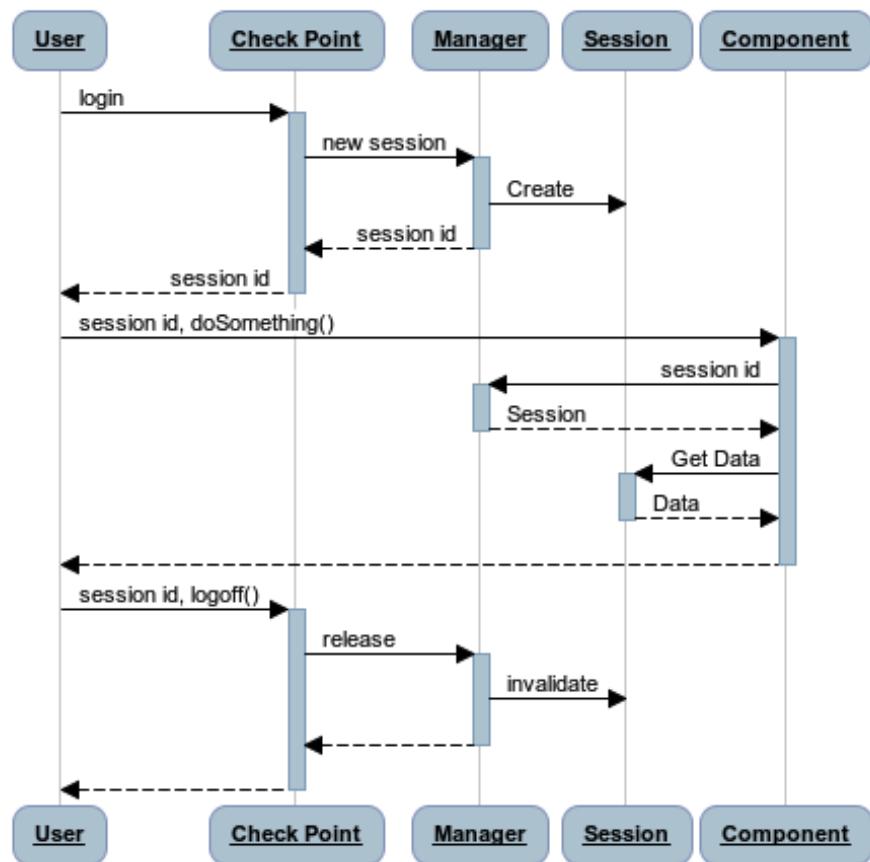


Abbildung 4.9.: Security Session: Interaktion der verschiedenen Akteure

- Komponenten können sich auf ihre Kernfunktionalitäten konzentrieren

## Nachteile

- Die Verfügbarkeit eines zentralen, globalen Objektes ermuntert möglicherweise zu unschönen Programmiertechniken
- Eine Überladung des Session Objektes mit grossen Datenmengen führt zu schlechter Systemperformance
- Schlecht gewählte Session ID's lassen möglicherweise Rückschlüsse auf den tatsächlichen Inhalt des Session Objekts

## Reallife Beispiele

- Jegliche Webapplikationen welche eine Benutzerautentifizierung benötigen greifen auf die Security Session zurück, um einen Stateful-Kontext über das eigentliche Zustandslose Medium HTTP zu erzeugen.
- Beispiel aus Bachelorarbeit *Alexandre Joly, Michael Weibel & Manuel Alabor*: Wird eine Webapplikation auf mehreren CPU-Kernen ausgeführt, kommt es durch Verwendung eines In-Memory-Storages für die Session-Objekte ggf. zu Problemen, da beim Neustart eines Kerns resp. beim Neustart der gesamten Applikation die Sessions verloren gehen. Abhilfe schafft die Auslagerung der Session Objekte wie persistente Datenbanken.

## Mögliche Prüfungsfragen

- *Wann wird eine Security Session erzeugt?*  
Nach erfolgreicher Authentifizierung eines Subjektes. Dies kann bspw. vom Check Point initiiert werden. Optimalerweise würde dieser die Session jedoch nicht selber erzeugen, sondern den Session Manager damit betrauen.

## 4.4. Firewall Architectures

### 4.4.1. Packet Filter Firewall

Werden verschiedene Computernetzwerke miteinander verbunden, entstehen unweigerlich Sicherheitsrisiken. In einem Netzverbund ist es nicht immer wünschenswert, dass Besucher aus einem (fremden) Netz Zugriff auf alle Ressourcen im eigenen Netz erhalten.

Mit der *Packet Filter Firewall* kann ein- und ausgehender IP-basierter Datenverkehr analysiert und mit entsprechenden Regeln gefiltert werden.

## Kontext

Das eigene Computernetzwerk wird mit verschiedenen anderen Netzen verbunden. Jedes dieser Netze besitzt unterschiedliche “Levels of Trust”.

Als kleinsten gemeinsamen Nenner läuft jegliche Kommunikation in diesen Netzen über das Internet Protocol (IP). Die dadurch entstehenden Datenpakete können aufgrund der Informationen in deren Header analysiert werden.

## Problem

Hosts in fremden Netzen sind potentielle Angreifer auf Ressourcen in unserem Netz. Gibt es eine Möglichkeit, diese Hosts zu erkennen und den von ihnen ausgehenden Netzwerkverkehr bestmöglich zu blockieren?

Folgende Faktoren spielen dabei eine wichtige Rolle:

- Eine komplette Abschottung des eigenen Netzes ist keine Option: Kommunikation ist ein wichtiger Bestandteil des “Daily Business”.
- Für den Benutzer soll die Sicherheitsmaßnahme transparent sein und keinen zusätzlichen Aufwand bedeuten (Login etc.)
- Der umzusetzende Mechanismus soll flexibel auf Änderungen anpassbar sein und die organisatorischen Sicherheitsrichtlinien so präzis wie möglich abbilden.
- Die Lösung soll so wenig Leistung wie möglich benötigen

## Lösung

Die *Packet Filter Firewall* analysiert ein- und ausgehenden Netzwerkverkehr. Dabei wird jedes einzelne IP-Paket auf den Inhalt in seinem Header betrachtet und mit einem Set von definierten Regeln geprüft.

Diese Regeln bestehen im Normalfall aus einer Kombination von Ports und IP-Adressen oder IP-Adress-Bereichen. Dabei kann eine Regel sowohl Zugriff gewähren als auch verbieten.

Auf diese Weise können sehr komplexe Sicherheitsdispositive gebildet und geprüft werden. Um aber auch bei komplexeren Regelsets eine optimale Performance zu erzielen ist die Reihenfolge der Regeln von grösster Bedeutung.

### Beispiel: Prüfung eingehendes IP-Paket

1. Ein fremder Host möchte auf eine Ressource im eigenen Netz zugreifen.
2. Die *Packet Filter Firewall* sucht anhand der Quell-IP-Adresse sowie des Ziel-IP-Adresse und -Ports nach einer passenden Regel
3. Wird eine passende Regel gefunden, wird das Paket entsprechend zugelassen (oder verworfen, falls die Regel dies so definiert)

4. Wird keine passende Regel gefunden, kommt eine Standard-Regel zum Zuge. Möchte man hohe Sicherheit gewährleisten, besagt diese meistens, dass das Paket verworfen werden soll.

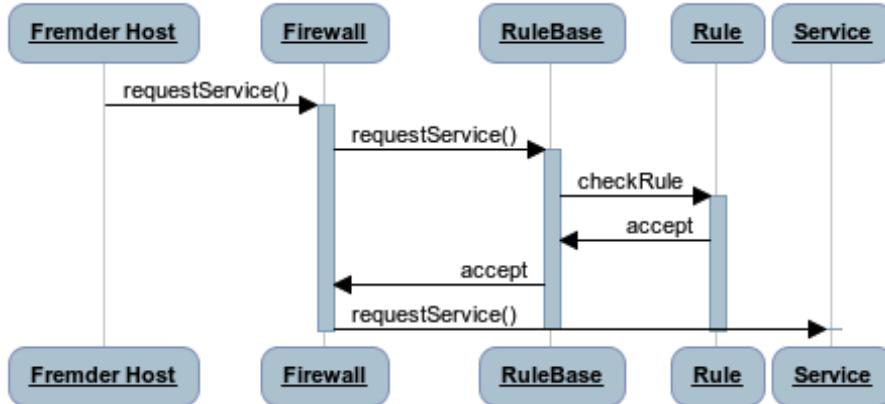


Abbildung 4.10.: Packet Filter Firewall Sequenzdiagramm

Der Akteur *RuleBase* bietet minimale Verwaltungsfunktionen (CRUD) für Firewall-Regeln.

### Vorteile

- Die Firewall filtert für den Benutzer transparent jeglichen Netzwerkverkehr
- Da die Firewall jedes IP-Paket beim empfangen oder senden einmal "in den Händen" hat, ermöglicht die Firewall ein ausführliches Logging an den Schnittstellen zwischen verschiedenen Netzwerken.
- Eine *Packet Filter Firewall* verschlingt minimale Ressourcen/Leistung. Unter anderem da lediglich die strukturierten Header-Informationen eines IP-Pakets analysiert werden.

### Nachteile

- Fälscht ein potentieller Angreifer seine IP-Adresse, kann dies die *Packet Filter Firewall* nicht erkennen und versagt in dieser Situation.
- Die Leistungsfähigkeit der Firewall ist stark von der Reihenfolge der definierten Regeln abhängig. Beispiel: Möchte man einen kompletten IP-Adress-Bereich blockieren, macht es wenig Sinn, diesen am Ende der Regelliste zu platzieren und so alle feingranularen Regeln zuerst zu prüfen.

- Die *Packet Filter Firewall* kann keine Attacken auf Layern über IP erkennen. Da nur IP-Headers analysiert werden, können im IP-Payload problemlos schädliche Befehle/schädlicher Code enthalten sein.
- Natürlich kann die Firewall nur erfolgreich Netzwerkverkehr analysieren, welcher auch über diese geleitet wird. Es gilt also sicherzustellen, dass alle Wege in das zu schützende Netz über die Firewall(s) geleitet werden (Single Access Point)

### Reallife Beispiele

- In einem Landwirtschaftsbetrieb ist jedes Tier (Netzwerkverkehr) mit einem RFID (IP-Header) ausgestattet. Will ein Tier in den Stall (zu schützendes Netz), wird es durch eine Schleuse (Firewall) geleitet. Anhand der Informationen auf dem RFID gelangt das Tier in den Stall, falls der Bauer dies vorneweg so erlaubt (Regeldefinition) hat. Darf das Tier den Stall nicht betreten, wird es wieder ins Freie geleitet.

### Mögliche Prüfungsfragen

- *Was ist ausschlaggebend für die Performance einer (Packet Filter) Firewall?*  
Die Optimierung der Reihenfolge der Firewall-Regeln.
- *Wie erreichen Sie ein Höchstmaß an Sicherheit mit der Verwendung einer (Packet Filter) Firewall?*  
Jeglicher Netzwerkverkehr muss über die Firewall geleitet werden. Weiter wird die Standardregel für Behandlung von eingehendem Netzwerkverkehr so eingestellt, dass dieser verboten wird. Anschliessend müssen nur noch Regeln für den erlaubten Verkehr erstellt werden.

#### 4.4.2. Proxy Based Firewall

Als Nachteil der “Packet Filter Firewall” wird erwähnt, dass lediglich der Inhalt des IP-Headers im Zuge der Überprüfung analysiert wird.

Die *Proxy Based Firewall* fügt der “Packet Filter Firewall” spezifische Applikations-Proxies hinzu, welche den ein- und ausgehenden Traffic überprüfen und ggf. an den internen, eigentlichen Dienst weiterleiten.

Der interne Dienst wird auf diese Weise für den externen Host komplett unsichtbar; er kommuniziert lediglich mit dem Proxy.

### Kontext

Netzwerkverkehr soll auf der Ebene des Application-Layers gefiltert werden können (vgl. “Packet Filter Firewall” tut dies lediglich auf dem Network-Layer). Auf diese Weise soll sichergestellt werden, dass keine schädlichen Befehl/schädlicher Code ins eigene Netz hinein gelangt resp. aus dem eigenen Netz heraus gesendet werden kann (Würmer, Trojaner etc.).

## Problem

Wie kann die “Packet Filter Firewall” so erweitert werden, dass nicht nur der IP-Header zur Filterung von Netzwerkverkehr verwendet werden kann? Wie kann auch der IP-Payload in die Filterung miteinbezogen werden?

Ergänzend zu den für die Packet Filter Firewall definierten Forces kommen folgende ergänzend hinzu:

- In unserem Netzwerk werden verschiedenste Dienste angeboten. Entsprechend Umfangreich muss auch das Wissen der Firewall über die jeweiligen Dienste sein.

## Lösung

Die Firewall stellt für jeden zu schützenden Dienst einen Proxy zur Verfügung. Will ein fremder Host mit einem Dienst kommunizieren, kommuniziert er lediglich mit dem entsprechenden Proxy.

Aufgrund von definierten Regeln analysiert der Proxy den ein- oder ausgehenden Verkehr. Dabei bleibt es ihm frei überlassen ob er diesen weiterleiten, blockieren oder gar modifizieren will.

## Vorteile

- Die *Proxy Based Firewall* kann Netzwerkverkehr auf Applikationsebene filtern. Sie kann dabei gezielt auf applikationsspezifische Eigenheiten eingehen und die Kommunikation ggf. sogar verändern.

## Nachteile

- Für jeden Dienst wird eine konkrete Proxyimplementierung benötigt.
- Das Betreiben der Proxies sowie die genauere Analyse des kompletten IP-Pakets führt zu höheren Kosten sowie tendenziell höherem Performance Overhead.
- Erhöhte Komplexität aufgrund der zusätzlichen Sicherungsebene.

## Reallife Beispiele

- NAT - Network Address Translation

## Mögliche Prüfungsfragen

- Nennen Sie konkrete Anwendungen für die *Proxy Based Firewall*. Zugang zu bestimmten Internetseiten blockieren (HTTP Proxy), “Network Address Translation” um die interne Netzwerkstruktur zu verschleiern. Idee: Telnet-Proxy welcher gewisse Kommandos nicht zulässt.

#### 4.4.3. Stateful Firewall

Mit der Proxy Based Firewall wurde bereits eine Erweiterung der einfachen Packet Filter Firewall vorgestellt.

Die Stateful Firewall erweitert die Proxy Based Firewall. Dabei erhält sie die Möglichkeit, die verarbeitete Kommunikation nicht nur Paketweise zu bewerten und zu klassifizieren, sondern diese auch mit bereits vergangenen Kommunikationsvorgängen in Zusammenhang zu bringen.

#### Kontext

Um bessere Sicherheit bspw. gegen Denial of Service [wikc] zu gewährleisten, soll eine zustandslose Firewall die Möglichkeit erhalten, die untersuchten Pakete in einen höheren Zusammenhang bringen zu können.

#### Problem

Wie können eingehende Pakete nicht nur einzeln kontrolliert (vgl. Packet Filter Firewall) sondern auch miteinander in Verbindung gebracht werden?

Wie kann dieser Sachverhalt weiter dazu verwendet werden, eine höhere Sicherheit zu gewährleisten?

#### Lösung

Stellt ein Client eine Verbindung zur Firewall her, wird diese Verbindung in einer Liste/Tabelle zwischengespeichert und als *geöffnet* markiert.

Dies ermöglicht das optimierte Überprüfen (oder eben gerade nicht-Überprüfen) von weiteren eingehenden Paketen des selben Clients.

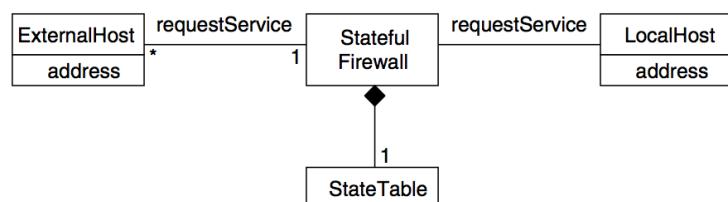


Abbildung 4.11.: Stateful Firewall: Schematischer Aufbau

### Implementierung: Handling eines Requests

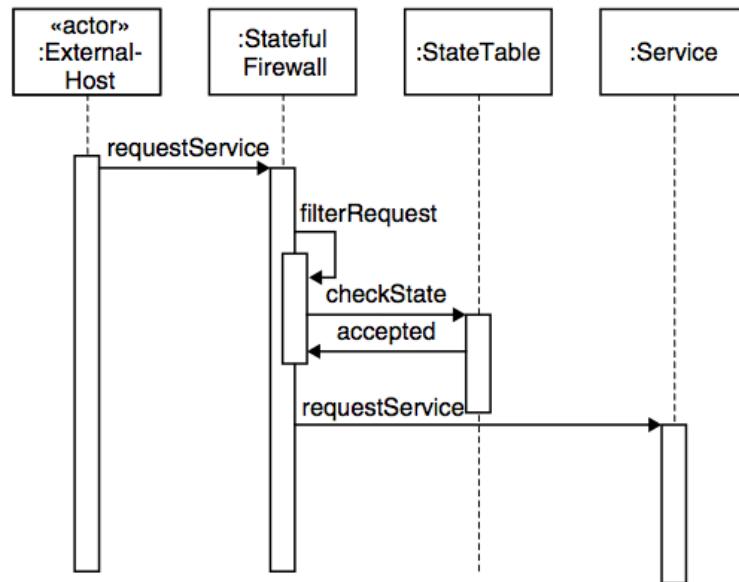


Abbildung 4.12.: Stateful Firewall: Ablauf [Sch+06]

1. Client (External Host) greift auf das System zu
2. Die Stateful Firewall prüft, ob die Verbindung zum Client bereits in der State Table vorhanden ist. Sollte dies der Fall sein, wird der Request weitergeleitet.
3. Existiert die Verbindung noch nicht in der State Table, wird der Request gem. der Packet Filter Firewall geprüft. Soll der Request zugelassen werden, wird die Verbindung zum Client in die State Table eingetragen und der Request anschliessend weitergeleitet.

Neben einer Kombination mit einer Packet Filter Firewall ist auch eine Verwendung der Stateful Firewall mit der Proxy Based Firewall problemlos umsetzbar.

### Vorteile

- In der einfachen Packet Filter Firewall-Kombination ist die Implementierung relativ kostengünstig und bietet einen guten Schutz
- Die Effizienz im Bezug auf den Sicherheitsaspekt der einfacheren Firewall Patterns kann durch die neuen Zustandsinformationen gesteigert werden
- Für neue Attacktypen müssen lediglich neue Vergleichsalgorithmen/Regeln implementiert werden

## Nachteile

- Mit dem Wissen über die Existenz einer State Table kann theoretisch eine Attacke speziell zur Überlastung der Firewall geplant werden
- Es können nur Attacken erkannt werden, für welche auch entsprechende Erkennungsalgorithmen vorhanden sind (Firmwareupgrades nötig?)

## Mögliche Prüfungsfragen

- *Ist eine Stateful Firewall ein eigenständiges Pattern?*  
Nein, es erweitert mindestens das Packet Filter Firewall Pattern.

## 4.5. Secure Internet Applications

### 4.5.1. Information Obscurity

Grundsätzlich ist anzunehmen dass jedes System irgendwann einer Attacke nachgibt. Das Information Obscurity Pattern stellt sicher, dass sensible Daten auch innerhalb des geschützten Systems bei einem möglichen ungewollten Zugriff weiter geschützt sind.

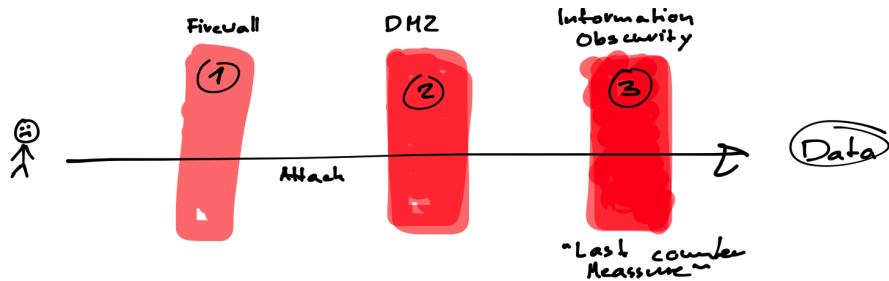


Abbildung 4.13.: Information Obscurity als letzte Sicherheitsmaßnahme

### Kontext

Ein Internet Server-System (Webserver, Applikationsserver, Datenbankbackend etc.) tauscht zwischen den einzelnen Komponenten Informationen aus. Das System an sich ist bereits nach Aussen geschützt.

### Problem

Wie kann sichergestellt werden, dass sensitive Informationen welche zwischen den einzelnen Systemkomponenten ausgetauscht wird und evtl. innerhalb von diesen abgelegt ist beim Zugriff von Unbefugten weiterhin geschützt sind?

- Es gilt abzuwägen welche Informationen überhaupt besonders geschützt werden sollen. Nicht jede Information ist hoch sensitiv und rechtfertigt die nötige Leistung zur Ver- und Entschlüsselung.

- Zu einer Ver- und Entschlüsselung sind entsprechende kryptographische Schlüssel notwendig. Diese gilt es wiederum entsprechend zu sichern.

## Lösung

Nachdem alle vorhandenen Informationsarten klassifiziert wurden, werden die als sensitiv bewerteten mittels passender Verschlüsselungsmechanismen zusätzlich gesichert.

Folgende Komponenten sind beim Information Obscurity Pattern beteiligt:

- Ein *Key* zur Ver- und Entschlüsselung der Informationen
- Ein *Key Store* welcher die sichere Aufbewahrung und Herausgabe der Keys gewährleistet
- Eine *Kryptographiekomponente* welche die eigentliche Arbeit übernimmt

Nicht zu vergessen ist natürlich die eigentliche Anwendungskomponente welche über die Kryptographiekomponente auf die verschlüsselten Informationen zugreift. Weiter sollte der Key Store über eine *Protected Location* verfügen, in welcher er die Keys vor unbefugtem Zugriff geschützt (USB-Stick etc.) aufbewahren kann.

## Ergänzungen

- Es ist nicht immer nötig, dass verschlüsselte Informationen entschlüsselt werden müssen, um auf deren Inhalt schliessen zu können.

Beispiel: Passworthashes werden mit dem Hash der Benutzereingabe verglichen

- Jeder Schutz für sensitive Daten ist zwecklos, wenn bspw. der Webserver über einen Cache verfügt welcher alle gerenderten HTML-Informationen unverschlüsselt zwischenspeichert. Es gilt also jede Komponenten im System einem genauen Audit zu unterziehen.
- Der Sicherung von Konfigurationsinformationen (Keys) sollte besondere Aufmerksamkeit geschenkt werden: Oft können diese ohne es zu bemerken kopiert werden und somit die Information Obscurity Massnahmen nutzlos machen.
- Information Obscurity muss nicht zwingend mit einer Verschlüsselung im eigentlichen Sinne zusammenhängen: Bspw. kann man durch generische Bezeichnungen für Server (Sv1, Sv2 usw. statt Dataserver, Keyserver usw.) bereits dem potentiellen Eindringling bereits einen Stein in den Weg legen und dem eigenen Sicherheitsteam einen Zeitvorteil verschaffen.
- Wie so oft ist auch bei Information Obscurity die Balance zwischen Nutzen, Kosten und Performance zu finden.

## Vorteile

- Sollte ein Angreifer in das gesicherte System gelangen, bietet Information Obscurity ein weitere Schicht an zusätzlicher Sicherheit für sensitive Informationen
- Da im optimalen Fall nicht alle Informationen mittels Information Obscurity geschützt werden, halten sich Performanceeinbussen in vertretbaren Grenzen.

## Nachteile

- Die Performance kann tendenziell leiden wenn zu viele Informationen durch einen Ver- und Entschlüsselungsprozess geschleust werden müssen
- Die Komplexität des Systems erhöht sich, was wiederum Auswirkungen auf Wartbarkeit usw. hat.
- Komponenten welche von der Information Obscurity betroffen sind werden tendenziell aufwändiger und teurer in der Entwicklung.

## Reallife Beispiele

- Verschiedenste OpenSource Projekte legen keine Klartext-Passwort in der Benutzerdatenbank ab sondern lediglich einen Hash (z.B. MD5) von diesem. Beim Login wird vom eingegebenen Passwort ebenfalls ein Hash erstellt und mit dem in der Datenbank hinterlegten verglichen.
- Beim Einbruch ins PlayStation Network von Sony (Online Gameing Platform für die Sony PlayStation Konsolen) wurden zig tausende Kreditkarteninformationen abgerufen und anschliessend auf dem Schwarzmarkt verkauft. Entsprechende Information Obscurity Massnahmen für eben diese Informationen hätten evtl. grösseren (Image-) Schaden eingrenzen können.

## Mögliche Prüfungsfragen

- *Welche Informationen sollten durch Information Obscurity geschützt werden?*  
Dies kann nicht generell beantwortet werden. Jedes System, jede Situation bedarf einer spezifischen Analyse und Klassifizierung der vorliegenden Informationen. Grundsätzlich sind aber für ein Unternehmen geschäftskritische Daten (seien diese operationeller oder aber auch rufbezogener Natur) tangiert.

### 4.5.2. Secure Channels

Kommunikation über Netzwerke/das Internet können und werden abgefangen und gelesen. Gibt es eine Möglichkeit, diese Netze zu verwenden und trotzdem eine sichere Punkt zu Punkt Verbindung zu gewährleisten?

## Kontext

Das System soll Informationen zu Clients in einem Netzwerk/dem Internet senden und von diesen empfangen können. Dabei sollen sensitive Informationen verschlüsselt resp. für fremde Augen abgeschirmt ausgetauscht werden können. Sensitive Informationen haben dabei einen geringen Anteil an der gesamten ausgetauschten Kommunikation.

## Problem

Wie können sensitive Informationen auf einem öffentlichen Netzwerk gesichert übertragen werden?

Wichtige Faktoren sind dabei:

- Die meiste Kommunikation bedarf keiner Sicherheitsmassnahmen. Sensitive Informationen müssen jedoch gesichert übertragen werden können, sobald sie das/die gesicherten Systeme verlassen.
- Verschlüsselung von Daten benötigt mehr Leistung
- Die verschlüsselte Kommunikation soll auch mit Unbekannten Partnern möglich sein; es soll dementsprechend nicht notwendig sein, spezialisierte Software oder Hardware zu installieren (Client und/oder Server)

## Lösung

Sollen sensitive Informationen ausgetauscht werden ist ein Secure Channel, ein sicherer Kanal zu erstellen, welcher die entsprechenden Informationen verschlüsselt.

