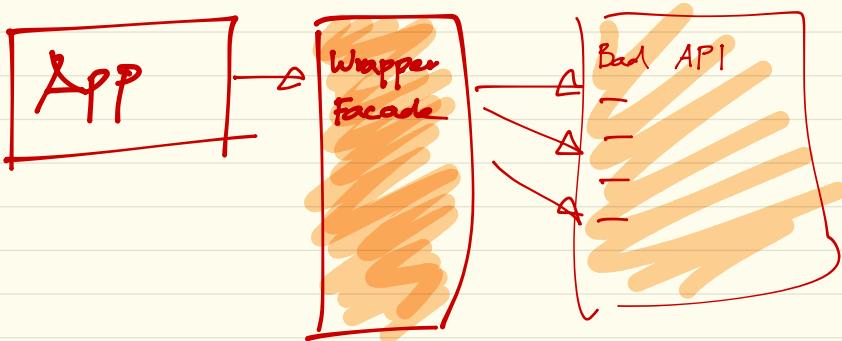


Wrapper Facade



- 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. Sonnenblod:

- 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

Fail Silent: Bei Fehler übernimmt automatisch andere Komponente

Fail Consistency: Man muss herausfinden welche Systemkomponenten fehlerhaft sind

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

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

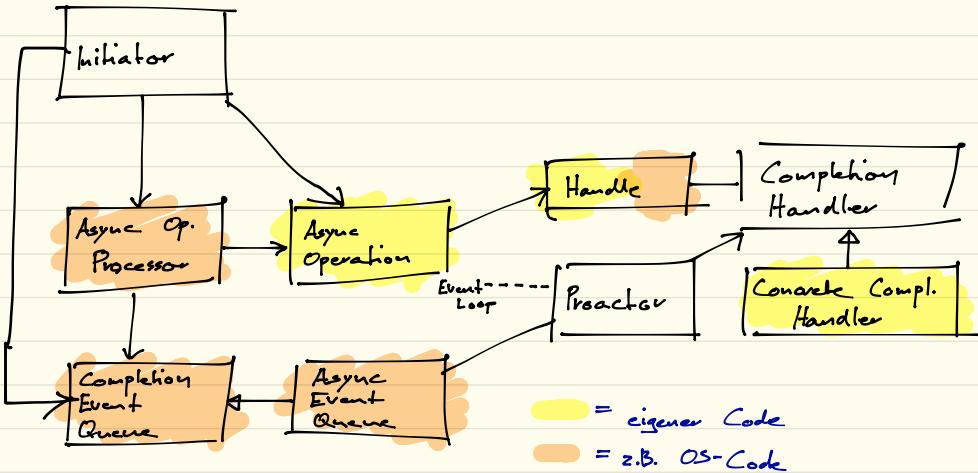
Problem: Was passiert, wenn Fehler immer wieder auftritt?

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)

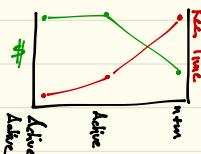
Fault Tolerant Systems: Redundancy

Redundanz sagt nicht, dass "Not-system" identisch sein muss... 15.03.2013

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 Abwagen, optimieren

Recovery Blocks

Gleiche Aufgaben von verschiedenen Implementationen ausführen lassen, Ergebnis vergleichen & bestes wählen

\Rightarrow n-Version-Programmierung

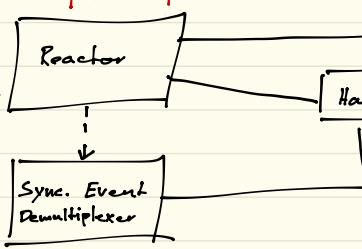
Beispiel: Verschiedene Sort-Algos



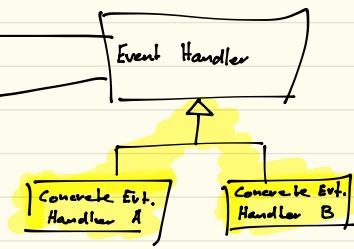
Reactor

Alternativen?

Früher: Main-Event-Loop



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



Gutes Beispiel?

Multiplexing auf einzelnen CPU-Core

↳ z.B. vom OS
 \Rightarrow select-Funktion bei Posix

= eigene Impl.

Command Processor Reactor loop über registrierte Events, und führt entsprechende Handler aus

VS
Reactor
Command Processor führt lediglich Code aus, ohne "Events"

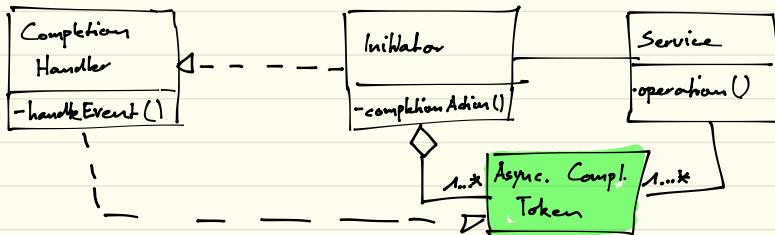
Was tun gegen Freeze?

