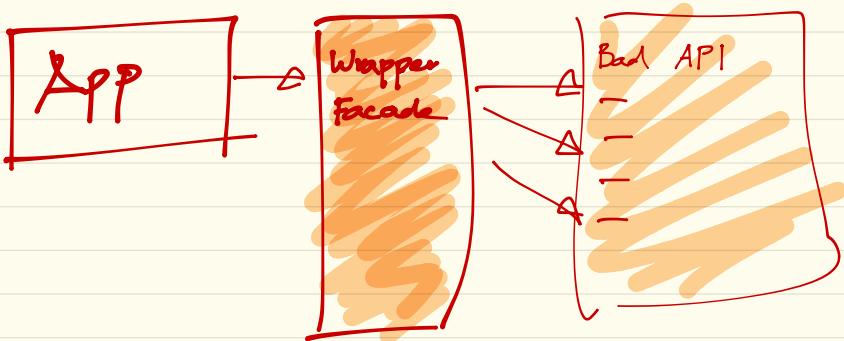


## Wrapper Facade



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

Anmerkungen P. Sommerlad:

- Error-Handling ist wichtig  $\Rightarrow$  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 Tolerance Systems

Introduction: Zusammenhang Fault, Error & Failure

Fault: Bug, Ursache

Error: Zustand

Failure Effektives Problem

↳ Zu vermeidendes Problem

- Failure definieren sich im Normalfall durch Abweichung von der Spec

- Unterschiedliche Faults können zu gleichen Errors/Failures führen

- Coverage: Wahrscheinlichkeit dass sich ein System innerhalb gegebener Zeit wieder erholen kann:  
Mean Time To Failure }  
Mean Time To Recover + } Mean Time Between Failure

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

- MTI:  $\frac{\# \text{ Failures}}{1 \cdot 10^5 \text{ h}}$  ⇒ Failures in Time

⇒ Stichwort: Server-Zuverlässigkeit

Fault Silent: Bei Fehler übernimmt automatisch andere Komponente

Fault Consistency: Man muss herausfinden welche Systemkomponenten fehlerhaft sind

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

# Architekturpatterns

## Fault Tolerance

"Allgemeingütige" Patterns für gesamte Architekturen

### Units of Mitigation

Problem: Fehler soll nicht gesamtes System beeinträchtigen

⇒ Beschränkung auf "Unit", bspw. try-catch-Block

⇒ Unitgröße ist essentiell (zu gross: sinnlos,  
zu klein: Code-Aufwand)

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

Beispiele für Units: → Funktionsgruppen

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

Fehler bleiben Unit-spezifisch ⇒ Erkennung & Behandlung 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: Detekte 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 (Versch. Verrechnungen des selben Wertes), macht der Wert Sinn?

⇒ Datendesign für einfache Prüfung auslegen

Korrigieren: Direkt vom Programm

Repeat Finden, um Korrektheit zu prüfen

## Escalation

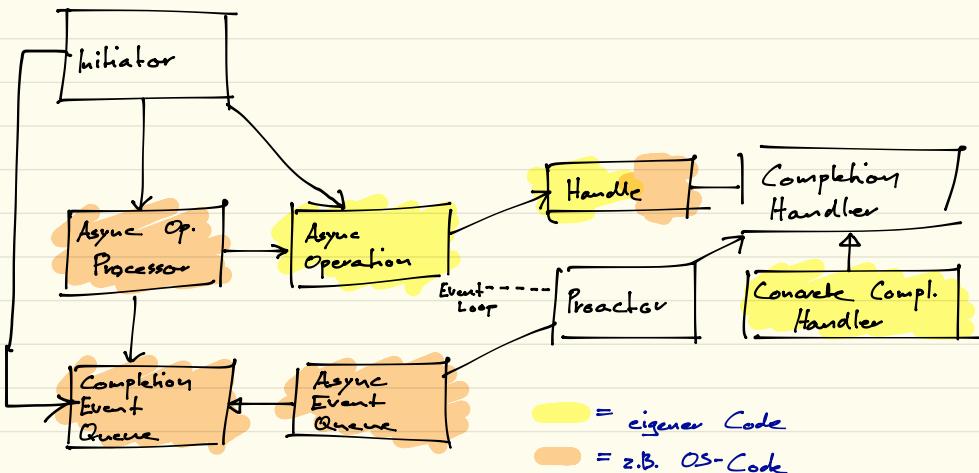
Problem: Was passiert, wenn Fehler immer wieder auftritt?

Lösung: Fehler als externe Instanz nach aussen weitergeben

- bspw. Operator

- bspw. Fehlerbehandlungsdienst.

## Proactor



⇒ NSOperationQueue? 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

Knickpunkte: Asynchrone Entwicklung vs. sequentiell

Vorteile:

- Parallelisierung I/O ↔ Completion Handler

- Tendenziell robuster, da entkoppelt (Async hält)

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