Für “normal” klassifizierte Informationen soll weiterhin der standardmässige Kommunikationskanal verwendet werden.

Dabei hängen die einzelnen Komponenten wie folgend beschrieben voneinander ab:

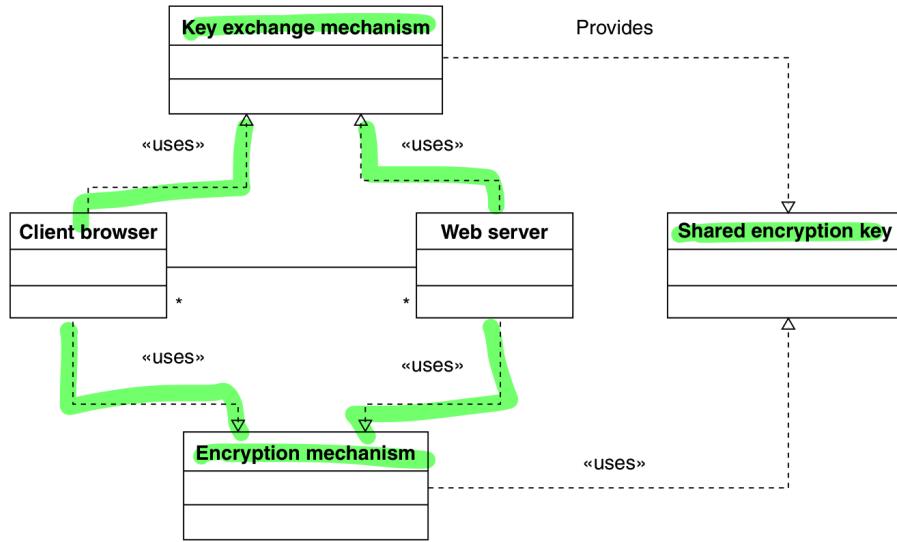


Abbildung 4.14.: Komponenten des Secure Channels Patterns [Sch+06]

- Der *Web server* stellt die eigentlichen Informationen bereit und kann mit dem *Client browser* über einen universellen *Key exchange mechanism* Schlüssel zur gesicherten Kommunikation aushandeln.
- Der *Client browser* verfügt ebenfalls über den universellen *Key exchange mechanism* über welchen er mit dem *Web server* das Setup einer gesicherten Verbindung durchführen kann.
- Sowohl der *Client browser* als auch der *Web server* können auf einen *Encryption mechanism* zugreifen, welcher sie befähigt, mittels dem ausgehandelten *Shared encryption key* einen Secure Channel einzurichten.

### Beispiel: Secure Socket Layer (SSL)

Ein heute täglich verwendetes Beispiel bietet SSL. Alle gängigen Browser und Web Server Implementation ermöglichen den verschlüsselten Datenaustausch via dem Secure Socket Layer Protokolls.

Beim Erstellen eines Secure Channels wird dabei zuerst immer ein sogenannter *Session Key* ausgetauscht, welcher zur symmetrischen Verschlüsselung von Nachrichten verwendet werden kann:

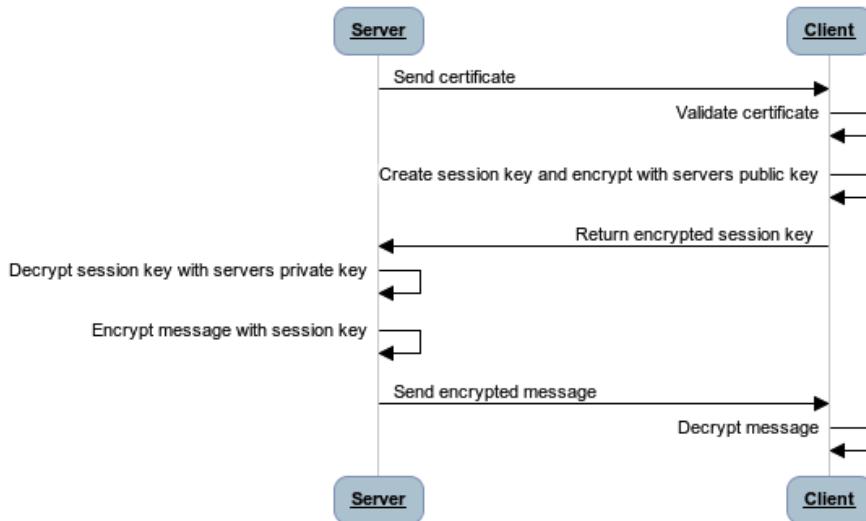


Abbildung 4.15.: Aushandeln eines Session Keys zur sicheren Kommunikation via SSL

Dabei fällt auf, dass zur Ermittlung des Session Keys eine asymmetrische Verschlüsselung zur Anwendung kommt. Diese benötigt mehr Leistung und kommt aus diesem Grund anschliessend nicht mehr zur Verwendung.

Der Session Key kann mitten in einer aktiven Verbindung ausgewechselt werden. So erschwert man potentiellen Angreifern zusätzlich das dechiffrieren der Nachrichten.

## Ergänzungen

- Die Verwendung eines Load Balancers und mehreren Web Servern erschwert die Verwendung eines Secure Channels: Nach dem Aufbauen einer Verbindung zum einen Web Server muss der Load Balancer nicht gezwungener massen bei einem nächsten Request wieder auf den selben Web Server weiterleiten.  
Um dieses Problem zu umgehen kann ein Load Balancer eine Verbindung an einen spezifischen Web Server „pinnen“, solange die entsprechende SSL aktiv ist.

## Vorteile

- Die Sicherheit von übertragenen Informationen ist gewährleistet. Sie können auf dem Weg zu ihrem eigentlichen Ziel nicht gelesen werden.
- Es ist keine spezifische Software/Hardware notwendig; SSL wird von allen aktuellen Browsern unterstützt.
- Durch den Key exchange mechanism ist es möglich, dass sich eigentlich unbekannte Partner einen Secure Channel aufbauen können.
- Normale, ungesicherte Kommunikation wird nicht beeinträchtigt.

## Nachteile

- Die Verwendung eines Secure Channels benötigt an beiden Enden mehr Leistung.
- Verschiedene Massnahmen um ein System skalierbar zu machen (Load Balancing) verkomplizieren die Verwendung eines Secure Channels
- Erhöhte Wartungskosten, evtl. sogar höhere Anschaffungskosten für Serverhardware um zusätzliche Leistung stellen zu können.

## Mögliche Prüfungsfragen

- *Warum verwendet nur der Key exchange mechanism eine asynchrone Verschlüsselung?*

Asynchrone Kryptographie Methoden sind aufwändiger zu berechnen. Aus diesem Grund wird lediglich der Session Key für den Secure Channel über diese sicherere Verschlüsselungsmethode übertragen.

Der Session Key wird während dem Aufrechterhalten des Secure Channels beliebig ausgetauscht um so ebenfalls eine hohe Sicherheit zu gewährleisten.

### 4.5.3. Protection Reverse Proxy

Einen Web- oder Applikationsserver “direkt” im Internet zu platzieren ist mehr als unvorsichtig. Natürlich kann man den Server in einer DMZ hinter einer entsprechenden Firewall platzieren, was ihn auf dem Netzwerklayer vor Angriffen schützen kann. In Verbindung mit einem “Protection Reverse Proxy” kann der Server aber zusätzlich auch auf dem Applicationlayer geschützt werden.

## Kontext

Ein System, welches HTTP/HTTPS zur Kommunikation verwendet soll geschützt werden.

## Problem

Der Einsatz einer Packet Filter Firewall kann einen Web- oder Applikationsserver auf Ebene des Netzwerklayers vor Attacken schützen. Kann dieser nun aber auch auf einer höheren Ebene, nämlich dem Applikationslayer vor schädlichen Kommandos etc. geschützt werden?

Das Absichern eines kompletten Webservers (“Hardening”) kann mit grossem Aufwand und nötigem KnowHow verbunden sein, welches evtl. nicht vorhanden ist oder mit zu grossen Kosten verbunden ist. Das Verändern der Konfiguration oder das Einspielen von Patches für einen Webserver stellen ein zusätzliches Sicherheitsrisiko dar und bedeutet so erneut Aufwand für das Hardening. Entsprechend müssen Integrationstests und Sicherheitsaudits bei jeder strukturellen Änderung wiederholt werden.

## Lösung

Mittels einem *Protection Reverse Proxy* wird der eigentliche Web- oder Applikationsserver von der Umgebung abgeschirmt.

Zwei Packet Filter Firewall stellen sicher, dass keine ungewollte Kommunikation zum eigentlichen Server gelangen kann.

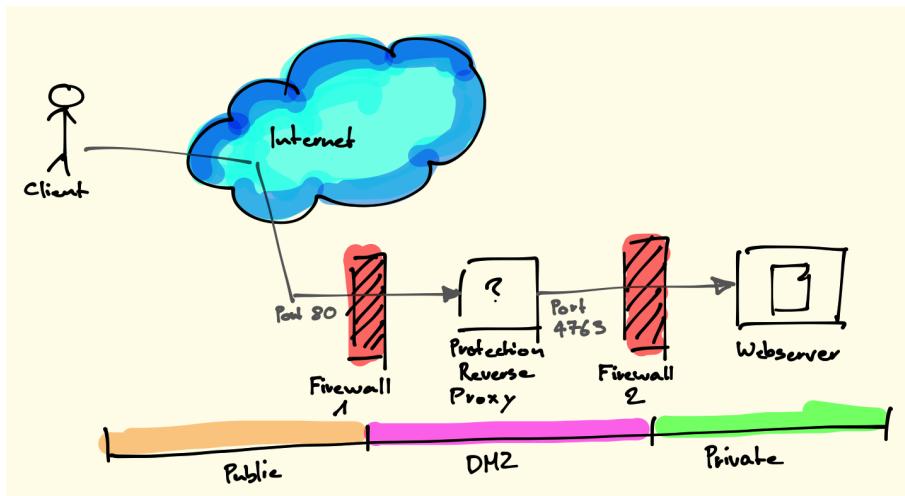


Abbildung 4.16.: Struktureller Aufbau Protection Reverse Proxy

## Ablauf

Jede eingehende Verbindung wird vom *Protection Reverse Proxy* geprüft und ggf. an den eigentlichen Server weitergeleitet. Sowohl beim Weiterleiten an den eigenen Server als auch beim senden der Antwort an den Client können mit dem *Protection Reverse Proxy* Veränderungen an Header-Fields vorgenommen werden. Das Schreiben von Logs sollte optimalerweise nach dem Versenden der Response geschehen.

Verweigert der Proxy das Weiterleiten eines Requests, ist es von der Sicherheitsrichtlinien abhängig, ob neben dem Senden eines HTTP Statuscodes auch gleich die TCP/IP Verbindung zum Client komplett geschlossen werden soll.

## Vorteile

- Web- und Applikationsserver sind für Angreifer nicht mehr direkt erreichbar.
- Web- und Applikationsserver können einfacher konfiguriert werden, da diverse Sicherheitsaspekte vernachlässigt werden können. Lediglich der spezialisierte *Protection Reverse Proxy* verfügt über direkte Berührungspunkte mit dem Internet.

## Nachteile

- Die Verwendung von Blacklist-Filtering suggeriert evtl. eine falsche Sicherheit: Diese Listen verfügen lediglich über bekannte Angriffsmuster.
- Verwendet man Whitelist-Filtering, hat man zwar eine grössere Sicherheit, ist aber anfälliger auf infrastrukturelle Veränderungen im eigenen System. Entsprechende Integrationstests sind notwendig und kostenintensiv.
- Der *Protection Reverse Proxy* ist eine zusätzliche aktive Komponente im Netzwerk. Dementsprechend leidet die Antwort- und Reaktionszeit.
- Clients verfügen über keine End-To-End-Verbindung mit dem eigentlichen Web- oder Applikationsserver mehr. Dies kann als positiv bewertet werden, im Falle von HTTPS oder allgemeinem Session-Handling als zusätzlicher Komplexitätsfaktor angesehen werden.
- Fällt der Proxy aus, ist tendenziell das gesamte System nicht mehr erreichbar: Neuer Single Point of Failure.
- Hoher Aufwand für Konfiguration und Unterhalt dank erhöhter Komplexität.

## Mögliche Prüfungsfragen

- *Ist ein Protection Reverse Proxy nur für Webserver, also das HTTP denkbar?*  
Nein. Wenn entsprechende Software vorhanden, sind *Protection Reverse Proxies* auch auf anderen Protokollen problemlos machbar. Bspw. könnte ein SMTP *Protection Reverse Proxy* zur Filterung von unerlaubten Befehlen verwendet werden.

### 4.5.4. Integration Reverse Proxy

Mit dem Protection Reverse Proxy können Web- und Applikationsserver bereits vom eigentlichen Berührungsplatz mit dem Internet separiert werden. Durch die Erweiterung des Konzepts zum *Integration Reverse Proxy* kann das ganze System nun auch unabhängig von physischer und logischer Struktur bereitgestellt und angesprochen werden.

#### Kontext

Ein System, bestehend aus mehreren Web- und Applikationsservern soll transparent angesprochen werden können.

#### Problem

Verschiedene Web- und Applikationsserver bilden ein Gesamtsystem. Im einfachsten Fall ist jeder Server über ein eindeutige URL erreichbar (bspw. "shop.business.com", "info.business.com" usw.). Auf diese Weise wird die interne Systemstruktur unweigerlich nach Aussen sichtbar gemacht. Zudem kann der Ausfall einzelner Komponenten zu

grösseren Problemen führen, da bspw. ein Load Balancing nicht ohne weiteres machbar ist.

Wie kann ein solcher Verbund von Servern für den Aussenstehenden transparent verfügbar gemacht werden, wenn folgende Faktoren eine Rolle spielen?

- Die Implementation auf einem einzigen physischen Server ist keine Option (Fehlertoleranz etc.)
- Die grundlegende Netzwerkstruktur (Hosts etc.) soll nicht nach Aussen getragen werden
- Einzelne Systemkomponenten sollen problemlos untereinander kommunizieren können (keine Hardlinks mit IP's o.Ä.)
- Ergänzung, Austausch und bis zu einem gewissen Mass Entfernung von Komponenten soll andere Komponenten nicht beeinflussen
- Load Balancing auf mehrere Komponenten soll möglich sein
- Optimalerweise muss lediglich ein einziges SSL Zertifikat angeschafft werden (Kosten)

## Lösung

Analog zum Protection Reverse Proxy wird ein *Integration Reverse Proxy* ins System integriert.

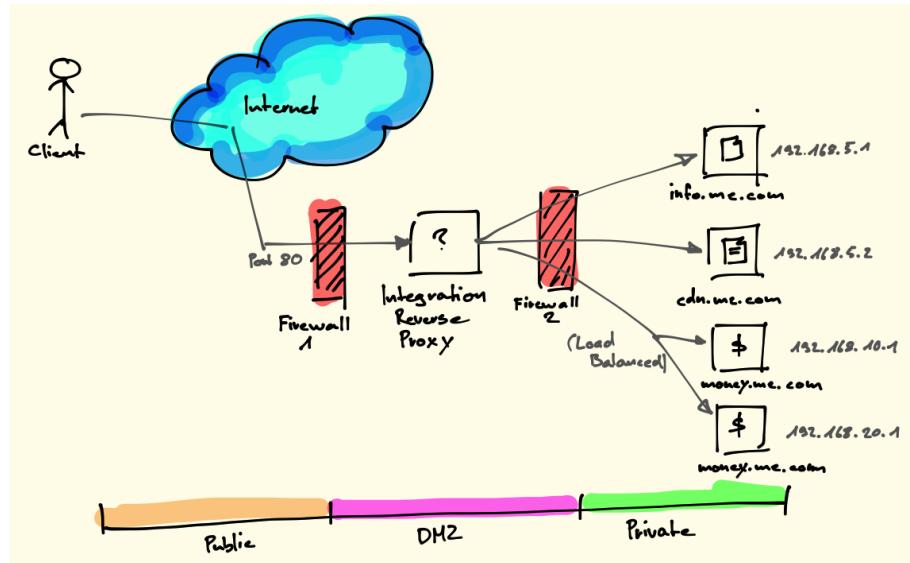


Abbildung 4.17.: Struktureller Aufbau Integration Reverse Proxy

Neben der einfachen Überprüfung und Weiterleitung von Requests wird nun selektiv entschieden, an welchen Server im eigenen Netz der jeweilige Request weitergeleitet werden soll.

## Vorteile

- Lediglich ein einziger Server/Host ist von Aussen erreichbar
- Die interne Netzwerkstruktur wird nicht preisgegeben
- Einzelne Komponenten können problemlos ausgetauscht werden oder auch nur temporär durch andere ersetzt werden: Alles eine Frage der Konfiguration des *Integration Reverse Proxy*
- Ein zentralisiertes Logging über verschiedene Systemkomponenten wird ermöglicht
- Nur ein einziges, teures SSL Zertifikat. Statt mehrere. Yay! ;)

## Nachteile

Neben den vom Protection Reverse Proxy bekannten Nachteile kommen folgende hinzu:

- Die max. Anzahl von Netzwerk-Verbindungen ist begrenzt! Diese Limitierung kann umgangen werden, indem bspw. bereits beim DNS-Server ein Load Balancing umgesetzt wird, welches nicht immer auf den selben *Integration Reverse Proxy* verweist.
- Testing-Aufwand steigt, da höhere Komplexität im Gesamtsystem

## Reallife Beispiele

- Viele Ruby- oder Node.JS-Applikationen (Liste nicht abschliessend ;)) verfügen über einen eigenen Webserver. Möchte man diese über einen bestehenden Webserver (z.B. nginx oder Apache) verfügbar machen, kommt ein entsprechendes *Reverse Proxy*-Modul zum Einsatz. Zwar laufen diese Applikationen im einfachsten Falle dann auf dem selben Host und ohne zusätzlich dazwischengeschaltete Firewall, die Technik an sich bleibt jedoch die gleiche wie beim voll ausgebauten Pattern.

### 4.5.5. Front Door

Nachdem der Integration Reverse Proxy den Zugriff auf verteilte (HTTP-)Ressourcen vereinfacht hat, soll nun das *Front Door* Pattern einen gesamtheitlichen Authentication-Mechanismus sowie Session-Kontext über alles Systemkomponenten ermöglichen.

#### Kontext

In einem System bestehend aus mehreren Webapplikationen soll ein Integration Reverse Proxy die Authentication aller Benutzer generalisiert übernehmen.

#### Problem

Die verschiedenen Komponenten im System verwalten im Normalfall jeweils eigene Datenbanken mit Benutzerinformationen. Wie kann für ein solches System ein Single Sign On umgesetzt werden, wobei auf folgende Faktoren zu achten ist:

- Bestehende Komponenten sollen weiterhin auch ihre eigenen Datenbanken verwenden können
- Es soll nicht für jede Komponente erneut nach einem Passwort o.Ä. gefragt werden (wobei dies je nach Sicherheitsrichtlinie möglich sein soll)
- Die einzelnen Komponenten sollen vom Authentication-Mechanismus unabhängig sein (austauschbar)
- Verschiedene Berechtigungen sollen abgebildet/implementiert werden können
- Neben dem *Single Sign On* soll auch ein *Single Sign Off* bereitgestellt werden

## Lösung

Die Umsetzung des *Front Door* Patterns (eine Spezialisierung des Integration Reverse Proxy) ermöglicht das zentrale Erstellen, Validieren, Löschen und Verwalten von Benutzersessions.

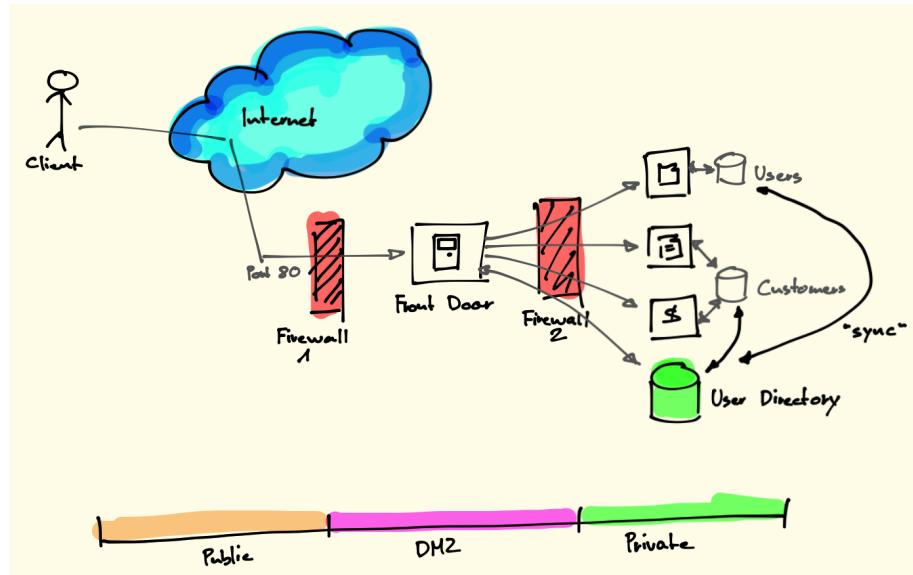


Abbildung 4.18.: Struktureller Aufbau Front Door

Ein zentrales *User Directory* ermöglicht das Authentifizieren von Benutzern.

## Umsetzung

1. Es muss eine klare Repräsentation aller Benutzerinformationen gefunden werden. Dies gestaltet sich dann schwierig, wenn bestehende Komponenten mit bestehenden Benutzerdatenbanken integriert werden müssen.
2. Die Authentication-Mechanismen müssen definiert werden (Username/Password, Username/Token usw. Siehe auch 4.2.1 "I&A Requirements")
3. Zugriffsberechtigungen müssen definiert werden. Auf Ebene der *Front Door* kann bspw. allen Mitarbeitern Zugang gewährt werden. Später bei einzelnen Komponenten nur noch ausgewählten (Siehe u. A. auch 4.1 "Access Control Models")
4. Definition, wie Sessions im System weitergegeben werden sollen (Cookies? Header-Fields? Interne und externe Repräsentation? Siehe auch 4.3.4 "Security Session")
5. Umsetzung des effektiven Session Contexts für die *Front Door*
6. Umsetzung eines *Cookie Jars* für Cookies der internen Web- und Applikationsservern (Auch hier müssen entsprechende Sessions aufrechterhalten werden können)

7. Gestaltung und Umsetzung der Anmelde- und Portalseite der *Front Door*. Nicht vergessen: Single Sign Off. (Siehe auch 4.3.3 "Check Point")

## Vorteile

- Single Sign On und Off ermöglichen einfaches Sessionhandling über mehrere Systemkomponenten hinweg
- Ein Benutzerprofil über mehrere Systemkomponenten hinweg
- Einzelne Komponenten müssen sich nicht mehr um I&A kümmern (analog 4.3.3 "Check Point")
- Wie beim Integration Reverse Proxy ein zentralisiertes Logging

## Nachteile

- Sessiontimeouts zwischen *Front Door* und effektiven Systemkomponenten
- Ein zentrales Verwalten der Benutzerdatenbank ist unumgänglich
- Synchronisation von Benutzerdatenbanken verschiedener Systemkomponenten kann aufwändig und fehleranfällig werden

# 4.6. Operating System Access Control

## 4.6.1. Authenticator

Der *Authenticator* ist eine konkrete Implementierung des Single Access Point. Er ermöglicht einem Betriebssystem die Verifizierung der Identität eines bestimmten Benutzers.

Dabei ist die Art und Weise, wie eine Identität verifiziert werden kann flexibel austauschbar.

## Kontext

Ein Betriebssystem prüft beim ersten Anmelden am System (evtl. später erneut) die Identität des Benutzers. Das System erstellt dabei nötige Session Context Informationen. Der Benutzer wird anschliessend durch einen User Process im System abgebildet.

Beim Zugriff auf sensitive Daten kann das System erneut nach einer Authentifizierung verlangen.

Das *Authenticator*-Pattern soll auch auf einem verteilten System eingesetzt werden können.

## Problem

Es gibt Benutzer, welche berechtigt Zugriff auf ein System haben. Wie kann verhindert werden, dass unter Voraussetzung folgender Punkte ein Angreifer sich als ein solcher Benutzer ausgeben kann?

- Tendenziell gibt es eine grosse Anzahl an Benutzern, welche evtl. sogar verschiedene/flexible Authentifizierungs-Verfahren benutzen
- Benutzer dürfen die Sicherheitsprüfung nicht umgehen können.
- Benutzer/Angreifer dürfen das Resultat der Sicherheitsprüfung (Token etc.) nicht zu ihren Zwecken verfremden können (Session Hijacking etc.)
- Die potenzielle Komponente ist zentral und wird ziemlich sicher oft und auch wiederholt zum Einsatz kommen. Performance ist darum essentiell.

## Lösung

Eine spezifische Implementation des Single Access Point, der *Authenticator* übernimmt die Verifizierung der Identität eines Benutzers.

Konnte diese erfolgreich durchgeführt werden, erstellt er einen *Proof of Identity* welcher zu späteren Zeitpunkten wiederholt zur Autorisierung im System verwendet werden kann.

Der *Proof of Identity* dient ebenfalls dazu, um weitere, „aufbauende“ Sicherheitsüberprüfungen in sensibleren Systembereichen durchzuführen.

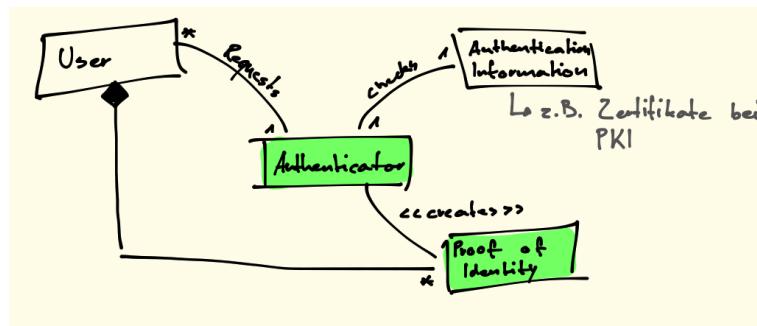


Abbildung 4.19.: Authenticator: Schematischer Aufbau

## Vorteile

- Verschiedene Authentifizierungsverfahren können angewendet werden (abhängig von Policies natürlich)
- *Authentication Information* wird getrennt vom *Authenticator* gespeichert und verwaltet und kann so explizit geschützt werden. (Readonly etc.)