Evtl. auslagern in eigene Threads

Asynchronous Completion Token

(ACT)

26.03.2015



Idee

Zustandslosen Dienst einfach mit einem Zustand versehen
Token kann "Alles" sein, Pointer, Wert, Funktion... \Rightarrow kann für Schabernack missbraucht werden

Token Passing

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

Beispiel

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

Minimize Human Intervention

Idee

Fehlerquelle "Mensch" minimieren

Situationen

- Logging wichtig
- Mensch vergisst etwas zu tun \Rightarrow unterstützen
 - Mensch versucht etwas unerwartetes/unerwartetes zu tun
 \Rightarrow 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 vom

Someone in Charge

Simple

Immer ist eine Komponente für den Fehler verantwortlich, resp. dessen Behebung \Rightarrow falls nicht befähigt: Escalation
z.B. wichtig in kritischen Systemen

Software Updates

03.04.2015

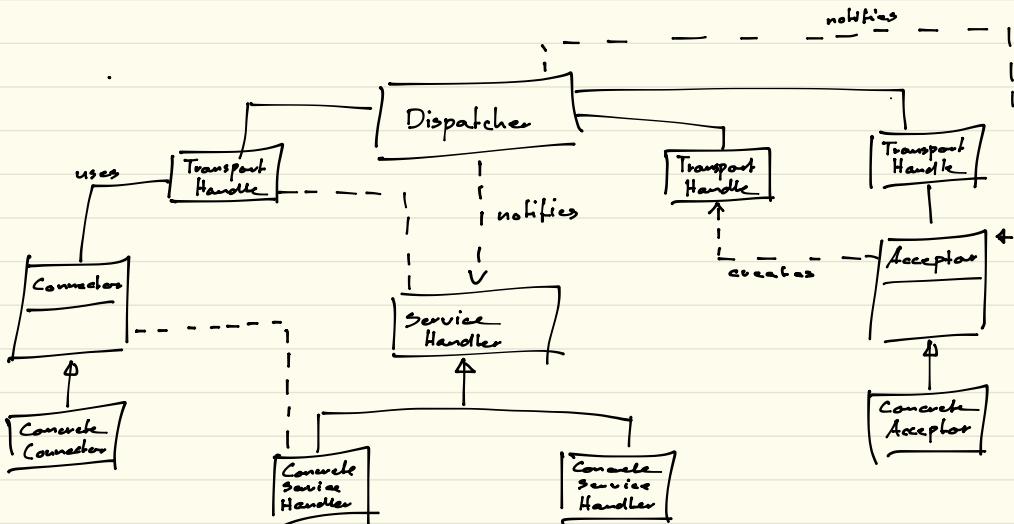
- Mehrere Tracks während Update
- Grundsätzlich einfach nicht vergessen beim Konzept :-)
- Beispiel Erlang: Fix in Sprache verankert mit "CodeChange"-Hook pro Modul
- Hot Swap, Planned Downtime, Live Patching

Acceptor-Connector

What's

Wie können Nodes in einem verteilten Netz verbunden werden?

Wie kann das Kommunikationsprotokoll transparent ausgetauscht werden?



Fault Tolerant Systems

Fault \rightarrow Error \rightarrow Failure

Detection Patterns

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.)

\Rightarrow So wenig Ressourcen wie möglich verschlingen

\Rightarrow Kategorisierung von Fehlern um Behandlungen zusammenzufassen

\Rightarrow Beispiel: try/catch-Block mit versch. Exceptions = Kategorisierung

Error Containment Barrier \Leftrightarrow Unit of Mitigation \Rightarrow eingrenzen von Fehlern im System
↳ "Quarantine für Fehler"

Riding over Transients Unterscheidung: Seltene & wiederkehrende Fehler

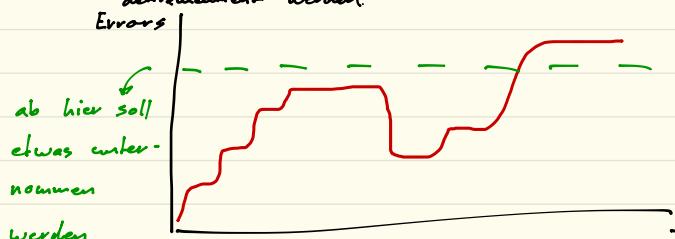
\Downarrow
transient

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

Leaky Bucket Counter

Zähler wie oft welche Fehlerkategorie aufgetreten ist.