- Durch die Struktur des *Authenticator*-Patterns kann es problemlos in verteilten Systemen angewendet werden.
- Die Wiederverwendung des *Proof of Identity* können wiederholende Authentifizierungen vermieden resp. vereinfacht werden

## Nachteile

- Zeitaufwand
- Komplexität und Kosten

## Reallife Beispiele

- Das aus Enterprise-Netzwerk-Systemen bekannte *Single Sign On* ist eine Variante des *Authenticator*-Patterns.  
Der *Proof of Identity* wird da meistens mittels einer PKI generiert und anschließend von verschiedenen Applikationen zur Identifizierung des Benutzers verwendet, ohne erneut nach den Authentifizierungs-Merkmalen (Passwort, Smartcard etc.) zu fragen.

## Mögliche Prüfungsfragen

- *Welche Security Patterns sind im Authenticator umgesetzt?*  
Der Authenticator ist eine konkrete Implementation des Single Access Point-Patterns.

### 4.6.2. Controlled Process Creator

Der *Controlled Process Creator* kümmert sich um die Erstellung neuer Prozess in einem Betriebssystem und stellt sicher, dass diesen korrekte Zugriffsberechtigungen zugewiesen werden.

## Kontext

Ein Betriebssystem, welches Prozesse und Threads für Anwendungen erstellen kann.

## Problem

Erstellt ein Betriebssystem einen neuen Prozess oder Thread, so erbt dieser im einfachsten Falle die Berechtigungen des übergeordneten Prozesses/Thread.

Dies kann sehr schnell zum Sicherheitsrisiko werden, kann sich ein Angreifer so doch ziemlich schnell Zugriff aufs gesamte System verschaffen.

Wie kann sichergestellt werden, dass jedem neuen Prozess oder Thread entsprechend aufgestellten Sicherheitsrichtlinien Berechtigungen zugeteilt werden? Ferner sind folgende Punkte zu beachten:

- Die Auswahl von Berechtigungen für neue Prozesse soll so einfach wie möglich sein

- Die Definition der Richtlinien, welche Prozesse welche Berechtigungen erhalten sollen, soll flexibel und einfach sein.
- Ein Prozess, welcher einem anderen untergeordnet ist, soll die Möglichkeit haben, die Berechtigungen des Übergeordneten zu übernehmen. Dies jedoch kontrolliert und unter Voraussetzung entsprechender Sicherheitsmaßnahmen.
- Die Anzahl untergeordneter Prozesse soll beschränkt werden, um Denial of Service Attacken zu verhindern.
- Ausnahmen müssen handelbar sein: Will ein Prozess auf etwas zugreifen, worauf er keinen Zugriff hat, soll dies irgendwie ermöglicht werden.

## Lösung

Prozesse werden im Normalfall direkt vom Betriebssystem selber über entsprechende Funktionen erstellt. Der *Controlled Process Creator* setzt hier an und stellt sicher, dass jedem neuen Prozess korrekte Berechtigungen zugewiesen werden.

Weiter kann der übergeordnete Prozess seinem neuen untergeordneten Prozess ein Subset seiner eigenen Berechtigungen zuweisen.

## Vorteile

- Ein neuer Prozess erhält nur die Berechtigungen, welche den definierten Richtlinien entsprechen
- Die Anzahl möglicher untergeordneter Prozesse kann zentral beschränkt werden
- Berechtigungen können den übergeordneten Prozess resp. dessen ID enthalten. So können, falls nötig, dessen Berechtigungen übernommen werden.

## Nachteile

- Das Übertragen von Berechtigungen von einem zum anderen Prozess wird tendenziell komplizierter/aufwändiger

### 4.6.3. Controlled Object Factory

Die *Controlled Object Factory* stellt sicher, dass neu erstellten Objekten (Files etc.) automatisch korrekte Rechte zugeteilt werden.

## Kontext

Ein System muss den Zugriff auf erstellte Objekte kontrollieren können. Dabei kommen Zugriffsregeln zum Zug, welche von der Klassifizierung der entsprechenden Objekte abhängig ist.

## Problem

Prozesse erstellen neue Objekte wie bspw. Dateien. Die Zugriffsregeln dieser Objekte sollen bei der Erstellung dieser gesetzt werden um unberechtigten Zugriff von Beginn an zu verhindern.

Ein weiterer Faktor stellen gepoolte Objekte dar: Zum Zeitpunkt der Zuweisung zu einem konkreten Prozess soll diesem auch gleich dynamisch der Zugriff gewährt werden.

- Prozesse erstellen verschiedene Arten von Objekten. Die Zuweisung von Zugriffsrechten soll jedoch generisch behandelt werden können.
- Für gepoolte Objekte soll eine dynamische Zuweisung von Zugriffsrechten ermöglicht werden
- Es gibt Richtlinien welche vorgeben, welche Zugriffsrechte neue Objekte erhalten sollen

## Lösung

Jedem Objekt welches neu erstellt wird, wird automatisch und zentralisiert ein Set von Zugriffsrechten zugewiesen.

## Vorteile

- Es gibt keine neuen Objekte mehr, welche Standardzugriffrechte zugewiesen haben, weil jemand vergessen hat, diese zu überschreiben.
- Es können Richtlinien definiert werden, welche Objekte wie geschützt werden sollen
- Ergänzend kann ein Betriebssystem eine *Ownership Policy* einführen. Der Besitzer eines Objekts hat dann bspw. alle möglichen Berechtigungen an einem Objekt.

## Nachteile

- Es entsteht ein entsprechender Overhead für die Definition der Rechte

### 4.6.4. Controlled Object Monitor

Mit der Controlled Object Factory wurde festgelegt, welche Prozesse wie auf ein Objekt zugreifen dürfen. Mit dem *Controlled Object Monitor* gibt es nun ein Werkzeug, wie diese Zugriffsbeschränkungen effektiv und zentral durchgesetzt werden können.

## Kontext

Ein Betriebssystem welches mehreren Benutzern Zugriff auf Objekte mit definierten Zugriffsbedingungen gewährt.

## Problem

Wie kann gewährleistet werden, dass jegliche Zugriffe auf die Objekte in einem System geprüft und ggf. abgebrochen werden, falls der zugreifende Prozess die nötigen Bedingungen nicht erfüllt?

- Es gibt viele verschiedene Objekte mit verschiedensten Zugriffsberechtigungen. Jegliche Zugriffe auf sie muss kontrolliert werden
- Es gibt verschiedene Zugriffsarten (read, write etc.)

## Lösung

Durch die Verwendung eines Reference Monitors werden alle Zugriffe von Prozessen Objekte abgefangen und überprüft.

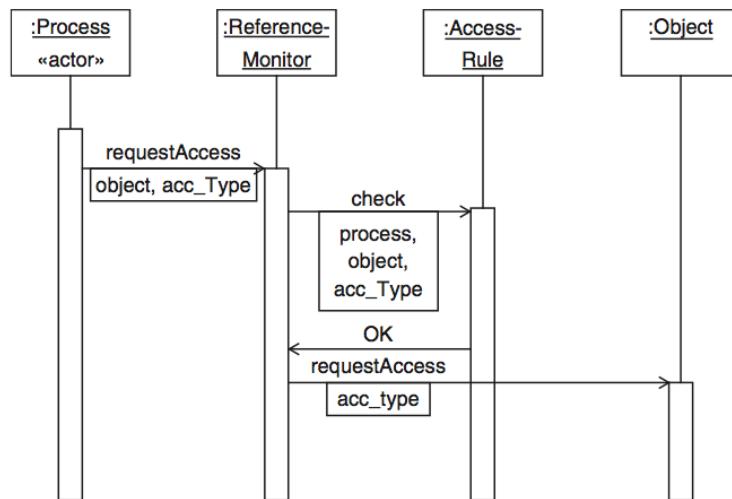


Abbildung 4.20.: Behandlung eines Zugriffs mit Controlled Object Monitor [Sch+06]

## Vorteile

- Jeder Zugriff eines Prozesses auf jegliche Objekte kann gem. Sicherheitsrichtlinien überprüft werden
- Durch Verwendung von Zugriffsmatrizen können verschiedenen Zugriffstypen definiert und gesondert geprüft werden.
- Inhaltsspezifische Regeln sind machbar (Achtung Performance!)

## Nachteile

- Zugriffsberechtigungen (z.B. Zugriffsmatrizen) müssen geschützt werden! Evtl. sogar über die gleichen Mechanismen (Oh hay, deadlock.)

- Die Überprüfung jedes Zugriffes führt zu einem entsprechenden Overhead

# Kapitel 5 Fault Tolerance

Autoren:

- Hauser Matthias
- Kretschmar Lukas
- Schmucki Michael
- Wirz Dominique

## 5.1. Introduction

In unserer Welt verlassen wir uns darauf, dass Dinge (in unserem Fall Programme) funktionieren. Aber alles kann kaputt gehen, meistens wenn wir es am wenigsten erwarten. Vor allem in der Bemannten Raumfahrt können Fehler zu verheerenden Katastrophen führen.

### Introduction to Fault Tolerance

Patterns for Fault Tolerant Software versucht mögliche Lösungen aufzuzeigen wie auf das Auftreten von Fehlern in Systemen reagiert werden kann. Und wie ein System noch lauffähig ist, auch wenn gewisse Teile ausgefallen sind oder nicht mehr korrekt funktionieren. Um aber über die Fehler Toleranz diskutieren zu können, müssen nachfolgenden Begriffe definiert werden.

#### 5.1.1. Fault, Error, Failure

- Failure (Ausfall): Ein System liefert nicht mehr die erwarteten Ergebnisse, da ein Fehler (Error) aufgetreten ist. Ausfälle können aber nur passieren, wenn definiert ist, was als korrektes Verhalten angesehen wird. Ist dies nicht spezifiziert, kann ein System auch nicht ausfallen.
- Error (Fehler): Der Zustand des Systems, wenn ein Defekt aufgetreten ist. Ein Fehler kann abgefangen werden, wenn er auftritt sodass ein Ausfall (Failure) vermieden werden kann.

- Fault (Defekt, Bug): Mögliche Ursache für das Auftreten eines Fehlers (Error). Bugs sind in jedem System vorhanden, werden aber erst erkannt, wenn sie zu einem Fehler (Error) führen.

## Abhängigkeiten

Fault führt zu Error, Error kann Failure zur Folge haben.

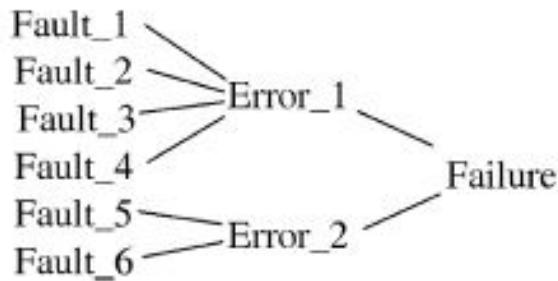


Abbildung 5.1.: Fault->Error->Failure Dependency

## Failures

Bei fail-silent Failure wird entweder das korrekte Ergebnis zurückgegeben oder keines. So kann einfach entschieden werden, ob ein Ausfall Einfluss auf andere Systeme hat. Bei korrektem Ergebnis können alle Systeme gleich weiterarbeiten und falls kein Resultat vorliegt, ist der Ausfall eines Systems bekannt und kann entsprechend behandelt werden. Falls beim ersten Auftreten eines fail-silent Failures ein System nicht mehr weiterarbeitet, spricht man von einem crash Failure.

Grob kann man Ausfälle in zwei Gruppen einteilen:

- Konsistent: gleiche Bedingungen führen zu gleichem Fehlverhalten, ist aber unabhängig vom Betrachter.
- Inkonsistent: Das Fehlverhalten ist abhängig vom Betrachter und von den Bedingungen, gleiche Bedingungen müssen aber nicht zu gleichem Fehlverhalten führen.

Fehlertolerantes Design versucht, inkonsistente Fehler in konsistente Fehler umzuwandeln, welche wiederum als fail-silent Failures behandelt werden können.

MINIMUM NUMBER OF COMPONENTS TO TOLERATE FAILURES	TYPE OF FAILURE
$n + 1$	<i>Fail-silent</i> failures
$2n + 1$	<i>Consistent</i> failures
$3n + 1$	<i>Malicious</i> failures

Abbildung 5.2.: Minimale Anzahl an Komponenten um Failures auszugleichen

### 5.1.2. Coverage

Die Absicherung (Coverage) ist die Wahrscheinlichkeit, dass ein System in vorgegebener Zeit einen Fehler automatisch beheben kann.

### 5.1.3. Reliability

Die Zuverlässigkeit (Reliability) ist die Wahrscheinlichkeit, dass in einem System in vorgegebener Zeit keine Fehler auftreten.

- MTTF (Mean Time to Failure): Durchschnittliche Zeit vom Starten einer Operation bis zum ersten Ausfall.
- MTTR (Mean Time to Repair): Durchschnittliche Zeit, die benötigt wird um eine ausgefallene Komponente wieder in einen funktionierenden Zustand zu versetzen.
- MTTR (Mean Time between Failures):  $MTTF + MTTR$
- MTTR (Failures in Time):  $\frac{\text{Failures}}{1e^9[\text{h}]}$

**Berechnung der Zuverlässigkeit:**  $reliability = e^{-(\frac{1}{MTTF})}$

### 5.1.4. Availability

Die Verfügbarkeit (Availability) ist die Wahrscheinlichkeit in der Zeit, dass ein System erreichbar ist und seine vorgesehenen Aufgaben korrekt erledigen kann.

EXPRESSION	MINUTES PER YEAR OF DOWNTIME
100%	0
Three 9s	525.6
Four 9s	52.56
Four 9s and a 5	26.28
Five 9s	5.256
Six 9s	0.5256
100%	0

Abbildung 5.3.: Downtime per year

**Berechnung der Verfügbarkeit:**  $availability = \frac{MTTF}{MTTF+MTTR}$

### 5.1.5. Dependability

Die Verlässlichkeit (Dependability) ist die Masseinheit für ein System wie stark man sich auf dessen Verlässlichkeit, Verfügbarkeit und Sicherheit verlassen kann. Die Sicherheit wird durch zwei Faktoren beeinflusst. Zu meinem der „Safety“ (Nichtauftreten von Ausfällen, die einen grösseren Schaden anrichten, als der Mehrwert, aller Vorteile eines System, mit sich bringt) und der „Security“ (Verhinderung von unautorisierten Zugriffen/Aktionen).

### 5.1.6. Performance

Die Leistungsfähigkeit (Performance) ist stark gekoppelt an die Zuverlässigkeit. Anforderungen an die Leistungsfähigkeit, können auch als Anforderungen an die Zuverlässigkeit gesehen werden und umgekehrt. Grundsätzlich kann festgehalten werden, dass Ausfälle, welche auf zu viele Anfragen zurückzuführen werden können, die Leistungsfähigkeit betreffen.

Falls dies geschieht, sind drei mögliche Szenarien möglich:

- Crash: Das System bricht unter der Last zusammen
- Saturation: Das System läuft weiter, aber mit stark verminderter Leistungsfähigkeit (während der Überlastung) und kehrt schlussendlich wieder zu seiner erwarteten Leistungsfähigkeit zurück.

- Capacity decrease: Das System läuft weiter, aber die Leistungsfähigkeit wird dauerhaft vermindert bleiben.

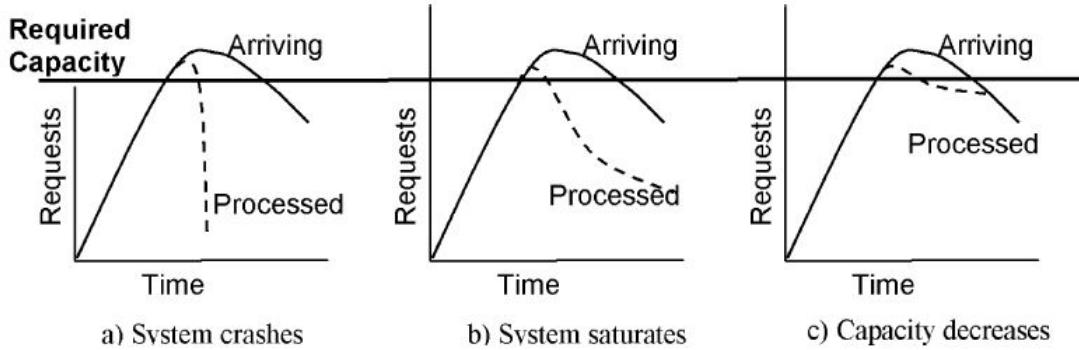


Abbildung 5.4.: Performance

## 5.2. Fault Tolerant Mindset

Die Fehlertoleranz im Hinterkopf zu behalten ist essentiell für die Entwicklung einer fehlertoleranten Software. Dabei ist es wichtig sich in allen Phasen der Entwicklung fortlaufend zu fragen: Was könnte falsch laufen bzw. zu Fehlern führen?

### 5.2.1. Design Tradeoffs

Every problem in computer science boils down to tradeoffs. – Professor L. J. Henschen

Dies trifft natürlich auch auf Fehlertoleranz zu, hierbei findet der Tradeoff zwischen MTTF und MTTR statt. Es kann sein, dass es wichtiger ist, dass die MTTR klein ist und es nicht so sehr auf die MTTF ankommt. Wiederum gibt es andere Beispiele, wie in der Raumfahrt, in der die MTTR länger dauern kann, die MTTF jedoch auch sehr lange sein muss, nämlich solange, wie die Mission dauert, also bis das Shuttle wieder zurück auf der Erde ist.

### 5.2.2. Quality vs Fault Tolerance

Der wesentliche Unterschied besteht darin, dass sich Fault Tolerance darum kümmert, dass sich ein Programm fehlerfrei ausführen lässt. Die Qualität bezieht sich dagegen darauf, wie wenige Faults sich im Code finden lassen. Somit steht Qualität für möglichst wenige Faults im Code, was jedoch nicht zu einem fehlerfreien Betrieb führen muss.

### 5.2.3. Keep it Simple (KIS)

Wichtigster Punkt ist, dass je komplexer/länger ein Codestück ist, desto schwieriger wird es, dass ein Programm fehlerfrei bleibt. Eine mögliche Variante ist die Anzahl der Zeilen zu reduzieren und z.B. lieber den einfachen Mechanismus zu implementieren und dadurch den Unterhalt des Codes zu vereinfachen.

### 5.2.4. Incremental Additions of Reliability

= nicht alle "Fault Tolerant Patterns auf einmal anwenden, da sich dadurch meist neue Fehler einschleichen.

### 5.2.5. Defensive Programming

Mit defensivem Programmieren wird beschrieben, dass man sich in jeder Situation folgende Fragen beantworten soll:

- Was kann momentan falsch laufen?
- Welche Fehler können auftreten?
- Wie kann dieses Codestück scheitern?
- Wie kann ich diesen Code vor Fehlern schützen bzw. diese beheben?

#### Faults in Fault Tolerance Code

Immer im Hinterkopf halten: mit komplexen Fehlerbehandlungen wird die Möglichkeit von Fehlern erhöht.

#### Memory Corruption

Es kann immer vorkommen, dass ein Hardwareteil Fehler aufweist, wodurch Daten, die gelesen werden, fehlerhaft werden können. Deshalb ist es wichtig, Daten immer auf ihre Richtigkeit zu prüfen, vor allem in Fällen, in denen die Daten den weiteren Verlauf des Programms entscheiden.

#### Data Structures

Datenstrukturen sollten immer so designt werden, dass sie auf Korrektheit geprüft werden können (Wenn möglich immer bereits implementierte Datenstrukturen verwenden und nicht eine neue Linked List implementieren).

#### Design for Maintainability

Systeme, die Fehler tolerant sind, leben sehr lange. Deshalb ist es wichtig sie wartbar zu halten. Dies bedeutet, den Code simpel zu gestalten, verständliche Funktionsnamen zu verwenden und wenn nötig Kommentare zu setzen. -> KIS

### Coding Standards

Damit die Qualität des Code hoch gehalten wird, sollten Coding Standards verwendet werden. Jedoch sollten nur eine übersehbare Anzahl Coding Guidelines erstellt werden, da es dadurch einfacher ist, sie einzuhalten und zu verifizieren.

### Redundancy

Es ist wichtig das Redundanz Fault Tolerance unterstützt und nicht weitere Funktionalität hinzufügt.

### Static Analysis Tools

Statische Code-Analyse oder kurz statische Analyse ist ein statisches Software-Testverfahren welches zur Compilezeit stattfindet. Der Quelltext wird hierbei einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Sorten von Fehlern entdeckt werden können, noch bevor die entsprechende Software (z. B. im Modultest) ausgeführt wird. Die Methodik gehört zu den falsifizierenden Verfahren, d. h. es wird die Anwesenheit von Fehlern bestimmt. Quelle: [http://de.wikipedia.org/wiki/Statische\\_Code-Analyse](http://de.wikipedia.org/wiki/Statische_Code-Analyse)

### N-Version Programming (NVP)

Die Idee dahinter ist, die Implementation eines Anforderungskatalogs durch 2+ Teams in unterschiedlichen Sprachen implementieren zu lassen. Vorteil daraus ist, dass jedes Programm meist an unterschiedlichen Stellen Fehler aufweist, wodurch die Codes optimiert werden können.

### Redundant Disks (RAID)

Die Möglichkeit mehrere Festplatten zusammenzufassen und dadurch die Fehlertoleranz und Redundanz zu erhöhen.

#### 5.2.6. The Role of Verification

Verifikation ist wichtig, da aufgrund von Tests und Verifikation Berechnungen zur Verfügbarkeit erstellt werden können.

#### 5.2.7. Fault Insertion Testing

Beschreibt das Testverfahren mit der Absicht fehlerhafte Eingaben zu tätigen, damit der Verlauf des Programms bei Fehleingaben analysiert werden kann. Damit werden oft auch Grenzverhalten getestet.

#### 5.2.8. Fault Tolerant Design Methodology

1. Finde Punkte die scheitern können
2. Definiere Strategien zur Fehlerminderung

3. Definiere Systemgrenzen und Teile die redundant sein müssen
4. Architektur und Haupt-Design Entscheide können nun stattfinden
5. Kontrolliere, ob die zuvor definierten Fehlmöglichkeiten abgedeckt sind oder ob sich neue aufgetan haben
6. Identifizierte Probleme, die mit der Interaktion eines Benutzers entstehen und behoben werden können

### 5.2.9. Fragen

**Beschreiben sie ein realistisches Beispiel in welchem der Tradeoff zwischen MTTF und MTTR stimmt.**

In einem Telefon Netzwerk ist der Tradeoff der Switches wichtig. Dabei kann die MTTF eher vernachlässigt werden, solange die MTTR kurz ist.

**Ist ein Pointer eine potentielle Fehlerquelle? Wenn ja, was kann dagegen gemacht werden?**

Ja, es ist wichtig sich immer zu fragen wie der Pointer genutzt werden kann und wie er geprüft werden soll.

**Was ist der Vorteil von RAID5 gegenüber RAID0?**

RAID0: nur Verteilen der Daten auf unterschiedliche Festplatten wodurch nur die Performance besser wird, jedoch nicht die Redundanz.

RAID5: durch Paritätsbereiche gibt es die Möglichkeit, dass eine Festplatte ausfallen kann -> Redundanz.

**Nennen sie die 6 Punkte der "Fault Tolerant Design Methodology"**

- Was könnte scheitern bzw. welche Fehler könnten auftreten
- Definiere Strategien zur Fehlerminderung
- Definiere Systemgrenzen und Teile die redundant sein müssen
- Architektur und Haupt-Design Entscheide können nun stattfinden
- Kontrolliere ob die zuvor definierten Fehlmöglichkeiten abgedeckt sind oder ob sich neue aufgetan haben
- Identifizierte Probleme die mit der Interaktion eines Benutzers entstehen und behoben werden können

### 5.3. Introduction to the Patterns

Die vier Phasen des Lebenszyklus eines Fehlers sind:

- error detection
- error recovery
- error mitigation
- fault treatment

Diese spielen wie folgt zusammen:

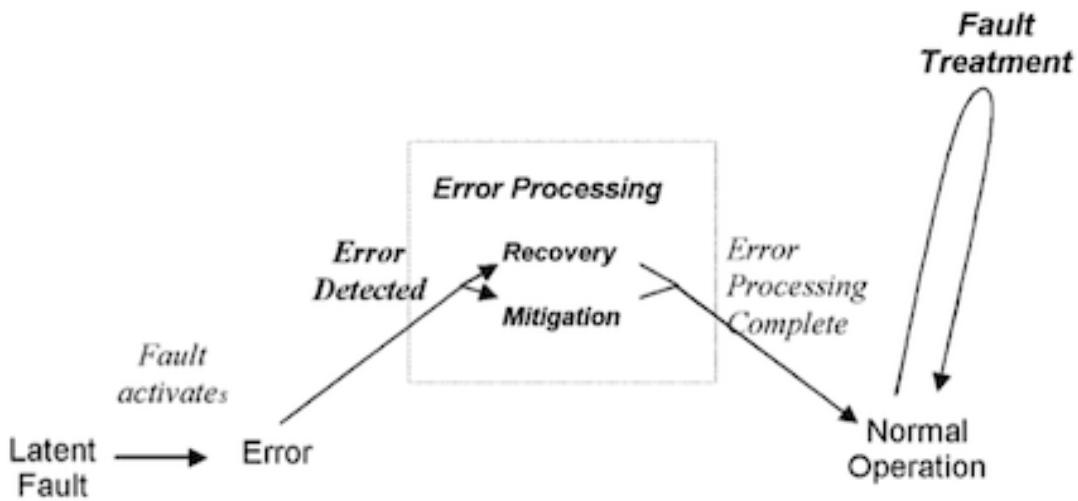


Abbildung 5.5.: introduction four phases of fault tolerance

#### 5.3.1. Shared Context for These Patterns

Die Patterns zielen auf folgende Attribute ab

##### Real-Time

Zwei unterschiedliche Varianten

- soft real-time: Beispiel Webserver antwortet direkt auf Anfrage, ist jedoch nicht schlimm wenn es mal länger dauert
- hard real-time: Beispiel Flugzeug, falls ein Kontrollsysteem nicht in Echtzeit reagiert kann dies katastrophale Folgen haben

### State or Stateless

- stateless, eine Anfrage wird verarbeitet und ist danach nicht mehr Teil des Systems
- stateful, Systeme dessen Verarbeitung längere Zeit in Anspruch nehmen und sich Daten merken müssen

### External Observers

Oft besitzen Fehlertolerante Systeme Observer bzw. Monitoring Teile welche Informationen über aufgetretene Fehler sammeln. Dies ist eine wichtige Anforderung an das System, da dadurch Fehler analysiert und reduziert werden können.

### Integrated Fault Tolerance

Meist ist die Fehlerbehandlung in Programm integriert, dies macht es schwierig eine Fehlerbehandlung in anderen Situationen wieder zu verwenden.

### Fault Tolerance is Not Free

Behalte im Hinterkopf das eine Fehlerbehandlung meist nicht gratis ist, z.B. braucht eine redundante Daten Kopie zusätzlichen Speicher.

### Long Lived Systems

Lang lebend = höhere Entwicklungskosten = Erwartung das Fehler Tolerant

### 5.3.2. Terminology

Ein "System" besteht aus "Komponenten" ("Klassen" oder "Module"), diese Komponenten laufen in oder enthalten verschiedene "Tasks", diese wiederum laufen auf einer Einheit von Hard- oder Software. Solange keine Fehler den Verlauf eines Systems stören, nennt man dies "normal processing". Wenn ein System mit unterschiedlichen Komponenten zusammenarbeitet nennt man diese "peers".

### 5.3.3. Fragen

1. Wie heißen die vier Phasen eines Faults?

- error detection
- error recovery
- error mitigation
- fault treatment

2. Weshalb ist Fault Tolerance nicht gratis?

- Redundanz kostet Speicher
- Monitoring kostet Rechenleistung

## 5.4. Units of mitigation

### Einleitung

Dieses, wie auch die folgenden Patterns, zielen auf alle Teile eines Systems ab, und nicht nur auf ein bestimmtes Modul oder eine bestimmte Klasse.

Die Architektur eines Systems hat einen beträchtlichen Einfluss auf die Fehlertoleranz. Deshalb sind die Architectural Patterns auch die ersten, die auf ein neues Projektdesign angewendet werden und sind aus diesem Grund auch die ersten in diesem Buch beschriebenen.

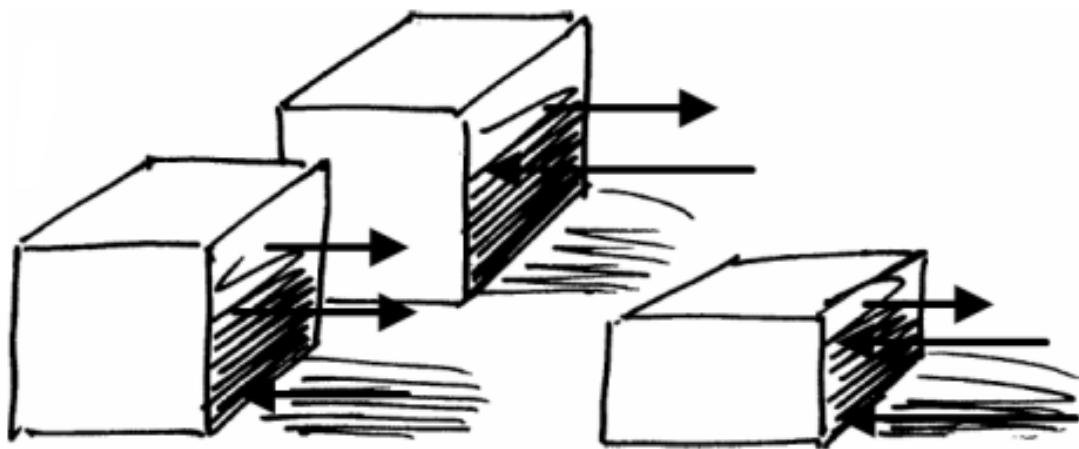


Abbildung 5.6.: unitsOfMitigation

### Problem

Wie kann ich verhindern, dass das ganze System ausfällt, sobald irgendwo ein Fehler auftritt? Es sollte möglich sein, dass nur ein Teil ausfällt, den Fehler bestensfalls behebt und sich dann wieder ins System einklinkt.

Dieser eine Teil wird hier 'unit of mitigation' genannt. Wieviele davon das System enthält, und wie gross diese sind, hängt sehr von der jeweiligen Situation ab.

Wie alle Massnahmen zur Fehlertoleranz liefert auch die unit of mitigation ('Einheit zur Schadensmilderung') einen overhead an Code und Komplexität. Sie sollte deshalb nicht zu klein gewählt werden (beispielsweise jeden Ort 'einkapseln', an dem ein Fehler auftreten kann). Ist sie zu gross, sind wir wieder beim Beispiel, dass das ganze System ausfällt. Es gilt also, einen guten Mittelweg zu finden.

### Lösung

*Teile das System so ein, dass jeder Teil sowohl mögliche Fehlerquellen wie auch Massnahmen zur Erkennung und Behebung beinhaltet.* Wähle die Aufteilung, die am besten für das System geeignet ist.

lung so, dass sie für dein System Sinn macht.

Als mögliche, natürliche Trennlinien können in Frage kommen:

- Tiers / Layers
- Teilapplikationen
- Prozessoren, Cores
- Threads
- Tasks
- Terminals / Protokol Handlers
- Gruppen von Funktionalitäten

Die units of mitigation sollten gegen Aussen genau definierte Interfaces haben. Diese bilden dann eine strikte Barriere für die Fehler und es sollte höchstens bekannt gegeben werden, dass ein solcher aufgetreten ist. Dies bedingt, dass eine Einheit in der Lage sein muss, Fehler zu erkennen und zu beheben.

Das Hinzufügen von Redundanz kann helfen, das System lauffähig zu halten. Ist eine unit damit beschäftigt, einen Fehler zu beheben, kann das redundante Gegenstück einspringen und die Mehrarbeit übernehmen.

Lässt sich ein Modul gut restarten, spricht dies für eine gute Wahl einer unit of mitigation.

### Correcting Audits

#### Begriffe

##### dynamisch vs. statisch

Daten können dynamisch oder statisch sein. Die Telefonvorwahl eines bestimmten Kantons wird wohl nicht so bald ändern und gehört somit zu den statischen Daten. Der Wechselkurs einer Fremdwährung dagegen ändert sich sehr dynamisch.

#### Problem

Defekte (faulty) Daten führen zu Fehlern (error). Solche Fehler können und werden immer wieder auftauchen, und müssen auch nicht deterministisch sein. Die Gründe sind vielfältig:

- Verfälschung auf Hardware-Ebene: Memory, Strahlung, ...
- Andere Systemfunktionen, welche Daten falsch gespeichert haben

Dauert die Erkennung des Fehlers zu lange, kann es sein, dass andere Module die Daten nutzen und im schlimmsten Fall das System zum Crashen bringen.

## Lösung

**Finde und korrigiere defekte Daten so früh wie möglich. Prüfe, ob ähnliche, bzw. zugehörige, Daten auch korrupt sind und protokolliere den Fehlerauftreten.**

### Finden und Korrigieren

Daten können anhand verschiedener Kriterien geprüft werden:

- Struktur: Prüfe, ob (double) linked lists korrekt verkettet sind, Pointers auf Listen oder Queues innerhalb der bekannten Grenzen sind, Sizecounters die korrekte Anzahl Elemente angeben, ...
- Zusammenhang: Passen gleiche oder ähnliche Daten zueinander? (z.B. könnten Daten an einem Ort in °C gespeichert sein, an einem anderen Ort in °F.)
- 'Ergibt das Sinn?': Ein int mit Wert 2013 kann zwar ein gültiges Jahr repräsentieren, aber kaum ein gültiges Jahr. Checksummen können hier helfen, Fehler zu erkennen.
- Vergleich: Redundant gespeicherte Daten können direkt miteinander verglichen und so auf Fehler überprüft werden. Dies macht v.a. für statische Daten Sinn, zum Teil aber auch für sehr wichtige dynamische.

Daten müssen natürlich so designet werden, dass sie geprüft werden können:

- Doppelt verkettete Listen halten die Verkettung redundant
- Nutze redundante Speicherorte an verschiedenen Stellen im System
- Nutze nicht-triviale Wertevorgaben; Fehler werden so offensichtlicher

Verschiedene Patterns helfen, defekte Daten zu korrigieren:

- Data Reset
- Error Correcting Code
- Marked Data
- Complete Parameter Check
- Rollback
- Return To Reference Point

Falls die Korrektur erfolgreich war, führe den Schritt noch einmal mit den korrigierten Daten aus.

### Korrupte Verwandte

Wurde ein Fehler automatisch gefunden, sollte überprüft werden, ob nicht andere Daten an anderen Orten auch betroffen sind. Möglicherweise entstand der Fehler aus der Berechnung von schon fehlerhaften Daten. Oder der korrekte Datenbestand wird für die Berechnung weiterer Daten gebraucht. Diese Daten sind potenzielle weitere Fehlerquellen und sollten dann ebenfalls überprüft werden.

### Protokollieren

Ein gefundener Fehler sollte immer geloggt werden. Wiederholte Auftritte von Fehlern können helfen, die defekten Daten zu finden.

#### 5.4.1. Escalation

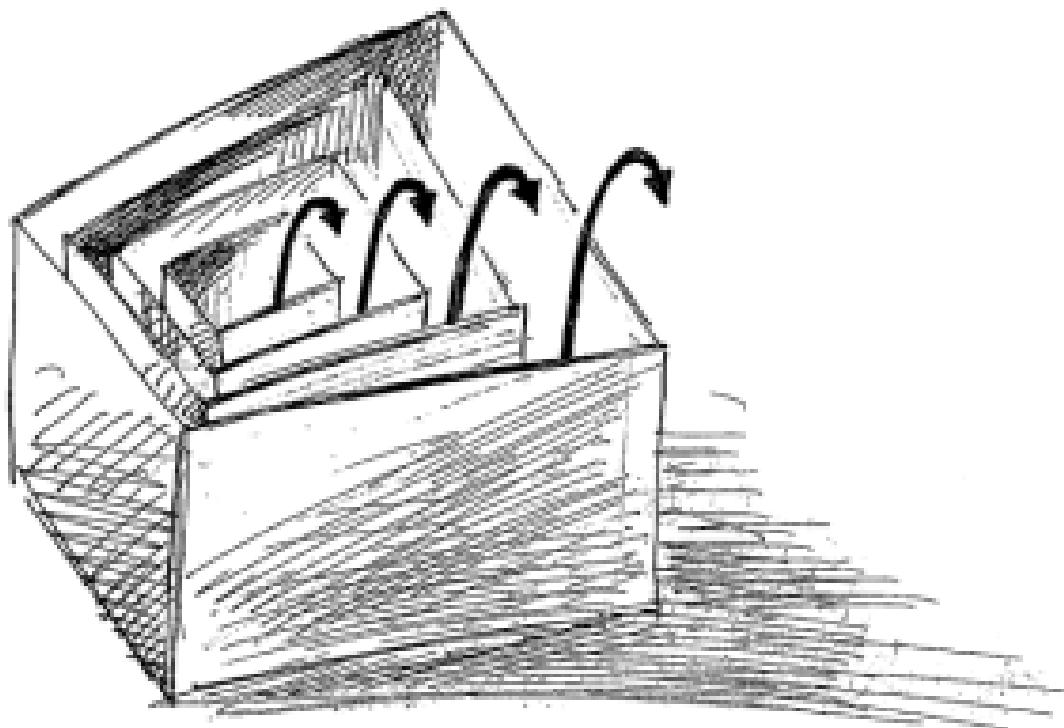


Abbildung 5.7.: escalation

### Problem

Trotz allen Versuchen kann es sein, dass sich das System im Fehlerfall nicht wiederherstellen kann. Sowohl Correcting Audits, als auch Restarts, Data Resets und Rollback-

oder Rollforwardversuche verlaufen manchmal erfolglos. In einigen Fällen kann es genügen, diese Versuche wiederholt anzuwenden, bis das gewünschte Ergebnis erzielt ist. Doch wie bei einer beschädigten LP, die immer wieder dieselbe Stelle wiederholt, ist manchmal drastischeres Eingreifen gefragt. Da das System das Prinzip von Minimize Human Intervention verfolgt, soll es dafür aber selbst verantwortlich sein.

### Lösung

**Wenn Fehlerkorrektur und -abschwächung fehlschlagen, führe die nächst drastischere Massnahme aus.**

Es kann nützlich sein, einen Zähler zu haben, damit das System weiß, wie oft ein Wiederherstellungsversuch in welcher Zeit schon fehlgeschlagen ist. Wird ein gewisser Wert überschritten, kann die Eskalation in die nächste Phase ausgelöst werden.

### Menschliches Eingreifen

Ab einer bestimmten Phase wird menschliches Eingreifen unumgänglich. Dem Operator muss dann aber eine Liste mit möglichen Schritten vorliegen. Diese sollte so geordnet sein, dass zuerst Massnahmen ausgeführt werden, welche eine hohe Recoverywahrscheinlichkeit haben, aber gleichzeitig schnell sind und einen minimalen Einfluss auf andere Systemoperationen haben. Später folgen dann die schwereren Geschütze.

## 5.4.2. Redundancy

### Definition

Redundanz bedeutet, dass man über zusätzliche bzw. andere Wege verfügt, um die gleiche Arbeit zu verrichten.

### Einleitung

Es gibt zwei Arten wie Errors mit Hilfe von Redundanz behandelt werden können; Mit einigen Verfahren gelingt es, Errors zu behandeln ohne den Systemzustand zu ändern (z.B. Block code). Normalerweise ist dies jedoch nicht möglich und erst nach der Behandlung des Errors kann das System wieder in einen fehlerfreien Zustand überführt werden.

Für beide Arten der Fehlerbehandlung kann Redundanz helfen, einen solchen Error zu behandeln bzw. dessen Auswirkungen so gut wie möglich zu überbrücken. (MTTR)

### Typen von Redundanz

#### Spatial

Von räumlicher Redundanz spricht man, wenn von einem System mehrere Kopien existieren. Die Zeit welche zwischen dem Erkennen des Fehlers bis zur Rückführung in einen fehlerfreien Zustand verstreicht, wird als „Recovery Time“ bezeichnet. Räumlich redundante Systeme können diese Zeit mit einer fehlerfreien Kopie des Systems überbrücken.

Beispiel: Nameserver (DNS), RAID1, N-Version Programming

### Temporal

Zeitliche Redundanz, wobei die Redundanz über eine Zeitdauer vorhanden ist, um korrekte Resultate zu erhalten. (Nachteil: Unavailability kann erhöht werden)

Beispiel: I-frame ( <http://en.wikipedia.org/wiki/I-frame> )

### Informational

Informatorische Redundanz, wobei Information zur Erkennung und Behebung von Fehlern wiederholt wird.

Beispiel: Block Code

### Methoden für Temporal Redundancy

#### Active-Active

Vollständig Redundante Systeme agieren parallel und teilen sich die Arbeit (Load Balancing). Trotzdem ist jedes System fähig die gesamte Arbeit alleine zu verrichten.

#### Active-Passive

Eine Abwandlung von Active-Active wobei das zweite/”redundanteSSystem erst beim Auftreten eines Errors die Arbeit übernimmt.

#### N+M Redundancy

Active-Active bzw. Active-Passive setzen eine teure 1:1-Assoziation zwischen den Systemen voraus. N+M Redundancy reduziert die Kosten indem für N aktive Systeme M redundante Systeme bestehen, wobei  $M < N$

### Tradeoff

Redundanz bedeutet auch immer einen Mehraufwand und somit höhere Kosten. Ausserdem handelt es sich bei Computern um deterministische Maschinen welche bei gleichen Rahmenbedingungen, Konfiguration und Input auch gleichermaßen auf Fehlsituationen reagieren.

### Prüfungsfragen

- Definieren Sie den Begriff Redundanz.
- Nennen Sie je ein Beispiel für Spatial, Temporal und Informational Redundancy.
- Unterscheiden Sie die Begriffe Active-Active, Active-Passive und N+M Redundancy.

- Unterscheiden Sie die Begriffe Active-Active, Active-Passive und N+M Redundancy.

### 5.4.3. Recovery Blocks

#### Einleitung

Im Kapitel [http://wiki.ifs.hsr.ch/APF/U11\\_3\\_Redundancy](http://wiki.ifs.hsr.ch/APF/U11_3_Redundancy) wurde bereits darauf hingewiesen, dass eine exakte Kopie auch die gleichen Fehler produzieren wird. Daher müssen alternative Lösungen für eine Problemstellung unterschiedlich implementiert werden. Dies kann wie folgt bewerkstelligt werden

- Redundante Implementierungen parallel ausgeführt. Mittels Voting wird anschließend das beste Resultat gewählt. (Nachteil: Overhead)
- Alternative werden nur ausgeführt, wenn das erste Resultat nicht befriedigend war, die ist die Recovery Block Strategie.

#### Beispiel

```

Try
  Heap sort
  if not okay then throw
Catch
  Try
    Insertion sort
    if not okay then throw
  Catch
    Bubble sort
  
```

#### Tradeoff

- Akzeptanz-Tests: Es ist nicht immer einfach zu entscheiden ob ein Resultat akzeptiert werden kann.
- Alternativen: Für gewisse Algorithmen gibt es keine oder nur unbefriedigende Alternativen.
- State: Der System State muss vor dem ersten Block gespeichert werden, damit für den nächsten Block die gleichen Bedingungen herrschen.
- Komplexität: Recovery Blocks an sich führen zu mehr Code und somit zu mehr Komplexität. Auch die vorherigen Punkte tragen zu einer höheren Komplexität bei.

## Prüfungsfragen

- Um welchen Typ von Redundanz handelt es sich bei Recovery Blocks?
- Stellen Sie die Funktionsweise von Recovery Blocks anhand von Pseudocode und einem Fluss-Diagramm dar.
- Nennen Sie drei Nachteile von Recovery Blocks.
- Nennen Sie drei Nachteile von Recovery Blocks.

### 5.4.4. Minimize Human Intervention

#### Ausgangslage

Für viele Fehler in Systemen sind deren Benutzer verantwortlich, da sie mit falscher Anwendung diese provozieren.

#### Lösungsansatz

Es gibt drei Kategorien von Fehler in Systemen:

1. Hardware
2. Software
3. Prozedurale

#### Vergessene Aktionen

Grundsätzlich verhalten sich Personen schlechter als Computer, wenn sie immer dieselben Schritte (Prozeduren) durchlaufen müssen. Ein Computer hält sich strikt an die vorgegebene Reihenfolge (ausser Ausnahmen sind explizit zugelassen), eine Person hingegen kann einzelne Schritte überspringen, sei das nun weil sie diesen vergessen oder absichtlich nicht erledigt hat. Läuft ein System fehlerfrei, dann wird diesem System weniger Aufmerksamkeit geschenkt, und dabei können wichtige Aktionen vergessen gehen, falls diese dennoch erwartet werden. Tritt ein Fehler auf, kann ein Computer viel schneller reagieren als eine Person, die das System bedient.

#### Unerlaubten Aktionen

Es kann aber auch vorkommen, dass eine Person denkt, dass ein System nicht mehr richtig funktioniert, obwohl dies nicht der Fall ist. Dieser Umstand ist darauf zurückzuführen, dass einer Person nicht genügend Feedback gegeben wird, dass ein System noch am Arbeiten ist. Die einfachste Form hierfür ist eine Progressbar oder die animierten Mauszeiger, welche aus den Betriebssystemen bekannt ist. Im Allgemeinen gilt: „A quiet system is a dead system“ Tritt diese Situation ein, kann es sein, das seine Person Aktionen auslöst, die nicht erwartet werden und die Situation meist verschlechtern. So kann sie einen einwandfrei laufenden Prozess abschalten oder weitere Prozesse starten, die das System noch mehr auslasten.

### Schlussfolgerung

Ein System soll so designet werden, dass es selbst Fehler behandeln und automatisch wieder einen normalen Zustand erreicht. Dies führt zu einer schnelleren Fehlerbehandlung, kürzerer Ausfallzeit und es wird vermieden, dass Prozedurale-Fehler während der Behandlung eines Fehlers auftreten können.



Abbildung 5.8.: MinimizeHumanIntervention

Erreicht kann dies wie folgt werden:

1. Fehler müssen einem Fault Observer (10) gemeldet werden.
2. Es dürfen keine internen Meldungen nach aussen gesandt werden.
3. Die Unit of Mitigation (1) soll Fehler selbst erkennen, behandeln und beheben können, ohne dafür auf Interaktion mit einer Person angewiesen zu sein.

Fehlerbehandlung:

- Recovery Blocks (4)
- Error Handler (30)

Vermeidung von Prozeduralen-Fehlern:

- Maximize Human Participation (6)
- Maintenance Interfaces (7)
- Reintegration (59)
- Revise Procedure (63)

### 5.4.5. Someone In Charge

#### Ausgangslage

Ein System besteht aus mehreren Units of Mitigation (1), kann Redundancy (3) verwenden und verfolgt das Prinzip von Minimize Human Interaction (5). Dennoch können überall Fehler auftreten, selbst in der Fehlerbehandlung. Tritt dieser Fall ein, kann es sein, dass das System, zusätzlich zum Ausfall der Funktionalität auch noch die Fehlerbehandlung nicht weiterführen kann.

#### Lösungsansatz

Bei der Fehlertoleranz gibt es zwei Arten der Fehlererkennung:

1. Fehler erkennen, der passiert ist (komplexer)
2. Die Komponente im System finde, welche nicht mehr korrekt funktioniert (einfacher)

Hat ein System eine Komponente, die weiß was bei der Erkennung eines Fehlers gemacht werden muss, führt das zu einem robusteren System. Mindestens sollte diese den Fehler einem Fault Observer (10) oder System Monitor (15) melden können, nebst einer möglichen Fehlerbehandlung. Ein Fault Observer (10) ist zuständig für das Sammeln und Weiterleiten von Informationen von Fehlern, ein System Monitor (15) hingegen erkennt Fehler. Wichtig ist einfach, dass im Falle eines Fehlers bekannt ist, wer sich darum kümmern muss. Das Problem dabei ist aber, dass dadurch ein Single Point of Failure entsteht, da diese Komponente auch ausfallen kann. Es ist deshalb nicht definiert, dass es nur eine Komponente geben darf, die reagieren kann. Es ist lediglich definiert, dass zu jedem Zeitpunkt klar sein muss, wer reagieren muss. Wichtig ist auch zu beachten, dass die verantwortlichen Komponenten sich nicht um zu viel kümmern müssen. Es ist einfacher, wenn sich viele verschiedenen Komponenten um verschiedene Fehlerbehandlungen kümmern, da diese so einfacher wartbar sind und bei einem Ausfall einer Komponenten nicht die gesamte Fehlerbehandlung betroffen ist.

Action	In Charge
Checkpointing	Each task
Rollback	Component R
Roll-Forward	Component R
Load Shedding	Component S

Abbildung 5.9.: ResponsibilitiesList

#### Schlussfolgerung

Alle Aktivitäten, die fehlertolerant sein müssen, benötigen eine einzige Komponente, die in einem Fehlerfall reagiert. Es muss auch möglich sein, dass Fehler bei der Fehlerbe-

handlung erkannt werden können und falls nötig Escalation (9) eingesetzt wird.

### Verwandte Patterns

Fehler-„Verwaltung“:

- Fault Observer (10)
- System Monitor (15)
- Heartbeats (16)
- Acknowledgements (17)
- Escalation (9)

#### 5.4.6. Maximize Human Participation

##### Ausgangslage

Soll ein System Anwender total ignorieren, um die prozeduralen Fehler zu minimieren?

##### Lösungsansatz

Das System soll dem Benutzer die Möglichkeit bieten das Systems bei der Fehlerbehandlung zu unterstützen. Beispielsweise versucht das System immer wieder mit einem ROLLBACK zu einem CHECKPOINT (37) zu springen, obwohl der Speicher, an dem der CHECKPOINT gespeichert wurde, beschädigt ist. Der Benutzer könnte nun das System dazu zwingen einen RESTART (31) durchzuführen, anstatt eines ROLLBACK (32).

Wichtig ist dem Benutzer die wichtigsten Systeminformation zu präsentieren. Dabei soll darauf geachtet werden, dass weniger wichtige Informationen nach den kritischen Informationen folgen. Nachrichten, welche Fehler melden, werden im Fault Tolerance Bereich als 'Action Messages' bezeichnet.

Das System sollte weiter einen 'Safe Mode' bieten, in welchem es keine weiteren automatischen Fehlerbehandlungen vornimmt. Dies ist vor allem in Kritischen System sehr wichtig.

##### Schlussfolgerung

Ein System soll so designet werden, dass es für erfahrene Benutzer einfach ist in einem positiven Weg auf das laufende System einzuwirken. Hierzu kann ein gutes MAINTANCE INTERFACE (7) und ein gescheiter FAULT OBSERVER (10) hilfreich sein.

Ebenfalls wichtig für die Stabilität des Systems ist es, nach einem Error eine ROOT CAUSE ANALYSIS (62) durchzuführen, um mögliche Ursachen zu identifizieren und falls nötig ein SOFRWARE UPDATE (11) einzuspielen.

### 5.4.7. Maintenance Interface

#### Ausgangslage

Sollen Wartungs- und Anwendungsanfragen in den Ein- und Ausgangskanälen des Systems vermischt werden?

#### Lösungsansatz

Es sollte ein eigenes Interface für Wartungsanfragen zur Verfügung gestellt werden. Wobei diese Anfragen priorisiert werden und in der Anwendung auch bei Überlast abgearbeitet werden, wodurch das Wartungspersonal immer in der Handlungsmacht ist. Des Weiteren bringt dies eine höhere Abschottung mit sich, da es aus der 'normalen' Applikation nicht möglich ist in den Wartungsmodus zu gelangen. Ebenfalls kann dieses Interface mit nötigen Security Features geschützt werden.

#### Schlussfolgerung

Stellen sie ein eigenes Interface für die exklusive oder fast exklusive Möglichkeit für Wartungsarbeiten zur Verfügung.

### 5.4.8. Fault Observer

#### Ausgangslage

Das System ist so designt das es Fehler entdeckt und automatisch behandelt und löst. Wie erfahren nun Personal oder andere Systemkomponenten von einem möglichen Problem?

#### Lösungsansatz

Der herkömmliche Ansatz ist das PUBLISHER-SUBSCRIBER Pattern, welches einen effektiven Weg zur Verteilung von aufgetretenen Fehler darstellt. Hierbei registrieren (subscriben) sich die Observer für Informationen/Komponenten. Sobald neue Informationen oder Fehler eintreffen werden diese an die registrierten Observer weiterverteilt.

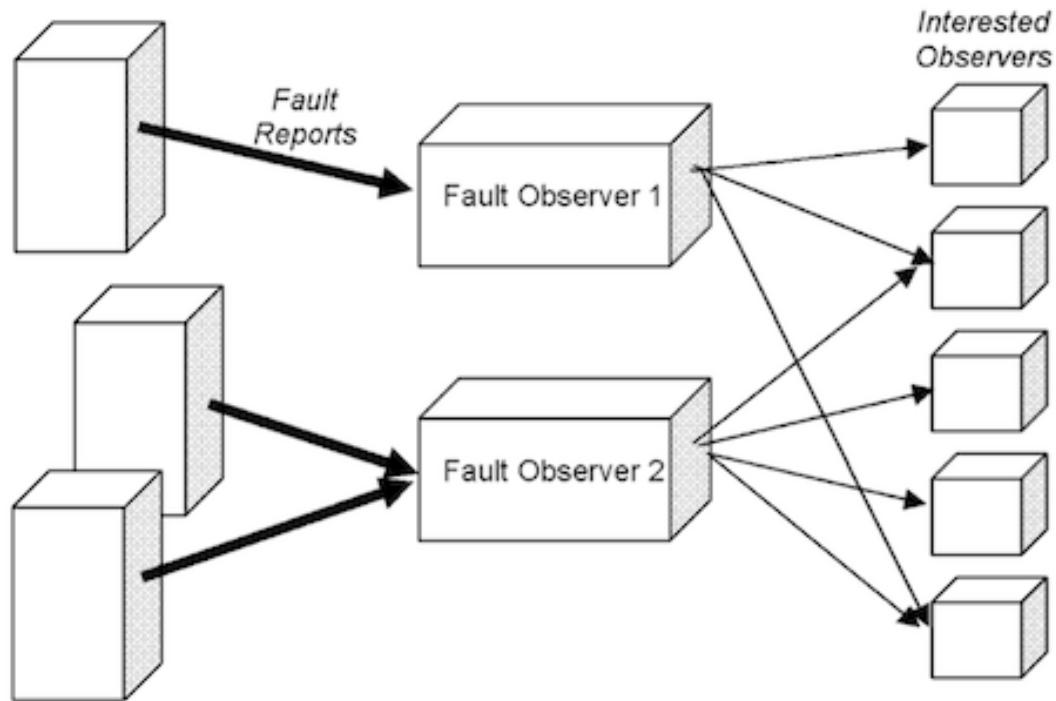


Abbildung 5.10.: U11 3 FaultObserver 2013

Wichtig ist darauf zu achten das die Publisher einmalig oder als redundante Komponenten vorhanden sind und nicht mehrere Publisher die den selbe Information an einen Observer übermitteln. (führt zu Konfusion beim Observer und zu duplicated Code im System) Des Weiteren wäre es wünschenswert, dass bei einem Error der dazugehörige Fault mit geloggt wird um spätere Analysen zu vereinfachen.

### Schlussfolgerung

Raportiere alle Errors an einen Fault Observer, welcher alle interessierten Parteien darüber informiert.

#### 5.4.9. Software Update

##### Ausgangslage

Trotz guter Qualitätssicherungsmethoden zur Vermeidung von Fehlverhalten kann es zu Fehlern kommen. Wodurch Software Updates nötig werden, diese dürfen jedoch das System nicht zu Anhalten zwingen und die damit verbundene Downtime minimieren.

### Lösungsansatz

Vor der ersten Applikationsauslieferung muss geklärt sein wie sie erweitert bzw. erneuert werden kann, ohne das System zu stoppen.

Falls die neue Version einer Applikation zur gleichen Zeit wie die alte ausgeführt werden kann, ist es möglich einen Failover Routine laufen zu lassen die zwischen der alten und der neuen Version interagiert. Dies ermöglicht eine minimale Downtime. Zudem können automatisierte Akzeptanztests gestartet werden, welche prüfen ob die neue Version korrekt arbeitet. Falls dies nicht der Fall ist, kann mittels Recovery Blocks wieder zur alten Version zurück gewechselt werden.

In einem komplexen System können nicht alle Komponenten des Systems gleichzeitig ausgetauscht werden. Es kann also erforderlich sein, dass das neue Komponenten rückwärtskompatibel sein müssen. Dies kann unter anderem erreicht werden, indem ein Versionsindikator eingeführt wird und die neuen Funktionen Regeln enthalten, welche festlegen wie sie mit fehlenden oder geänderten Attributen umgehen soll.

### Schlussfolgerung

Designen sie die Applikation so, das Änderungsmöglichkeiten bereits in ihrerem ersten Release miteingeflossen sind. Halten sie sich immer vor Augen wie sie die Software im ausgelieferten Zustand erweitern und verbessern können, da ein einbauen einer Update Routine nachdem Ausliefern keine einfache Sache ist.

## 5.5. Detection Patterns

Die erste Phase von Fault Tolerance ist Entdeckung. Faults und daraus resultierende Errors müssen erst erkannt werden, bevor Recovery- oder Schadensminderungsaktionen etc. ausgeführt werden können.

Zwei Paare von Konzepten sind dabei wichtig: 'Errors' vs. 'Failures' und 'a priori Wissen' vs. 'Vergleich von redundanten Elementen'.

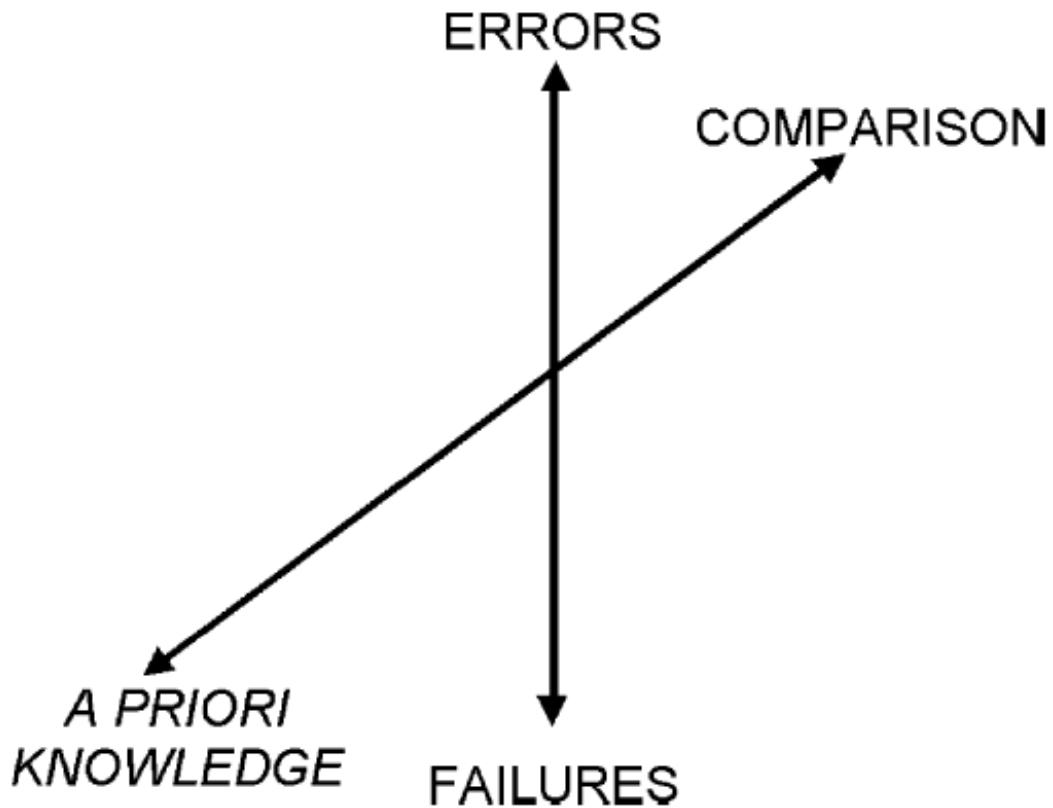


Abbildung 5.11.: detection concepts

#### 'a priori Wissen' vs. 'Vergleich von redundanten Elementen'

Wir können Wissen über das System und darin geltende Constraints nutzen um zur Laufzeit Zustände, Resultate, Seiteneffekte etc. überprüfen zu können.

Die meisten Patterns nutzen eher den zweiten Ansatz, wie z.B. REDUNDANCY(4).

Ein weiterer Aspekt wäre, dass das Programm selbst in der Lage ist, zu erkennen, dass (und welche) Fehler immer wieder auftauchen und so automatisch Errors und Failures erkennt.

#### 'Errors' vs. 'Failures'

Das System muss in der Lage sein, sowohl Errors als auch Failures zu erkennen. Wir sind natürlich v.a. an den Errors interessiert, da diese die "Wurzel der Problemeßind. Bestenfalls finden wir sie, wenn sie noch gar nie im System Schaden anrichten konnten.

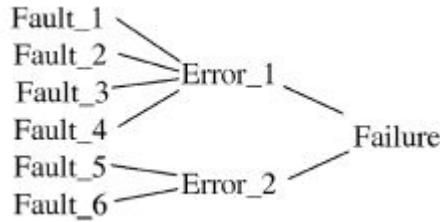


Abbildung 5.12.: Fault Error Failure Dependency

## Übersicht über die Patterns im folgenden Kapitel

### 5.5.1. Fault correlation

#### Problem

Ein Fehler ist aufgetreten und wurde detektiert. Es gibt aber viele verschiedene mögliche Ursachen. Die Frage lautet nun: Welcher Fault führte zum Fehler?

#### Lösung

Ein Error oder Failure kann von diversen Faults ausgelöst werden. Beim Auftritt eines Fehlers sollten diverse Fragen gestellt werden, wie z.B.:

- Was hat der Fehler im System schon angestellt?
- Wurde die Ausführung gestoppt? Wenn ja, welche Funktionen sind nicht mehr verfügbar?
- Wurden Logs erstellt?
- Welche Daten waren nicht korrekt?
- ...

Die ursprüngliche Fehlerquelle zu finden ist extrem wichtig!

- Ein Error kann weitere Errors verursachen.
- Ähnliche Errors oder Failures könnten evtl. die gleiche Behandlungsprozedur besitzen. So kann z.B. in einem Redundanten System auf eine andere Komponente ausgewichen werden und die Fehlerbehebung hat noch etwas Zeit. Andererseits kann es sein, dass der Fehler so schnell wie möglich behoben werden muss, damit das System nicht davon 'verseucht' wird.

Um letzteres zu ermöglichen, sollten gewisse Bug-Kategorien erstellt werden, damit klar ist, wie mit einem entdeckten Fault umgegangen werden muss.

PATTERN	PATTERN INTENT
FAULT CORRELATION (12)	Analyze multiple error indications to identify the actual active fault.
ERROR CONTAINMENT BARRIER (13)	Isolate errors so that they do not spread.
COMPLETE PARAMETER CHECKING (14)	Check all the inputs and parameters rigorously to prevent bad results from causing errors during execution.
SYSTEM MONITOR (15)	Some errors will only manifest themselves at a system level. Check for them at this level.
HEARTBEAT (16)	Send a status report at regular intervals to let other parts of the system know their status.
ACKNOWLEDGEMENT (17)	Send a reply message to let a communicating party know that the sender is alive.
WATCHDOG (18)	Build a special entity to watch over another to make sure that it is still operating well.
REALISTIC THRESHOLD (19)	Thresholds for detection of problems should be set realistically. This applies to communications (HEARTBEAT (16), ACKNOWLEDGEMENTS (17)) as well as anything counted (LEAKY BUCKET COUNTERS (27)).
EXISTING METRICS (20)	Monitor metrics that are already included in the system and won't take precious computing time to compute.
VOTING (21)	When more than one result is available for a computational result or question or task, vote to pick the correct one.
ROUTINE MAINTENANCE (22)	Periodically and automatically perform routine, preventive maintenance to prevent faults from silently accumulating.
ROUTINE EXERCISES (23)	Run ROUTINE EXERCISES to know that REDUNDANT (3) hardware is available for use when needed.
ROUTINE AUDITS (24)	Check data by a background task to make sure that it is correct.
CHECKSUM (25)	Add information to data or messages to verify that they are correct.
RIDING OVER TRANSIENTS (26)	Sometimes the prudent thing to do is to ignore an error if it is something that might be due to a transient situation.
LEAKY BUCKET COUNTER (27)	Implement a method to RIDE OVER TRANSIENTS (26) by keeping a counter that is automatically decremented and incremented by errors.

Abbildung 5.13.: pattern thumbnails

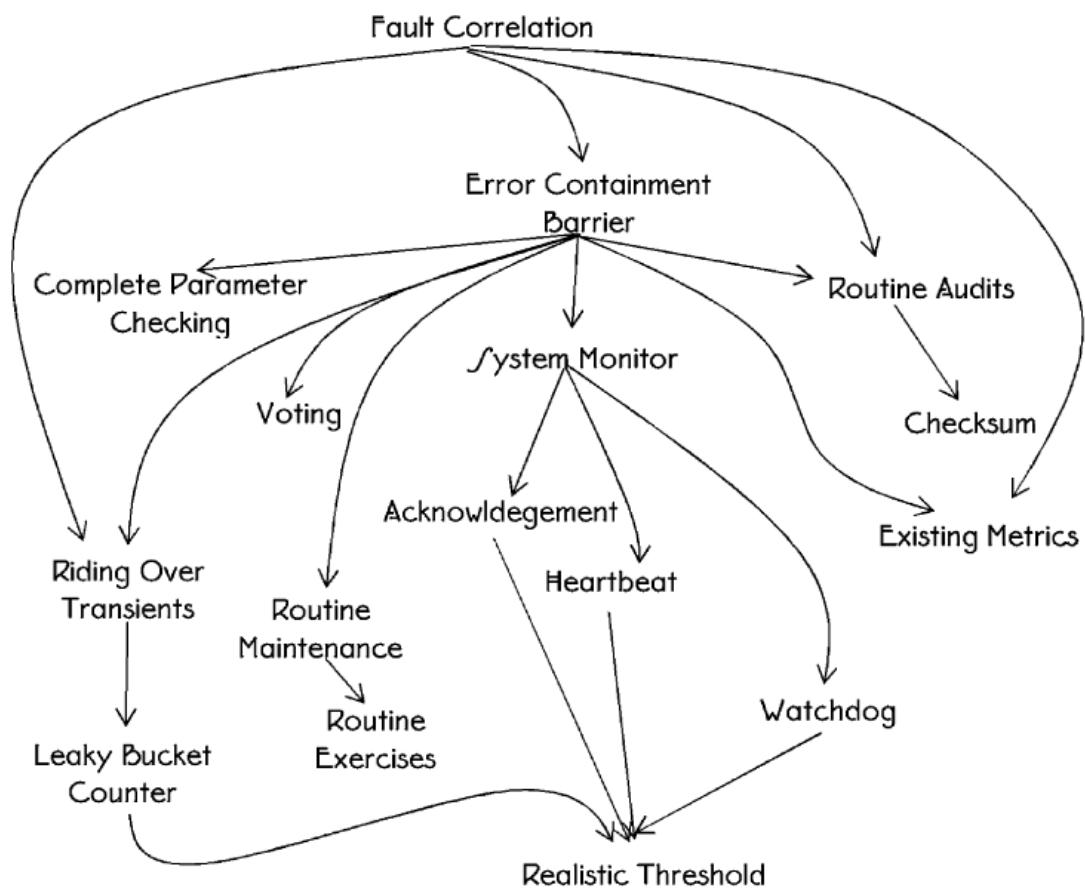


Abbildung 5.14.: pattern map

### 5.5.2. Error containment barrier

#### Problem

Was soll das System als erstes tun, wenn ein Error detektiert wird? Ohne Behandlung wird er für immer im System bleiben oder aber früher oder später in einem Failure enden. Was genau geschehen wird kann allerdings selten genau vorausgesagt werden.

Was aber soll getan werden? Eine Möglichkeit ist, 'HILFE' zu schreien und zu beenden, was aber MINIMIZE HUMAN INTERVENTION widerspricht (kann aber nötig sein, wenn sicherheitskritische Fehler geschehen). Den Fehler weitestgehend zu ignorieren ist auch nicht immer die beste Lösung. Es ist auch nicht immer möglich, schadensbegrenzende Schritte (CORRECTING AUDITS etc.) einzuleiten.

Bis sie jemand stoppt, breiten sich Errors durch das System von Komponente zu Komponente aus.

#### Lösung

Der Error muss in einer UNIT OF MITIGATION isoliert und der Fluss in andere Teile des Systems mit einer Barriere gestoppt werden (Stichwort QUARANTINE).

Der Error darf nicht unbehandelt bleiben. Löse parallel dazu geeignete Benachrichtigungen, Logging, Mitigation und Recovery Funktionen aus.

### 5.5.3. Riding over transients

#### Problem

Einige Probleme treten nur sehr selten bis einmalig auf. Eine Bodenwelle bringt den Stossdämpfer kurzzeitig in Unruhe, sonst ist aber meist nicht viel los. Genau so können in Software Fehler selten auftreten und nur kurzzeitig eine Auswirkung haben (Rauschen auf einem Bus, Alphateilchen, ...). Wie kann man jetzt verhindern, dass das System für solche transiente Fälle unnötig viele Ressourcen verbraucht?

#### Lösung

Führe FAULT CORRELATION durch, wenn ein Fehler auftritt. Kann er keiner Kategorie zugeordnet werden, so beginne sofort mit der Fehlerbehandlung. Sieht er wie ein transienter Fehler aus, so rapportiere den Auftritt und unternimm nur etwas, wenn die Auftrittsfrequenz unerwartet hoch ist.

Ünerwartet hoch" kann natürlich je nach Anwendungsfall unterschiedlich verstanden werden. Wenn andere Daten in Mitleidenschaft gezogen werden muss schneller gehandelt werden, als z.B. bei Web Requests.

Geduld ist eine Kunst! Nicht immer sind die ersten Hinweise die echte Unterschrift eines Errors, weshalb zu frühes Handeln zu falschen Aktionen führen kann.

Beispiele von riding over transients:

- Ignorieren des Rückgabewerts bei Festplatten Schreibvorgängen

- Laden einer Webpage schlägt fehl. "Versuchen Sie es später noch einmal"

#### 5.5.4. Leaky bucket counter

##### Problem

Wie kann ein System wissen, ob ein Error transient oder sporadisch auftritt? Auch nicht-dauerhafte Fehler können zu Failures führen, sind aber ihrer transienten Natur entsprechend schwierig bis gar nicht zu isolieren und korrigieren.

Oft greift man bis zu einer bestimmten Anzahl selten auftretender Fehler gar nicht ein. Treten sie aber gehäuft ein, so soll das System eine gewisse Fehlerbearbeitung auslösen.

##### Lösung

Jede zu überwachende UNIT OF MITIGATION bekommt einen LEAKY BUCKET COUNTER. Tritt ein, als transient betrachteter, Fehler auf, wird der Counter inkrementiert. Periodisch wird er aber auch dekrementiert, allerdings nie unter den Anfangswert. Erreicht der Counter nun einen vorgegebenen Threshold, wird der Fehler nicht mehr als transient, sondern als permanent betrachtet.

Die Rate, mit welcher der Bucket geleert wird, muss sorgfältig gewählt werden. Geht es zu oft, so werden nicht-transiente Errors gar nie identifiziert.

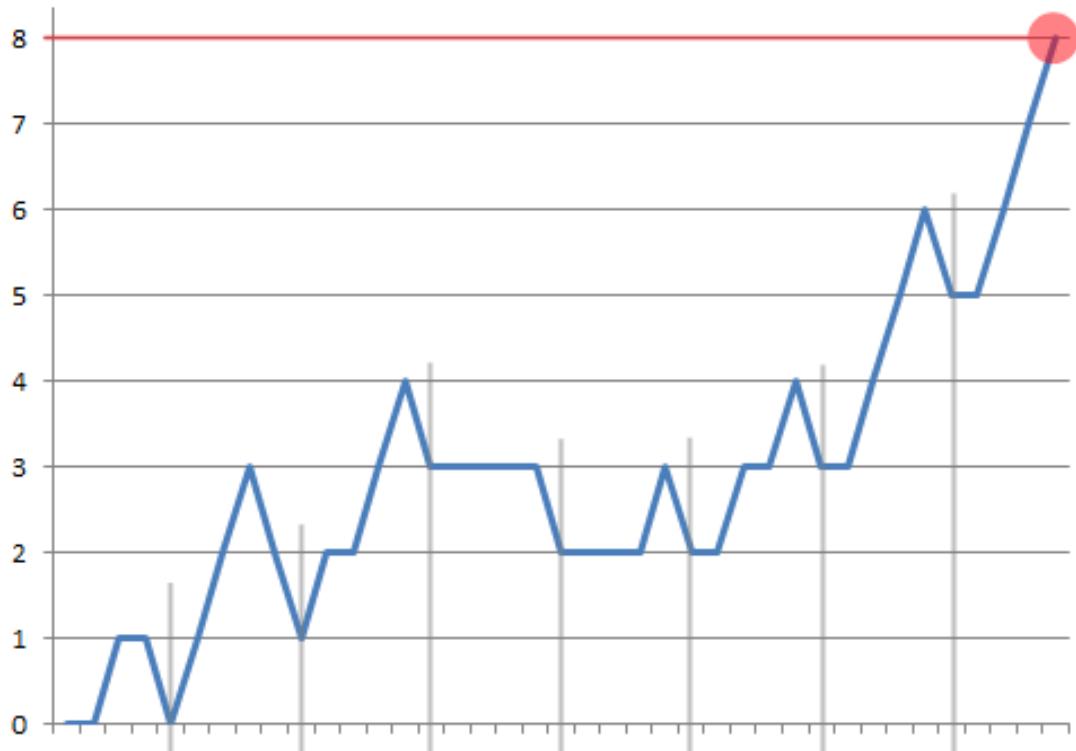


Abbildung 5.15.: leaky bucket counter

## 5.6. Error Recovery Patterns

### Einleitung

Im vorherigen Kapitel haben wir Error Detection kennengelernt; Patterns die Errors erkennen, diese aber nicht (selber) behandeln. Beim Error Recovery geht es darum, nachdem ein Fehler detektiert wurde, das System wieder in einen fehlerfreien Zustand zurück zu führen. Bei den nachfolgenden Patterns handelt es sich um Vorgehensweisen, ein System wieder in einen Zustand zu bringen, der nicht durch den Error beeinträchtigt wird (auch wenn der Fehler noch im System präsent ist). Man kann aber auch einen fehlerfreien Zustand erreichen, wenn man den Fehler maskiert und soweit abschwächt, dass er nicht mehr als Fehler gezählt werden muss. Patterns für diesen Ansatz werden später noch behandelt.

Bei den nachfolgenden Patterns geht es in erster Linie darum, eine grösstmögliche Verfügbarkeit zu erreichen. Das heisst, dass man beim Auftreten eines Fehlers diesen nicht explizit korrigiert sondern einfach in einen fehlerfreien Zustand springt. Als Folge davon verwenden viele Patterns Checkpoints um Zustände zu speichern um gegebenenfalls zu diesen zurückzukehren. Um eine hohe Availability zu gewährleisten bieten sich

der Einsatz von Error Recovery Patterns vor allem in redundanten Systemen an.

## Übersicht

Die untenstehenden Error Handler führen das System nach erfolgreicher Fehlerdetektion in einen fehlerfreien Zustand zurück.

### Error Recovery Patterns - Dependency

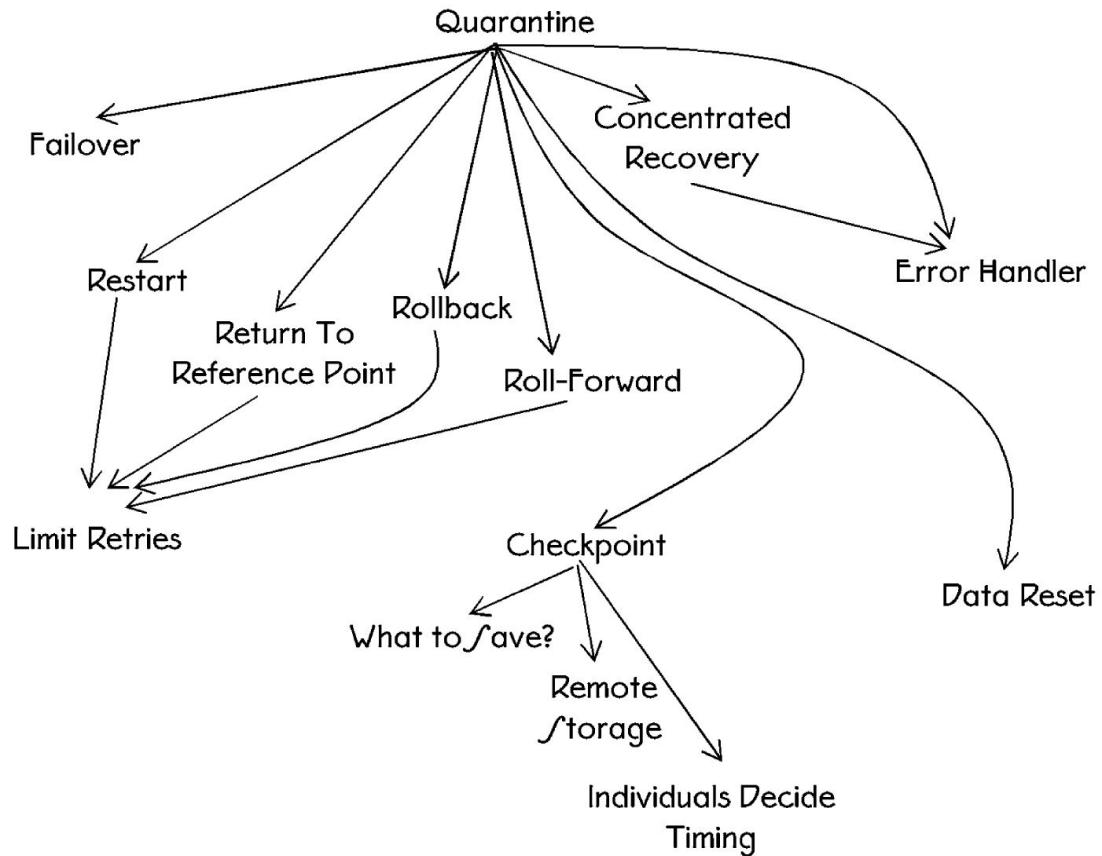


Abbildung 5.16.: RecoveryPatterns Dependency

## Error Recovery Patterns - Buch

PATTERN	PATTERN INTENT
QUARANTINE (28)	Take steps to isolate and confine a sick element to keep it from corrupting the rest of the system.
CONCENTRATED RECOVERY (29)	The system should have as few distractions as possible during error recovery.
ERROR HANDLER (30)	Provide a controlled manner for handling errors.
RESTART (31)	Resume execution by restarting the program from the beginning.
ROLLBACK (32)	Resume normal execution by moving to a state in the execution path but before the error occurred.
ROLL-FORWARD (33)	Resume normal execution by advancing to a future state that would have been reached if the error had not occurred.
RETURN TO REFERENCE POINT (34)	Resume execution by returning to a specific known state. That place might not have been in the execution path that led to the error, but it is known to be safe.
LIMIT RETRIES (35)	Do not return to the scene of an error without changing something, because the error might reoccur.
FAILOVER (36)	Recover by switching to a redundant unit.
CHECKPOINT (37)	Save state periodically so that it does not need to be regenerated from the beginning of execution.
WHAT TO SAVE (38)	Checkpoints should save information of global interest to long-duration processes.
REMOTE STORAGE (39)	Consider REDUNDANCY (3) and other recovery factors when deciding where to place checkpoints.
INDIVIDUALS DECIDE TIMING (40)	Let each process decide when to take a checkpoint based on their knowledge of their own needs.
DATA RESET (41)	Restore some data to its initial (or a predetermined) value when it is found incorrect.

Abbildung 5.17.: RecoveryPatterns

## Prüfungsfragen

- Erklären Sie den Unterschied zwischen Error Detection und Error Recovery
- Was ist das Ziel, wenn ein Error Handler "zum Einsatz" kommt?

### 5.6.1. Restart

#### Problem

Ein schwerwiegender Fehler wurde detektiert und kein Mechanismus kann bzw. konnte den Fehler beheben; D.h. alle Schritte der Escalation haben versagt. Wie kann das System wieder in einen fehlerfreien Zustand zurück geführt werden?

#### Lösung

Als letzte Möglichkeit für einen Software-Fehler, der mittels Escalation nicht behoben werden konnte, kann das System neu gestartet werden. Dieser radikale Ansatz wird Restart genannt. Ein Restart hilft aber nur, wenn es sich um ein Software-problem handelt. z.B. Bei einem Hardwarefehler würde der Fehler auch dem Restart im System erhalten bleiben.

Nachfolgenden Patterns sehen Sprünge zu fehlerfreien Zuständen vor wie Rollback, Roll-Forward und Return to Reference Point. Restart hingegen setzt alles auf den initialen Zustand zurück. Da dies aber der grösste Sprung und meist auch der zeitaufwändigste ist, sollte dieser nur wenn nötig ausgeführt werden. Es gibt auch die Möglichkeit, den Restart in verschiedene Stufen zu unterteilen. Eine mögliche Unterteilung wäre:

- warm: Es werden nur gewisse Teile des Systems initialisiert (bei den Teilsystemen, welche nicht neu gestartet werden, wird davon ausgegangen, dass sie noch einwandfrei funktionieren).
- cold: Das komplette System wird neu gestartet (nicht aber die Umgebung).
- reload: Das System wird neu in den Speicher gelesen und dann gestartet.
- reboot: Die Umgebung, auf der das System läuft, wird komplett neu gestartet.

Der „warm“ Restart wird meist bei transienten Fehlern verwendet, da diese mit grosser Wahrscheinlichkeit nicht nochmals auftreten werden und nicht das gesamte System betroffen ist.

#### Beispiel

- Bluescreen seit Windows XP

#### Tradeoff

- Cold reloads/reboots können zu inkonsistenten Daten führen.
- Sehr zeitaufwendig (Downtime/Availability)

### Prüfungsfragen

- Nennen Sie drei verschiedene Stufen des Restarts?

#### 5.6.2. Roll Back

##### Problem

Ein Fehler ist aufgetreten und behandelt worden. Da das System keinen Input ignorieren soll, muss es nach erlangen eines fehlerfreien Zustandes „ängestaute“ Requests/Messages verarbeiten. Wie soll das bewerkstelligt werden?

##### Lösung

Das System kann an eine Position zurückspringen, an der bekannt ist, dass der Fehler noch nicht aufgetreten ist. Dies ist meist der Anfang der aktuellen Verarbeitung oder ein Punkt, wo alle Komponenten synchronisiert werden/synchron sind.

Um nicht zu grosse Sprünge in Kauf zu nehmen, können mittels Checkpoints die Distanzen und somit die erneut auszuführenden Aktionen minimiert werden. Dies erfordert aber den grösseren Aufwand bei der Speicherung von Zustandsinformationen.

Wird ein Rollback durchgeführt, können zwangsläufig Aktionen mehrfach ausgeführt werden. Es ist deshalb wichtig, dass Seiteneffekte vermieden werden. Hier ist auch zu beachten, dass in hard-realtime Systemen eventuell Deadlines nicht eingehalten werden können. Springt man also an einen Punkt zurück, können respektive müssen nachfolgende Aktionen ausgelassen werden (da sie bereits einmal erfolgreich waren), die zu Seiteneffekten führen könnten oder einer zu langen Laufzeit.

Rollbacks sollten einem Fault Observer gemeldet werden. Someone in Charge kann helfen den „richtigen“ Checkpoint für den Rollback zu finden.

### Prüfungsfragen

- Was muss beim Einsatz von Roll Back beachtet werden?

#### 5.6.3. Roll Forward

##### Ausgangslage

Ein Fehler ist aufgetreten und wurde behandelt. Es kann in Kauf genommen werden, dass Requests, welche zwischen Fehlererkennung und –behandlung eingetroffen sind, ignoriert werden können.

##### Lösung

Nach der Fehlerbehandlung muss entschieden werden, wo das System weiterfahren soll. Sofern Checkpoints angelegt wurden, könnte zu diesen zurückgesprungen werden. Je nachdem sind diese aber so nahe an der Stelle, an der der Fehler passiert ist, dass sie wieder zum selben Fehler führen würden. Aus diesem Grund kann es Sinn machen, dass

man die aktuelle Verarbeitung verlässt und an einen Punkt springt, wo die nächste Aktionen (z.B. der nächste Request) verarbeitet werden kann. Hier ist aber darauf zu achten, dass der Fehler korrigiert wurde, denn dieser sollte beim nächsten Durchlauf nicht nochmals auftreten und sich nicht im System verteilen können.

Roll-Forward kann zudem schneller ausgeführt werden als Rollback. In hard-realtime Systemen wird es deshalb gegenüber dem Rollback bevorzugt. Roll-Forward darf aber nur eingesetzt werden, wenn das Verwerfen der aktuellen Daten in Kauf genommen werden kann. Ist dies nicht möglich, muss zwangsläufig Rollback eingesetzt werden.

Man soll zu einem zukünftigen Zustand springen, den man auch ohne Fehler erreicht hätte und von dem bekannt ist, dass der nicht fehlerbehaftet und mit allen Komponenten synchronisiert ist.

### Prüfungsfragen

- Was ist der Vorteil von Roll Forward gegenüber Rollback?
- Kann es vorkommen, dass durch ein Roll Forward daten verloren gehen?

#### 5.6.4. Return to Reference Point

##### Ausgangslage

Wenn ein Ablauf nicht Teil der eigentlichen Applikation ist, sondern von einer externen Quelle bearbeitet wird, kann hier kein Rollback durchgeführt werden, weil keine Informationen über Checkpoints bestehen.

##### Lösung

Beim Design der Software wurden statische Punkte im Programmcode definiert, welche sich eignen um die Ausführung fortzuführen. Dabei ist nicht definiert, ob diese Punkte vor oder nach dem Fehler sind, da sie eben statisch definiert wurden.

Gute Reference Points sind die Initialisierung einer Routine oder die Wiedereingliederung in den normalen Programmfluss.

### Prüfungsfragen

- Wann und von wem werden Reference Points festgelegt?

#### 5.6.5. Limit Retries

##### Ausgangslage

Nach dem die Error Recovery abgeschlossen ist, kann passieren, dass ein Fehler erneut auftritt. Dies kann dazu führen, dass das System in eine Endlosschleife der Fehlerkorrektur gerät und die Anforderungen an die Availability nicht einhalten kann.

## Lösung

Die Wiederholungen müssen limitiert werden: “Don’t retry if errors are likely!”

Um zu verhindern, dass das System zu lange mit dem Error Processing beschäftigt ist, muss der Loop von Error -> Error Detection -> Error Recovery durchbrochen werden. Um die Wahrscheinlichkeit des Erfolges zu steigern, kann bei den jeweiligen Wiederholungen unter Umständen der Input geändert werden. Sogenannte „Killer Messages“ können entfernt werden, falls die Gefahr besteht, dass bei deren Verarbeitung wieder ein Fehler auftritt. Dazu muss sich das System, welche Messages/Requests zum Zeitpunkt des Fehlers präsent waren.

Falls das System den Verlust einzelner Meldungen nicht tolerieren kann, müssen potentiell fehlerhafte Meldungen in einem separaten Buffer gespeichert werden, so dass Someone in Charge entscheiden kann, was mit den Messages passiert.

## Beispiel

- SMTP Mail Versand; Inkrementelle Wartezeit bis Retry

## Prüfungsfragen

- Welches Problem löst das Pattern Limit Retries?
- Falls keine Meldungen verloren gehen dürfen, was muss beim Einsatz von Limit Retry beachtet werden?

## 5.7. Error Mitigation Patterns

Die Error Mitigation Patterns versuchen, den Fehler an Ort und Stelle, an der er aufgetreten ist, zu behandeln und schliessend das System von diesem Punkt aus weiter arbeiten zu lassen. Dies steht im Gegensatz zu den Vorhergehenden Error Recovery Patterns, welche das System durch Springen in einen fehlerfreien Zustand wieder zur normalen Ausführung bringen.

Viele Fehler, die abgeschwächt (mitigated) werden können, betreffen die Zeit oder Ressource (zu wenig CPU-Zeit, zu viele Requests, zu wenig Ressourcen).

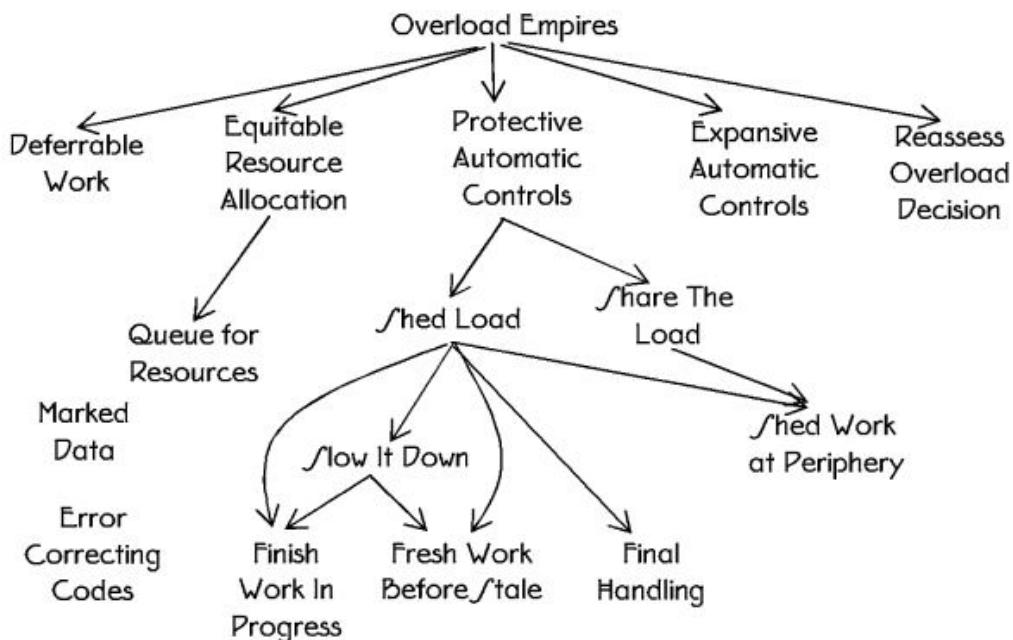


Abbildung 5.18.: MitigationPatterns Dependecy

PATTERN	PATTERN INTENT
OVERLOAD TOOLBOXES (42)	Have separate collections of techniques for dealing with different kinds of overloads.
DEFERRABLE WORK (43)	A system that is performing well in an overload situation does not need to be fixed by ROUTINE MAINTENANCE (22).
REASSESS OVERLOAD DECISION (44)	Periodically check that the error detection and correlation was correct.
EQUITABLE RESOURCE ALLOCATION (45)	Divide the resources up equitably between all the requestors.
QUEUE FOR RESOURCES (46)	Queue requests for resources in a way that will protect the system.
EXPANSIVE AUTOMATIC CONTROLS (47)	Protect the system from too much work/traffic by providing new ways to do the work.
PROTECTIVE AUTOMATIC CONTROLS (48)	Protect the system from too much work by restricting what work is allowed into the system.
SHED LOAD (49)	Discard some requests for service to offer better service to other requests.
FINAL HANDLING (50)	Gracefully remove resource allocations using the same means that normal processing does to save effort and reduce faults.
SHARE THE LOAD (51)	Move some processing to another processor. When deciding what to move, look for things that are clearly separable because this will reduce the amount of synchronization that is required.
SHED WORK AT PERIPHERY (52)	As work proceeds further into the system, more effort is expended on it. To minimize the wasted effort on work that will be shed, discard it where it first enters the system.
SLOW IT DOWN (53)	Sometimes the best thing to do when many errors are occurring is to slow down. Transients might clear and the permanent errors will become evident.
FINISH WORK IN PROGRESS (54)	Categorize arriving work as either new work or related to something that is already in progress. Give priority to work that continues work that is already in progress work.
FRESH WORK BEFORE STALE (55)	Giving better service to recent requests enables at least some of the requests to get good service. If all requests wait in a QUEUE FOR RESOURCES (46) then none of them receives good service.
MARKED DATA (56)	Mark erroneous data so that others are not corrupted by it.
ERROR CORRECTING CODES (57)	Add redundant information to data so that errors can be detected as in CHECKSUM (25) and also automatically corrected.

Abbildung 5.19.: MitigationPatterns

### 5.7.1. Overload Toolboxes

#### Ausgangslage

Das System stellt einen Fehler fest, erkennt aber, dass es sich nicht um einen Bug in der Hard- oder Software handelt sondern dieser ist entstanden durch zu viele Anfragen.

#### Lösungsansatz

Zu viele Anfragen können ein System auf drei verschiedene Arten beeinträchtigen:

1. **Memory:** Mehr Speicher wird benötigt um neue Anfragen zwischen zu speichern und abzuarbeiten. Dies kann dazu führen das aktuell abzuarbeitende Anfragen keinen Speicher mehr zur Verfügung haben.
2. **Tangible (greifbar) Resources:** Anfragen können weitere, periphere Ressourcen anfordern, die aber noch durch eine frühere Anfrage blockiert sind. Dies kann zu Verzögerungen in der Verarbeitung und zu weiteren Fehlern führen.
3. **Processor CPU Time:** Anfragen abzuarbeiten kann mehr Zeit in Anspruch nehmen als dem System zu Verfügung stehen. Dies führt dazu, dass Anfragen nicht mehr (richtig) verarbeitet werden können.

Falls diese Probleme in einer Knoten eines Netzes auftreten, kann auch eine Strategie entwickelt und umgesetzt werden, bei der, der entsprechende Knoten seine Nachbarn über die Überlastung informiert und diese ihm bei der Verarbeitung helfen können.

#### Schlussfolgerung

Verwende mehrere Toolboxes um jedes Problem auf die bestmögliche Art abzuschwächen. Eine Toolbox soll sich um Buffer und Ports kümmern, die vom System verwaltet werden. Eine andere soll sich um den Speicher kümmern und eine weitere um die CPU. Vermeide Toolboxes, die mehrere Probleme versuchen zu lösen, da diese kaum eines richtig behandeln können.

#### Verwandte Patterns

Anwendung bei zu wenig Ressourcen:

- Queue for Resources (46)
- Equitable Resource Allocation (45)
- Finish Work in Progress (54)
- Fresh Work before Stale (55)
- Share the Load (51)
- Shed Load (49)
- Finish Work in Progress (54)

### 5.7.2. Deferrable Work

#### Ausgangslage

Ein System hat übermäßig viele Anfragen. Um den korrekten Betrieb sicher zu stellen werden Routine Audits (24) und Routine Maintenance (22) angewandt. Durch die höhere Belastung des Systems treten aber keine Fehler auf, die behandelt werden müssen, lediglich die Ressourcen werden knapper.

#### Lösungsansatz

Ist ein System stabil und steht es vor einer Überlastung, kann es Sinn machen, die Prüfungen, welche Fehler verhindern sollen, später als geplant durchzuführen. Denn diese Routine-Checks sind nicht wichtig für das Abarbeiten der Anfragen. Das System soll sich bei einer Überlastung auf seine primäre Funktionalität konzentrieren.

#### Schlussfolgerung

Routinearbeit kann aufgeschoben (deferred) werden. Bei einem System, welches vor einer Überlastung steht, ist die Wahrscheinlichkeit gross, dass alle Komponenten richtig funktionieren. Die Routinearbeit soll dann wieder einsetzen, wenn die Ressourcen wieder vorhanden sind.

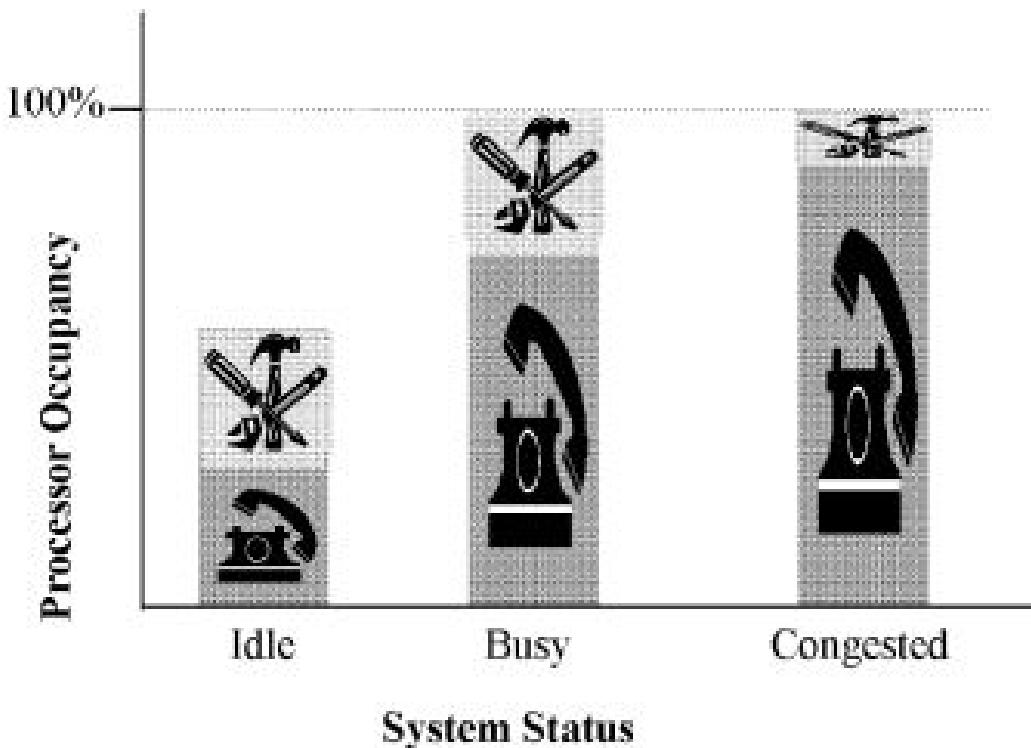


Abbildung 5.20.: DeferrableWork

### Verwandte Patterns

Falls ein Fehler Überlastung verursacht:

- Reassesses Overload Decision (44)

### 5.7.3. Reassess Overload Decision

#### Ausgangslage

Ein System verwendet Fault Correlation (12), um eine Überlastung abzuwenden. Deshalb wird Deferrable Work (43), Finish Work in Progress (54) und Shed Load (49) angewandt. Die Überlastung verringert sich aber nicht.

#### Lösungsansatz

Falls erkannt wird, dass die Überlastung trotz der Versuche diese abzuwenden gleich bleibt oder gar zunimmt, kann es sein, dass die Überlastung nur eine Effekt eines Feh-

lers ist, der behandelt werden muss. Nimmt die Überlastung zu, muss dies im System festgestellt werden können um weitere respektive andere Massnahmen zu ergreifen.

### Schlussfolgerung

Es soll ein Feedback-Loop vorhanden sein, der es ermöglicht, die gefällten Entscheidungen zu korrigieren. Dies ermöglicht es dem System, eine andere Fehlerbehandlung durchzuführen, falls der gewünschte Effekt durch die gewählte Strategie nicht erreicht wird.



Abbildung 5.21.: ReassessOverloadDecision

### Verwandte Patterns

Anwendung von:

- Escalation (9)
- Someone in Charge (8)

### 5.7.4. Equitable Resource Allocation

#### Ausgangslage

Verschiedene Typen von Anfragen werden von einem System behandelt. Hinzu kommt noch, dass diese unterschiedlich priorisiert sein können. Das System muss verhindern, dass es zusammenbricht, auch wenn nur ein Typ oder Priorität zur Überlastung neigt.

#### Lösungsansatz

Als Beispiel nehmen wir Anfragen an eine Firmen-Webseite. Einige wollen nur Informationen abgreifen, andere Bestellungen platzieren. Zudem sollen Anfragen von Angestellten mit höherer Priorität behandelt werden als jene von Kunden. Würde man nun strikt nach Fresh Work before Stale (55) vorgehen, würde dies bedeuten, dass immer die neuste Anfrage verarbeitet wird. Dies kann aber dazu führen, dass höher priorisierte Anfragen

tiefer priorisierten weichen müssen. Das führt dann zu einer Priority-Inversion, falls tiefer priorisierte Anfragen Ressourcen blockieren, welche von höher priorisierten benötigt werden. Eine andere Lösung wäre, basierend auf den eingehenden und bereits vorhandenen Anfragen die Ressourcen zu verteilen. So können so viele Anfragen wie möglich mit den vorhandenen Ressourcen verarbeitet werden und Überlastungen blockieren nicht alle Typen und Prioritäten von Anfragen.

### Schlussfolgerung

Bündle ähnliche Anfragen zusammen und stelle ihnen Ressourcen nach ihren Prioritäten bereit. Dies ermöglicht das Verarbeiten von allen Typen von Anfragen, auch wenn einige Gruppen zur Überlastung neigen.

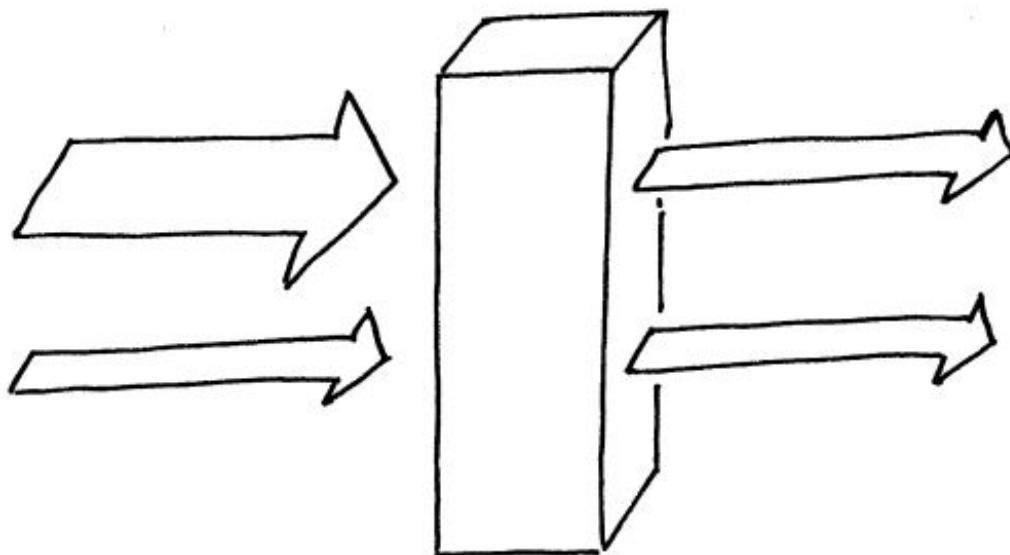


Abbildung 5.22.: EquitableResourceAllocation

### Verwandte Patterns

Anwendung:

- Queue for Resources (46)

#### 5.7.5. Queue For Resources

##### Ausgangslage

Ein System versucht eine Überlastung zu verhindern. Es ist aber nicht mit einer Fehlerbehandlung beschäftigt sondern erhält einfach zu viele Anfragen.

## Lösungsansatz

Eine Möglichkeit wäre, nur die Anfragen zu behandeln, welche mit den freien Ressourcen behandelt werden können. Alle anderen werden verworfen. Shed Load (49) löst eine Überlastung so. Mit diesem Ansatz gibt es aber einige Probleme:

- Eine Anfrage, welche sich aus mehreren kleineren Anfragen zusammensetzt, kann nicht abgeschlossen werden, weil eine Anfrage abgewiesen wurde.
- Wichtige Anfrage werden ohne Prüfung ignoriert (Siehe Maintenance Interface (7))
- Die Überlastung kann nur von kurzer Zeit sein und eine abgewiesen Anfrage könnte kurze Zeit später verarbeitet werden.
- Die Queue wird zu lange, sodass sie nicht mehr verwaltet werden kann.
- Es stehen auch nach einer Weile nicht genügend Ressourcen für die vorderste Anfrage bereit und diese blockiert nun die Queue oder muss verworfen werden.
- Die Verwaltung der Queue benötigt zusätzliche Ressourcen und kann sehr ineffizient umgesetzt sein.

## Schlussfolgerung

Speichere Anfragen, die nicht direkt verarbeitet werden können, in einer Queue.

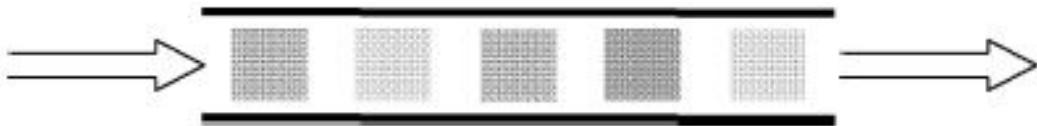


Abbildung 5.23.: QueueForResources

## Verwandte Patterns

Queues:

- FIFO (First In/First Out): Für Systemanfragen
- LIFO (Last In/First Out aka Stack): Für Anfragen von Benutzern. Derjenige, der als letztes die Anfrage abgesetzt hat, erhält am schnellsten Antwort. Derjenige, der als erstes eine Anfrage gestellt hat, hat wohl bereits aufgegeben.
- Equitable Resource Allocation (45)

### 5.7.6. Expansive Automatic Controls

#### Ausgangslage

Wie kann man es vermeiden Zeit unnötig für nicht behandelbare Anfragen zu vergeuden und gleichzeitig andere Anfragen schnellst möglich abarbeiten?

#### Lösungsansatz

Versuche das System so zu designen, dass bei Überlast alternative Wege genommen werden können. Dies könnte zu nicht Lastzeiten mit zusätzlichen Ressourcen erreicht werden, welche bei Last eingreifen können.

#### Schlussfolgerung

Dieses Pattern versucht Zusatzwege in ein zu entwickelndes System einzubauen, welche eingeschlagen werden falls eine Grenze überschritten wird. Erst wenn die Überlast nicht trotz zusätzlicher Ressourcen nicht gelöst werden kann soll Shed Load (49) eingesetzt werden.

#### Verwandte Patterns

Anwendung bei zu wenig Ressourcen:

- Shed Load (49)

### 5.7.7. Protective Automatic Controls

#### Ausgangslage

Wie soll das System mit dem Overhead von zu vielen Anfragen umgegangen werden die bearbeitet werden müssen?

#### Lösungsansatz

Hierbei bestehen drei mögliche Ansätze:

1. Versuche alles zu unternehmen um das System nicht zum Absturz zu bringen, dies könnte auch heissen keine Anfragen zu beantworten.
2. Versuche so viele Anfragen wie möglich zu beantworten und lass dabei unnötige Vorberitungsarbeiten weg.
3. Mache gar nichts, was jedoch meist in Instabilität endet.

Ein gutes Beispiel für dieses Pattern ist, ein Lichtsignal an der Einfahrtsrampe einer Autobahn, welches in Aktion tritt sobald die Autobahn anfängt zu verstopfen. Diese Pattern wird verwendet, wenn die Ressourcen begrenzt sind.

### 5.7.8. Marked Data

#### Ausgangslage

Ein Fehler wurde entdeckt, kann aber nicht korrigiert werden. Wie kann dieser verhindert werden, dass er sich im System weiterverbreitet?

#### Lösungsansatz

Falls der Fehler bereits im Speicher vorhanden ist, können die Error Correcting Codes des Speichers verwendet werden, um den Fehler zu erkennen. Wurde der Fehler nicht erkannt können die Daten beim ersten Gebrauch geprüft werden und als fehlerhaft markiert werden. Oftmals hilft eine Markierung nicht aus, da danach jeweils geprüft werden muss ob diese Markierung vorhanden ist oder nicht. Deshalb könnten wie im Beispiel des IEEE Standards für NaN, ein neuer Wert eingeführt werden der im Fehlerfall zur weiteren Verarbeitung weitergegeben wird. Dabei ist das Verhalten bei der weiteren Verarbeitung definiert, also wie sich das System Verhalten soll falls der Wert für eine Berechnung verwendet wird.

#### Schlussfolgerung

Markiere fehlerhafte Daten und definiere geeignete Regeln für die Verwendung dieser Daten.

### 5.7.9. Error Correcting Codes

#### Ausgangslage

Wie können Daten möglichst Fehlerfrei gehalten werden bzw. wie können fehlerhafte Daten möglichst schnell korrigiert werden?

#### Lösungsansatz

Mit dem Checksum Ansatz kann das System relativ schnell erkennen ob Daten korrekt sind, jedoch nicht welcher Teil ungültig ist. Deshalb müssen zusätzlich Korrekturbits eingeführt werden, womit das System erkennen kann welcher Teil zerstört bzw. ungültig ist.

#### Schlussfolgerung

Speichere mit jeder Checksum möglichst viele Informationen, damit fehlerhafte Daten möglichst korrigiert werden können.

### 5.7.10. Shed Load

#### Ausgangslage

Wie können zu viele Anfragen an ein System bestmöglichst behandelt werden, ohne es in die Knie zu zwingen?

#### Lösungsansatz

Es sollen frühstmöglich begonnen werden Anfragen abzuweisen, falls die Gefahr besteht das ein System anfängt zu überlasten. Dabei ist es wichtig sich zu vergewissern, dass das System dies gegen aussen richtig kommuniziert.

#### Schlussfolgerung

Lehne gewisse Anfragen ab, um das System am laufen zu halten.

### 5.7.11. Final Handling

#### Ausgangslage

Braucht es einen eigen Mechanismus um besetzte Ressourcen frei zu geben, welche von irregulär beendeten Prozessen besetzt sind?

#### Lösungsansatz

Die einfachste Lösung wäre sich nicht um die Ressourcenfreigabe zu kümmern und falls vorhanden den Garbage Collector seine Arbeit machen lassen. Der bessere Ansatz ist jedoch zur Programmierzeit bereits einen Ressourcenfreigabemechanismus einzubauen, welcher bei normaler und irregulärer Beendigung von Prozessen zum Zuge kommt. Dies vereinfacht die Programmierung und stellt sicher das Ressourcen freigegeben wurden.

#### Schlussfolgerung

Integriere das freigeben von Ressourcen in den normalen Programmfluss, so dass auch nach einem Fehler und dessen Behandlung genügend Ressourcen zur Verfügung stehen.

### 5.7.12. Share the load

#### Ausgangslage

Das System soll grossen Workload bearbeiten können. Es besteht evtl. aus mehreren parallelen Elementen, z.B. in einem Cluster oder mit Multicore.

**Frage**

Wie kann die verfügbare 'processing power' vergrössert werden? Einfach Prozessoren hinzuzufügen erhöht die Komplexität. Auch muss man aufpassen, was mit Funktionalitäten in Mehrkernsystemen passiert. Wird dabei viel herum geschoben, so ergibt sich erheblichen Overload (Synchronisation etc.).

**Lösungsansatz**

Lasse gewisse Arbeit von anderen Prozessoren erledigen. Wähle Arbeiten aus, welche nur geringen Synchronisationsaufwand benötigen.

**5.7.13. Slow it down****Ausgangslage**

Gibt es keine gesetzten Limiten für Systemrequests, so kann das System im schlimmsten Fall mit Arbeit überladen werden, bis gar nichts mehr geht.

Das System kann im Grenzfall auch nicht auf menschliche Hilfe warten (s. Minimize Human Intervention)

**Frage**

Was soll das System tun, wenn die Anzahl Requests die Effizienz zu beeinträchtigen drohen? Ein Shutdown bringt nicht den gewünschten Erfolg, da das System ja nützliche Arbeit erledigen sollte.

**Lösungsansatz**

Es gibt verschiedene Mechanismen um das System vom Overload zu befreien. Einige sind restriktiver als andere. Nutze daher eine Art von Escalation, um immer stärker auf die Bremse treten zu können, falls ein Mechanismus noch nicht den gewünschten Effekt bringt.

**5.7.14. Shed work at periphery****Frage**

Wie kann man dem System, das mit Shed Load arbeitet, möglichst viel Arbeit abnehmen? Das belastete System sollte ja nicht noch mit zusätzlichem Arbeitsaufwand belastet werden.

**Lösungsansatz**

Versuche zu verworfene Requests so nahe der Systemgrenze wie möglich zu erkennen. Dies hilft dem Systemkern, sich auf die wirklich wichtigen Aufgaben konzentrieren zu können.

### 5.7.15. Finish work in progress

#### Ausgangslage

Requests können in verschiedener Art und Weise miteinander in Beziehung stehen. Z.B. kann es sein, dass Requests auf früheren aufbauen.

Ansätze wie Slow it down und Shed load sind zwar im System eingebaut, scheinen aber nicht viel zu nützen.

#### Frage

Welche Requests soll das System akzeptieren und welche verwerfen? Besteht ein solcher z.B. aus mehreren Sub-Requests, so will man sicher nicht den letzten davon verwerfen, da sonst das System schon bald wieder mit dem ganzen Requestssatz bombardiert wird

#### Lösungsansatz

Verarbeite zusammenhängende Requests (von denen der Super-Request schon verarbeitet wurde) und verwerfe solche, die neu ankommen.

### 5.7.16. Fresh work before stale

#### Ausgangslage

Es kommen mehr Requests ins System rein, als es bearbeiten kann. Man möchte QoS jedoch so hoch wie möglich halten. Der Benutzer ist in der Lage, Requests abzubrechen und neue zu starten (Beispiel: Webseiten-Requests). Das System hat die Fähigkeit, eingehende Requests in unterschiedliche Kategorien zu sortieren. Dies ermöglicht dem System, FINISH WORK IN PROGRESS (54) sowie SHED LOAD (49) anzuwenden.

#### Frage

Wie kann man sicherstellen, dass so viele Requests als möglich den bestmöglichen Service erhalten?

#### Lösungsansatz

Wenn Requests sehr lange dauern, gibt der Nutzer meist auf und bricht ihn ab. Dies kann dazu führen, dass das System noch mehr zu tun hat, wenn es beispielsweise den Request startet und erst dann merkt, dass dieser bereits von der Userseite her abgebrochen wurde. Wenn das System so viele Requests behandelt wie es nur kann, ist es dazu gezwungen, die Requests in einer Queue zu halten. Der einfachste Weg für eine Queue ist ein Buffer, welcher wie eine FiFo Queue funktioniert. Das Problem bei diesem Buffer ist jedoch, dass abgebrochene Requests erst dann entdeckt werden, wenn sie behandelt werden.

Requests können schnell behandelt werden, wenn ein Stack eingesetzt wird (LiFo-Queue). Dies ermöglicht dem System, mit dem aktuellsten Requests zu arbeiten. Dies

impliziert, dass es wahrscheinlicher ist, dass die Requests noch nicht abgebrochen wurden.

Wenn das System weiß, wie lange Requests warten bis sie abgebrochen werden, kann es besser entscheiden, welche Requests bearbeitet werden müssen. Es ist jedoch schwierig die Übersicht zu behalten, welche Requests bereits wie lange warten. Dies führt darüber hinaus auch zu mehr Overhead.

## 5.8. Fault Treatment Patterns

- Wird ein Error mithilfe von **Detection Patterns** gefunden, wird das System entweder durch
  - **Recovery Patterns** zurück in einen fehlerfreien Zustand überführt, oder der Error wird durch
  - **Mitigation Patterns** maskiert, sodass die Auswirkungen so klein wie möglich gehalten werden
- Nun kommen die **Fault Treatment Patterns** zum Einsatz; Es wird versucht, den Fault welcher den Error verursacht zu finden und zu korrigieren.

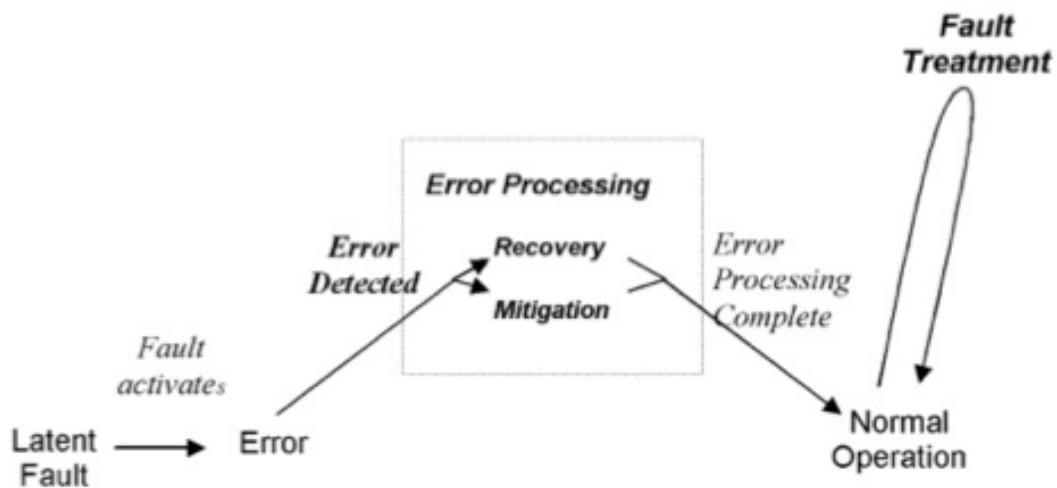


Abbildung 5.24.: fault treatment

Um einen Fault mithilfe der **Fault Treatment Patterns** zu eliminieren, werden folgende Schritte durchloffen:

- **Verification**

- Es wird geprüft, ob sich das System gemäss seiner Spezifikation verhält. Dies wird gemacht um zu prüfen, ob sich der Fault (immer noch) im System befindet.
- **Diagnosis**
  - Die Ursache des Fehlers wird untersucht. Es gilt, den Fault welcher zum Error führt, aufzuspüren und die genaue Erscheinungsform des Errors zu erforschen.
- **Correction**
  - Der Fehler wird aus dem System entfernt. (Sourcecode, System-Konfiguration etc.)
- **Verification #2**

## Prüfungsfragen

- Was ist das Ziel von Fault Treatment Patterns?
- Welche Schritte werden beim Einsatz von Fault Treatment Patterns durchloffen?

### 5.8.1. Let Sleeping Dogs Lie

#### Problem

Das System hat einen Error detektiert und ihn anschliessend korrigiert (Error Processing Patterns: Recovery oder Mitigation). Nun geht es darum, den Fault, welcher den Error aktiviert hat, zu korrigieren.

#### Lösung

- **Sollen alle Faults, welche vom System oder von einem Entwickler gefunden wird, behoben werden?**

Diese Frage lässt sich nicht generell beantworten! Einen Fehler in einem Software-System zu beheben verhält sich ähnlich wie ein medizinischer Eingriff. Der Eingriff an einem Patienten (die Software), birgt immer ein gewisses Risiko für Komplikationen. Deshalb muss vor einem solchen Eingriff geprüft werden, ob sich dieser überhaupt sinnvoll ist.

Dabei werden die Vorteile/Belohnung des Bugfixing den Kosten/Risiken gegenübergestellt. Folgende Fragen helfen bei der Entscheidung, ob ein bekannter Fault korrigiert werden sollte oder nicht:

- Welchen Risiken setzen wir das System aus, wenn der Fault nicht korrigiert wird?
- Wie stark wird das System durch die Fehlerbehebung verkompliziert?
- Wie gross ist die Wahrscheinlichkeit dass der Fehler korrigiert wird, ohne neue Fehler zu verursachen?

- Wie gross ist die Wahrscheinlichkeit, dass die Software neu installiert wird, ohne neue Fehler zu verursachen?
- Was kostest es uns neue Tests zu schreiben?
- Was kostet es uns den Fehler zu beheben und anschliessend erneut zu testen?
- Was kostet es uns eine verbesserte Software zu erstellen und anschliessend zu deployen?
- Wie teuer ist die Downtime des Systems, während gepatched wird?

Je nach dem entscheiden Sie sich den Fault zu suchen und zu beheben, oder Sie entscheiden sich den schlafenden Hund liegen zu lassen ;-)

### Prüfungsfragen

- Welche Faktoren stellen Sie einander gegenüber, wenn Sie entscheiden müssen ob ein neu entdeckter Fault korrigiert werden soll?
- Was ist mit **Let Sleeping Dogs Lie** gemeint? Erläutern Sie.

## 5.8.2. Reproducible Error

### Problem

Es ist ein Fehler aufgetreten. Durch Error-Mitigation konnte das System weiterlaufen. Nun geht es darum, den Fehler zu korrigieren. Zum Glück müssen wir nicht bei Null beginnen, sondern haben Informationen über den Fehler, welche vom System geloggt wurden.

Es ist wichtig, den eigentlichen Fehler zu behandeln und sich nicht mit einem eingebildeten Fehler zu versäumen. Das System ist nicht in einem statischen Zustand; seit dem Auftreten des Fehlers kann sich das System verändert haben. Vielleicht gab es unterdessen ein Software Update durch welches als Seiteneffekt der Fehler bereits behoben wurde. Ausserdem muss sichergestellt werden, dass derjenige Fault behandelt wird, der den konkreten Error oder Failure auch wirklich ausgelöst hat.

### Lösung

Löse auf kontrollierte Art und Weise den Fehler aus, um sicherzustellen, dass auch wirklich ein Fehler existiert. Dabei sollte das so beobachtete Verhalten mit der Systemspezifikation verglichen werden. Ein Fault wurde erst identifiziert wenn dieser mit bekannten Stimuli immer wieder reproduziert werden kann!

### Trade-off

Die Fehlersuche kann sehr zeitintensiv sein! Faults sind nicht nur im Quellcode zu suchen; Die Kombination aus Hardware, Software und Konfiguration muss untersucht werden.

### 5.8.3. Small Patches

#### Problem

Es gibt einen bekannten Fehler der zu beheben ist. Die Option LETTING SLEEPING DOGS LIE wurde ausgeschlossen. Es ist bekannt, wie der Fehler korrigiert werden kann.

Es geht nun darum, die Ausfallzeit des Systems und das Risiko für neu eingeführte Fehler zu minimieren. Je grösser das Update ist, je grösser der Codeumfang des Updates ist, um so grösser ist die Komplexität und umso mehr Möglichkeiten gibt es, einen neuen Fehler einzubauen. Ausserdem steigt der Aufwand für die Entwicklung und das Testen.

#### Lösung

Mache Updates so klein wie möglich. Dies hängt von den Tools und dem zu patchenden Bug ab.

#### Trade-off

Es ist nicht immer möglich, ein Patch in ein laufendes System einzuspielen. Sind keine "Hot Deployment"-Mechanismen vorhanden, muss das System neu gestartet werden.

Bei einem System ohne physischen Zugriff besteht die Gefahr, dass nach einem fehlerhaften Path nicht mehr darauf zugegriffen werden kann!

### 5.8.4. Root Cause Analysis

#### Problem

Der Fehler (fault) A wird zum Error (error) A und führt zum System Fehler (failure) A. Dieser führt zu Fehler B. Fehler B wird zum Error und führt zum System Fehler B. Dies kann unter Umständen immer so weiter gehen, dabei werden die einzelnen Fehler vom System geloggt. Das System hat inzwischen den Fehler behoben und arbeitet normal weiter. Jetzt ist es an der Zeit den Fehler zu analysieren und zu beheben, welcher den Error verursacht hat.

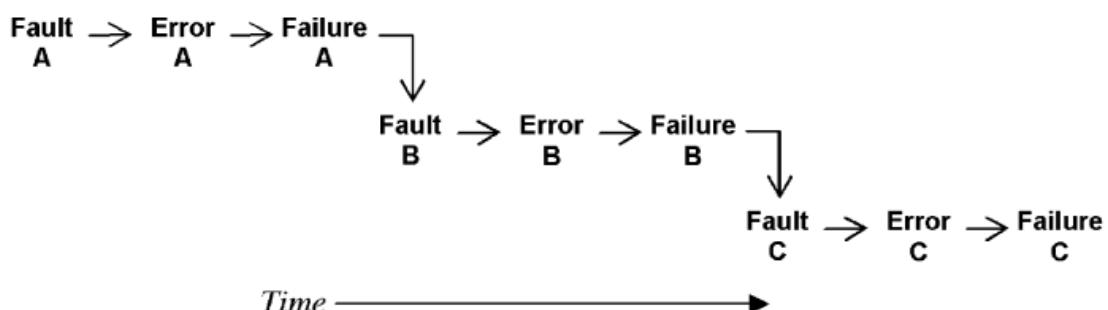


Abbildung 5.25.: failureSequence

Wie bereits angedeutet, kann der erkannte Error die Ursache für weitere Failure sein. Und umgekehrt, kann ein Error auch mehrere Faults als Vorgänger haben.

### Lösung

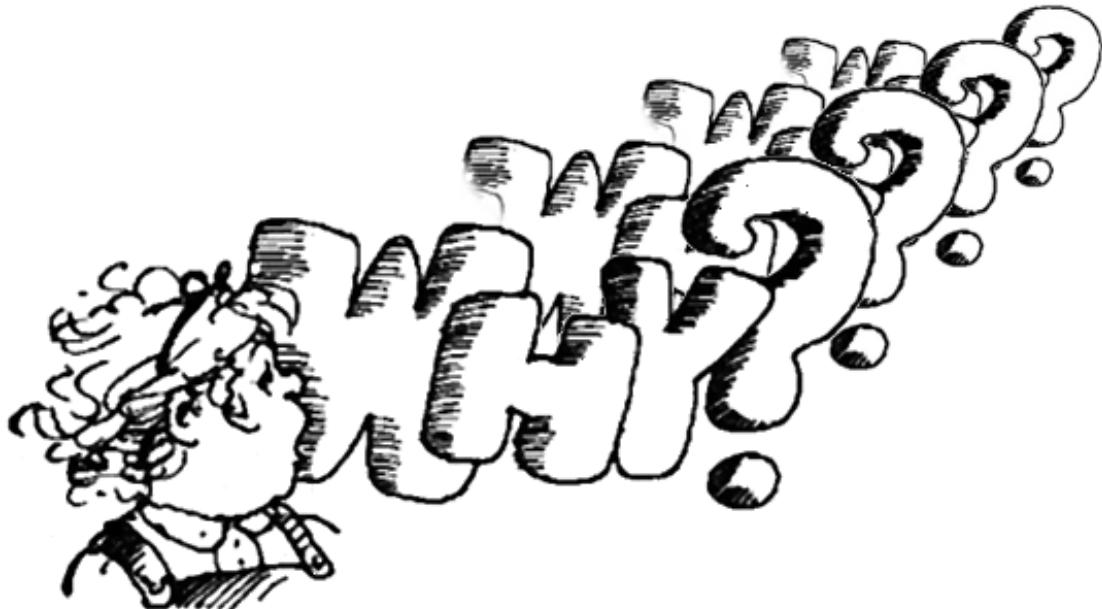


Abbildung 5.26.: why

Um einen Error zu beheben muss das Problem bei der Wurzel gelöst werden, also beim ursprünglichen Verursacher des Errors, auch root cause genannt. Dieser Fault muss als erstes korrigiert werden. Danach werden alle weiteren Faults nach und nach bis zum verursachenden Fault korrigiert. Um den sogenannten root cause zu finden, kann die 5 Warum-Fragetechnik verwendet werden.

- Why was the data record lost?
  - Because the transaction failed in the middle.
- Why did the transaction fail in the middle?
  - Because it ran out of memory.
- Why did it run out of memory?
  - Because there was no more memory available for allocation.
- Why was there no more memory available for allocation?

- Because the memory was inaccessible.
- Why was the memory inaccessible?
  - Because its owning task had terminated without releasing it.

Am Ende dieser Kette haben wir den ursprünglichen Auslöser gefunden. Es kann auch sein das es mehrere dieser Auslöser gibt.

### 5.8.5. Revise Procedure

#### Problem

Nach einem Failure wurden die Faults mit Root Cause Analysis gefunden und mit einem Update gelöst. Die Root Cause Analysis hat jedoch herausgefunden, dass die Failures durch menschliche Fehleinschätzungen bei der Programmierung oder der Wartung entstanden. Daher sollten Produktionsabläufe konzipiert werden, welche den Operators diese Fehler nicht ermöglichen.

#### Lösung

Experten wissen was sie eingeben müssen, weil diese das System sehr gut kennen (Entwickler). Jedoch ist nicht jeder Operator solch ein Experte:

- Falls ein „nichtexperte“ Fehlermeldungen falsch interpretiert, so kann dieser mit seinem Fix ein anderes Problem hervorrufen.
- Schlechte Instruktionsanweisungen führen zu willkürlichen Trial and Error und den daraus folgenden Miskonfigurationen.

Testen der Manuals, sowie verbessern der Manuals führt zu weniger menschlichen Fehlern.

#### Beispiel

- Checklisten im Flugverkehr

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

### Abbildungsverzeichnis

2.1.	Wrapper Facade Klassendiagramm . . . . .	9
2.2.	Interceptor UML-Diagramm . . . . .	11
2.3.	proactor uml diagram . . . . .	14
2.4.	Reactor UML-Diagramm . . . . .	18
2.5.	Reactor Sequenzdiagramm . . . . .	18
2.6.	Asynchronous Completion Token Klassendiagramm . . . . .	21
2.7.	Asynchronous Completion Token Sequenzdiagramm . . . . .	21
2.8.	Acceptor Connector Klassendiagramm . . . . .	24
2.9.	Acceptor Sequenzdiagramm . . . . .	24
2.10.	Connector Sequenzdiagramm . . . . .	25
2.11.	component configurator classdiagram . . . . .	27
2.12.	component configurator sequencediagram . . . . .	28
2.13.	component configurator statediagram . . . . .	28
2.14.	active object class diagram . . . . .	30
2.15.	active object sequence diagram . . . . .	31
2.16.	UML . . . . .	32
2.17.	SSD . . . . .	33
2.18.	Strategized Locking Class Diagram . . . . .	36
2.19.	Half-Sync/Half-Async Klassendiagramm . . . . .	39
2.20.	Half-Sync/Half-Async Sequencediagram . . . . .	40
2.21.	Leader/Follower Structure . . . . .	42
2.22.	Leader/Followers Sequence Diagram . . . . .	43
2.23.	Leader/Followers State Diagram . . . . .	44
4.1.	Authorization Klassendiagramm . . . . .	47
4.2.	Basic Role Based Access Control Klassendiagramm . . . . .	49
4.3.	RBAC mit Composite, Admins & Abstract Session . . . . .	49
4.4.	Multilevel Security Klassendiagramm . . . . .	52
4.5.	Reference Monitor - Klassendiagramm . . . . .	54
4.6.	Reference Monitor - Sequenzdiagramm [Sch+06] . . . . .	54
4.7.	Generischer Ansatz von I&A “Using functions” [Sch+06] . . . . .	58

4.8.	Security Session: Schematischer Aufbau [Sch+06] . . . . .	67
4.9.	Security Session: Interaktion der verschiedenen Akteure . . . . .	68
4.10.	Packet Filter Firewall Sequenzdiagramm . . . . .	71
4.11.	Stateful Firewall: Schematischer Aufbau . . . . .	74
4.12.	Stateful Firewall: Ablauf [Sch+06] . . . . .	75
4.13.	Information Obscurity als letzte Sicherheitsmaßnahme . . . . .	76
4.14.	Komponenten des Secure Channels Patterns [Sch+06] . . . . .	80
4.15.	Aushandeln eines Session Keys zur sicheren Kommunikation via SSL . . . . .	81
4.16.	Strukturerller Aufbau Protection Reverse Proxy . . . . .	83
4.17.	Strukturerller Aufbau Integration Reverse Proxy . . . . .	86
4.18.	Strukturerller Aufbau Front Door . . . . .	88
4.19.	Authenticator: Schematischer Aufbau . . . . .	90
4.20.	Behandlung eines Zugriffs mit Controlled Object Monitor [Sch+06] . . . . .	94
5.1.	Fault->Error->Failure Dependency . . . . .	97
5.2.	Minimale Anzahl an Komponenten um Failures auszugleichen . . . . .	98
5.3.	Downtime per year . . . . .	99
5.4.	Performance . . . . .	100
5.5.	introduction four phases of fault tolerance . . . . .	104
5.6.	unitsOfMitigation . . . . .	106
5.7.	escalation . . . . .	109
5.8.	MinimizeHumanIntervention . . . . .	114
5.9.	ResponsibilitiesList . . . . .	115
5.10.	U11 3 FaultObserver 2013 . . . . .	118
5.11.	detection concepts . . . . .	120
5.12.	Fault Error Failure Dependency . . . . .	121
5.13.	pattern thumbnails . . . . .	122
5.14.	pattern map . . . . .	123
5.15.	leaky bucket counter . . . . .	126
5.16.	RecoveryPatterns Dependency . . . . .	127
5.17.	RecoveryPatterns . . . . .	128
5.18.	MitigationPatterns Dependecy . . . . .	133
5.19.	MitigationPatterns . . . . .	134
5.20.	DeferrableWork . . . . .	137
5.21.	ReassessOverloadDecision . . . . .	138
5.22.	EquitableResourceAllocation . . . . .	139
5.23.	QueueForResources . . . . .	140
5.24.	fault treatment . . . . .	146
5.25.	failureSequence . . . . .	149
5.26.	why . . . . .	150

## Tabellenverzeichnis

4.1. I&A Requirements: Funktionale Anforderungen . . . . .	59
4.2. I&A Requirements: Nichtfunktionale Anforderungen . . . . .	60
4.3. Access Control Requirements Requirements: Funktionale Anforderungen . .	62
4.4. Access Control Requirements: Nichtfunktionale Anforderungen . . . . .	62

## Quellcodeverzeichnis

2.1. Condition für eine Plattformunterscheidung . . . . .	8
---	---

## 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).
- [wikc] wikipedia.org. *Denail Of Service*. URL: [http://de.wikipedia.org/wiki/Denail\\_of\\_Service](http://de.wikipedia.org/wiki/Denail_of_Service) (besucht am 14.04.2013).



## Anhang C **Glossar**

### *ACL*

Access Control List; eine Liste mit Zugriffsregeln für eine bestimmte Resource. 53

### *CRUD*

CRUD steht als Abkürzung für *Create, Read, Update and Delete* und ist damit ein Synonym für die grundlegenden Mutationsoperationen von Informationen.. 71

### *Hardening*

Beim Hardening werden die Funktionen eines Systems soweit reduziert, dass nur noch die eigentliche Hauptfunktionalität übrig bleibt. Bspw. werden bei einem Webserver alle Ports ausser Port 80 geschlossen.. 82

### *MTTF*

Mean Time To Failure. 98–100, 103

### *MTTR*

Failures in Time. 98

### *MTTR*

Mean Time between Failures. 98

### *MTTR*

Mean Time To Recover. 98–100, 103

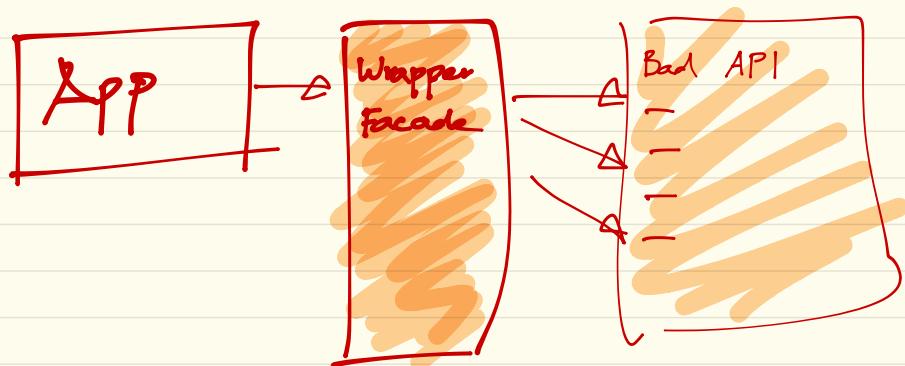
### *RBAC*

Role Based Access Control; Siehe Abschnitt 4.1.2. 56, 57

## Anhang D **Workshops**

Handschriftliche Notizen von Manuel Alabor.

## 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! ⚡

↳ 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:  $\frac{\# \text{ Failures}}{1 \cdot 10^3 h}$   $\Rightarrow$  Failures in Time

$\Rightarrow$  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  $\Rightarrow$  Byzantinische Generäle zur Abstimmung

# Architekturen patterns 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: simulös,  
 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

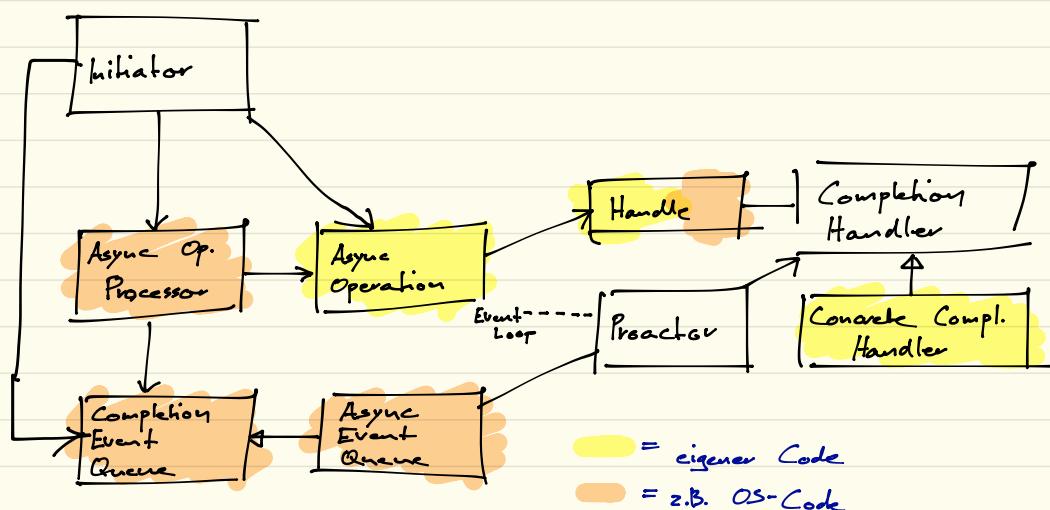
### Escalation

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)

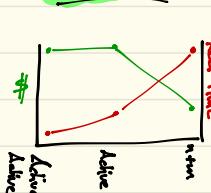
## Fault Tolerant Systems: Redundancy

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

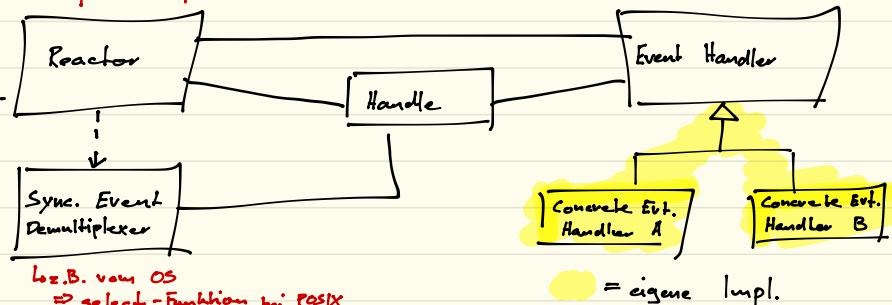
### Alternativen?

Früher: Main-Event-Loop

Gutes Beispiel?  
Multiplexing auf  
einem CPU-Core

Command Processor  
vs  
Reactor

### Event-Loop



- ① Ein schlechter Handler kann ganzen Reactor blockieren
- ② Contextswitching entfällt, da alles im gleichen Thread

Reactor looppt über registrierte Events, und führt entsprechende Handler aus

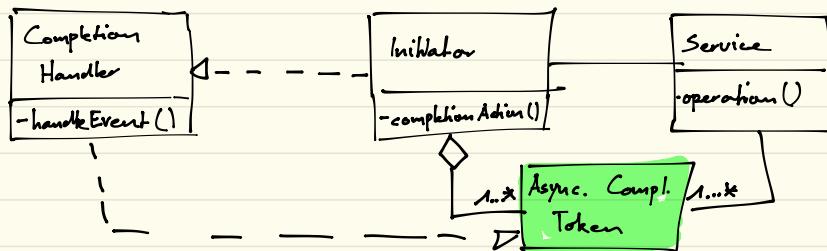
Command Processor führt lediglich Code aus, ohne "Events"

Was tun gegen Freeze?

Evtl. auslagern in eigene Threads

## Asynchronous Completion Token

26.03.2013



### Idee

Zustandsloses Dienst einfach mit einem Zustand verschen  
Token kann "Alles" sein; Pointer, Wert, Funktion... ⇒ kann für Schabernack missbraucht werden

### Token Passing

Ruft ein Service einen weiteren Service auf, kann das Token weitergereicht werden.

### Beispiel

- HTTP-Cookie ist ein "ACT" ⇒ Verschlüsselung & Signieren
- Starbucks Card

## Minimize Human Intervention

### Idee

Fehlerquelle "Mensch" minimieren

### Situationen

- Mensch vergisst etwas zu tun ⇒ unterstützen



- Mensch versucht etwas unerlaubtes/unerwartetes zu tun

⇒ verhindern, dass Mensch unerwartetes tun kann (z.B. UI-Blocking während Verarbeitung)

### Weiteres

- System soll versuchen, Fehler selber zu lösen (keine Popups, ...)
- Gut für unerfahrene Benutzer

### Beispiel

Zur HB Blackout: Kabel wurde durchtrennt, Operator hat fälschlicherweise Reparatur eingeleitet, austausch von

## Someone in Charge

### Simple

Immer ist eine Komponente für den Fehler verantwortlich, resp. dessen Behebung ⇒ Falls nicht befähigt: Escalation  
z.B. wichtig in verteilten Systemen

# Software Updates

03.04.2015

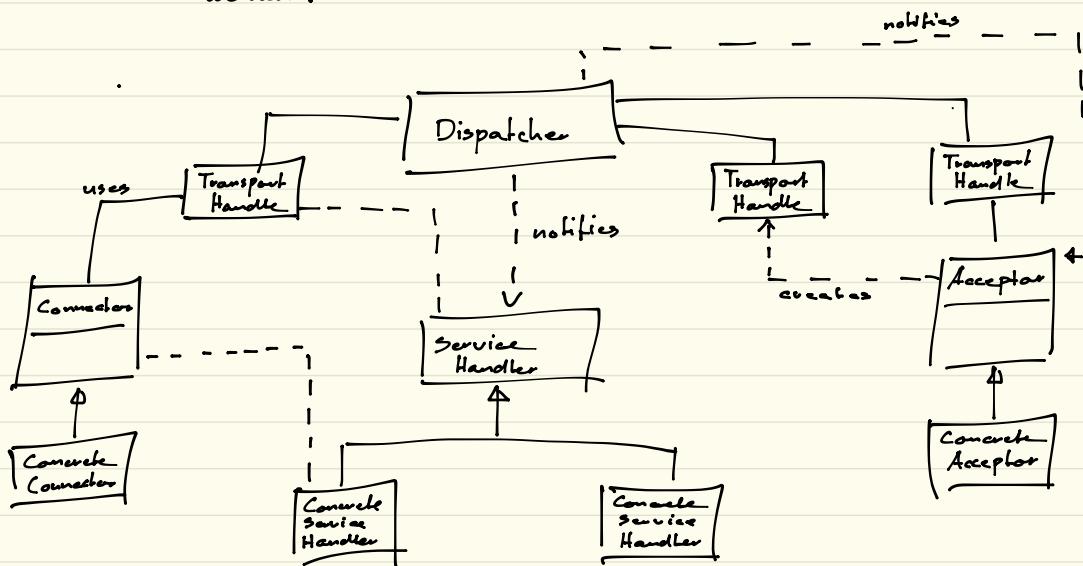
- Mehrere Tracks während Update
- Grundsätzlich einfach nicht vergessen beim Konzept :-)
- Beispiel Erlang: Fix in Sprache verankert mit "CodeChange"-Hook pro Modul
- Hot Swap, Planned Downtime, Live Patching

## Acceptor-Connector

What?

Wie können Nodes in einem verteilten Netz verbunden werden?

Wie kann das Kommunikationsprotokoll transparent ausgetauscht werden?



## Detection Patterns

Fault Tolerant Systems

Fault → Error → Failure

A-Priori Wissen Wissen, um Fehler im Voraus aufgrund von bestehender Logik zu finden  
entl. kann ein Programm auch selber lernen, und so dieses Wissen zu erarbeiten

Fault Correlation Komponente im System weiß, welcher Error/Fault/Error welche Behandlung benötigt (Auswertung von Logs, etc.)  
⇒ So wenig Ressourcen wie möglich verschlingen  
⇒ Kategorisierung von Fehlern um Behandlungen zusammenzufassen  
⇒ Beispiel: try/catch - Block mit versch. Exceptions = Kategorisierung  
Error Containment Barrier ⇒ Unit of Mitigation ⇒ eingeschränken von Fehlern im System  
↳ "Quarantine für Fehler"

Riding over Transients Unterscheidung: Seltene & wiederkehrende Fehler  
↓  
transient

⇒ Frustrationsgrenze für wiederkehrende, und nicht sehr tragische Fehler  
⇒ extrem abhängig von Requirements!

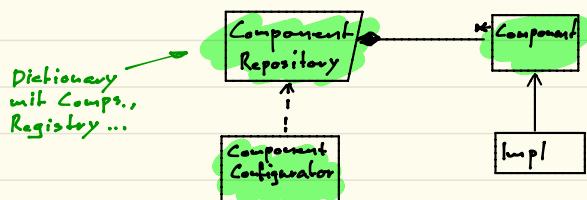
Leaky Bucket Counter Zähler wie oft welche Fehlerkategorie aufgetreten ist.  
Wird über gewisse Zeitspanne kein Fehler detektiert, kann Zähler dekrementiert werden.



⇒ Erkennung ob Fehler transient oder nicht

## Component Configurator

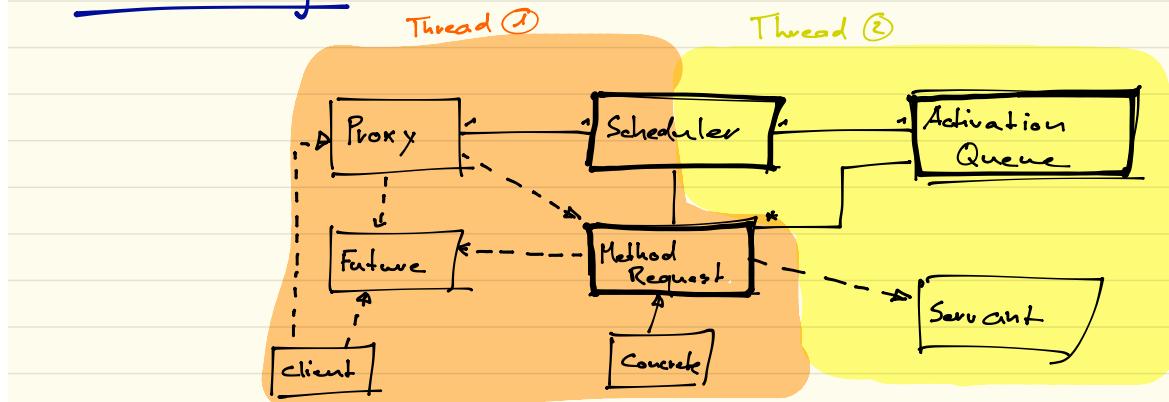
- Idee              In Komponenten aufgeteilte App zur Laufzeit einrichten /  
 Bestandteile      - Allgemeines Component-Interface (Load, Start, Stop) laden/stoppen  
                   - Abhängigkeitsinfos von Components



Beispiel          Plugin-System => dynamisches Nachladen von Code,  
                   welcher nicht unbedingt immer  
                   benötigt wird

## Active Object

23.04.2013



Methodeaufruf wird entkoppelt; es gibt aber kein Callback sondern ein Future wird gepölt um zu sehen, ob nun alles bereit ist  
 Kellner (Proxy) nimmt Bestellung auf; Schreibt diese auf Zettel (Future) und gibt sie dem Koch (Active Object). Koch arbeitet selbstständig ab (Activation Queue).

Beispiel

- Methodeaufrufe werden vom effektiven Aufruf entkoppelt
- Non-Blocking

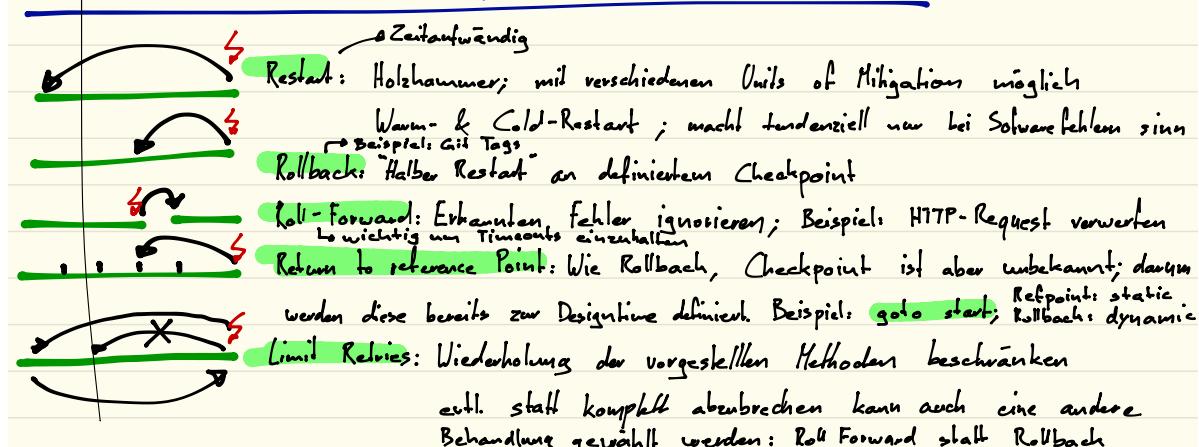
Contra

- Overhead
- Schwierig zu debuggen / testen

Ergänzungen

Timeouts für "fikt." Requests.

## Restart, Rollback, Roll-Forward, Return to Ref.-Point



## Error Mitigation Patterns

30.04.2013

Idee

Fehler an Ort und Stelle beheben

Part 1: Behandlung von Systemoverload: Overload Empires**Overload Toolboxes** Set aus Tools um Overload-Probleme generisch zu Handeln**Dearable Work**

"Unwichtige" Routine-Aufgaben auf Zeit mit wenig Last replanen

Dies geschieht dynamisch zur Laufzeit; "unplanned"

Beispiel: Backup auf später verschieben da CPU/IO zum Arbeiten nötig ist

**Reassesses Overload Decision**

Monitoring von getroffenen Overload-Behandlungs-Massnahmen

&amp; Rückgängig machen

**Equitable Resource Allocation**

Requests aufhand des Zugriffstyps poolen: DB, Cache, etc.

↳ Verkürzt Wartezeiten ↳ Priority Inversion: Tiefe Prio belegt Resource,

Hohe Prio kommt und muss warten

**Queue for Resources**

Shall Requests zu verarbeiten bei Overload, die Requests in Queue stellen

↳ Maschinelle Requests: FIFO Konschl. Requests: LIFO

Beispiel: OS-Kernel hat Queue für IP-Requests welche noch nicht vom Listener "behandelt" wurde

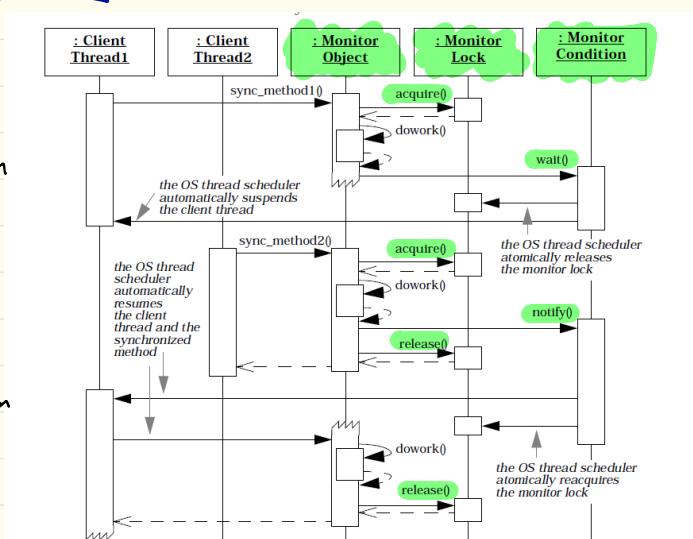
## Monitor Object

Idee

Objekte vor parallelen Zugriffen schützen

Gefahr:

- Deadlocks falls Locks nicht wieder freigegeben werden



## Fault Tolerant Systems

14.05.2013

Share the load System mit vielen Anfragen könnte überladen werden

Idee: Statt noch mehr zu parallelisieren (mehr Overhead), gezieltes parallelisieren von Aufgaben mit geringen Synchronisationskosten

Shed Work in Periphery Request, welche potentiell nicht bearbeitet werden können schon so früh wie möglich verwerfen  $\Rightarrow$  Noise Reduction

Kriterien: Prio? Nec? Widerstand?

Beispiel: Load Balancer / Firewall blockt bereits

Slow it down Strategien um "unwichtige" Prozesse zurückzuschreiben

Finish work in progress Zusammengehörende Requests priorisieren, um overall Responsiveness zu verbessern

• Neue Requests können wichtige Ressourcen belegen

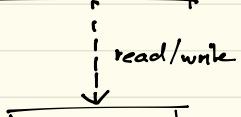
- Fast fertige Prozesse müssen nicht noch länger warten

Fresh work before stale Statt mit Queue jetzt mit Stack arbeiten

## Half-Sync / Half-Async

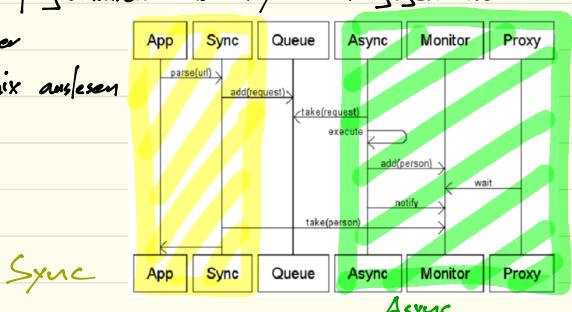
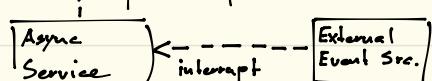


Idee: Wie kann das Beste aus der synchronen und asynchronen Welt verwendet werden



Umsetzung: Als Entwickler programmiert man synchron gegen den Queuing-Layer

Beispiel: Sockets auf Unix auslesen



## Fault Treatment Patterns

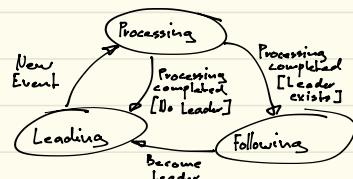
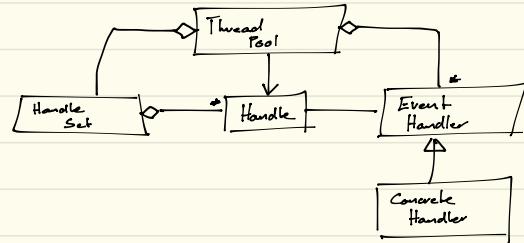


21.05.2013

1. Idee Zugrundeliegender Fehler korrigieren (oder eben nicht)  
 ① Let Sleeping Dogs lie Bewusst bekannte Fehler nicht korrigieren  $\Rightarrow$  Kosten/Eintrag abwagen, wie gross ist das Risiko für den Eingriff  
 ② Reproducible Error Wenn Eingriffen wird, muss durch Reproduktion des Fehlers sichergestellt werden, dass das Problem jetzt behoben ist.  
 ③ Small Patches Patches so klein wie möglich halten  
 ④ Root Cause Analysis Fault zuverfolgen, statt nur lokale Symptombehandlung vornehmen  
 ⑤ Revise Procedure Troubleshooting-Guide für Benutzer überarbeiten "Wenn A, versuch B. OK? Nein, dann C. OK? ...."

## Leader/Follower

- Ablauf Leader nimmt Request entgegen, bestimmt neuen Leader und verarbeitet Request. Sobald fertig, reicht er sich wieder in Queue/Pool ein  
 Vorteil Weniger Context-Switches da Thread direkt weiterarbeiten kann statt Task weiterzugeben.



Implementation: Monitor um accept()-Systemcall  $\Rightarrow$  Thread mit Monitor-Lock ist (möglich) Leader

Soll ein Thread nach Processing "hinten oder vorne" einschalten?  
 $\hookrightarrow$  vorne, da wieder weniger Context-Switches nötig sind