Wird über gewisse Zeitspanne kein Fehler detektiert, kann Zähler deaktiviert werden



\Rightarrow Erkennung ob Fehler transient oder nicht

Component Configurator

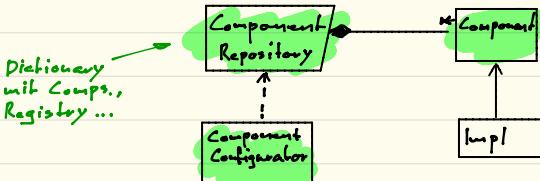
Idee

In Komponenten aufgeteilte App zur Laufzeit konfigurieren /
laden/stoppen

Bestandteile

- Allgemeines Component-Interface (Load, Start, Stop)

- Abhängigkeitsinfos von Components

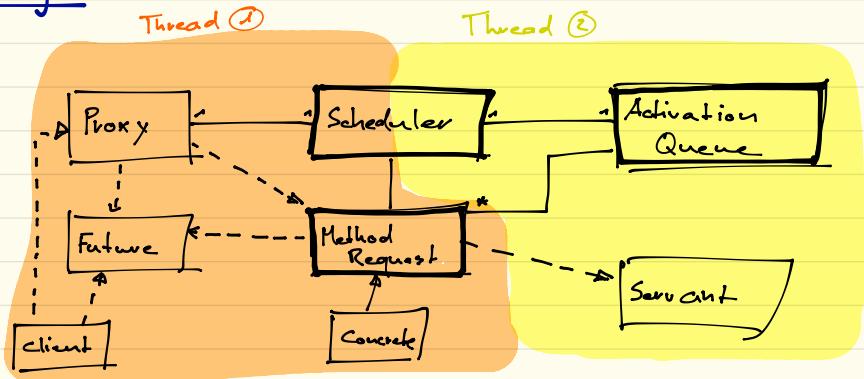


Beispiel

Plugin-System \Rightarrow 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 gepöllt 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 "tote" Requests.

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

Zeitaufwändig

Restart: Holzhammer; mit verschiedenen Units of Migrations möglich

Warm- & Cold-Restart; macht tendenziell nur bei Softwarefehlern sinn

Rollback: Halber Restart an definiertem Checkpoint

Roll-Forward: Erkannten Fehler ignorieren; Beispiel: HTTP-Request verwerfen

Return to reference Point: Wie Rollback, Checkpoint ist aber unbekannt; darum

werden diese bereits zur Designtime definiert. Beispiel: **goto start**; Rollback: static

Limit Retries: Wiederholung der vorgestellten Methoden beschränken

entl. statt komplett abbrechen kann auch eine andere Behandlung gewählt werden: Roll Forward statt Rollback

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 & Rückgängig machen

Equitable Resource Allocation Requests anhand des Zugriffstyps poolen: DB, Cache, etc.

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

Höhe Prio kommt und muss warten

Queue for Resources Shall Requests zu verarbeiten bei Overload, die Requests in Queue sammeln

↳ Maschinelle Requests: FIFO Mensch. Requests: LIFO

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

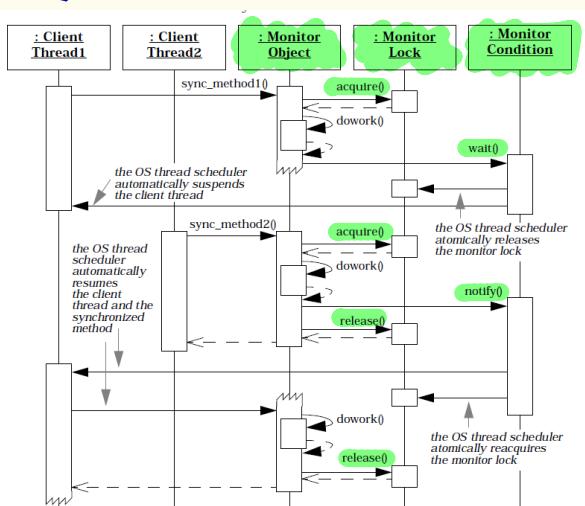
Monitor Object

Idee

Objekte vor parallelen Zugriffen schützen

Gefahr:

- Deadlocks falls Locks nicht wieder freigegeben werden



Fault Tolerant Systems

19.05.2013

Parallelität

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 im

Periphery

Request, welche potentiell nicht bearbeitet werden können schon so früh wie möglich verwerten \Rightarrow Noise Reduction

Kriterien: Prio? Neu?/Wiederholend?

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

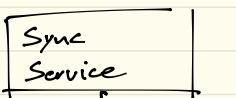
Laufende Requests können wichtige Ressourcen belegen

- fast fertige Prozesse müssen nicht noch länger warten

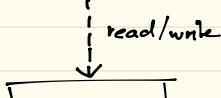
Fresh work before stale

Statt mit Queue jetzt mit Stack arbeiten

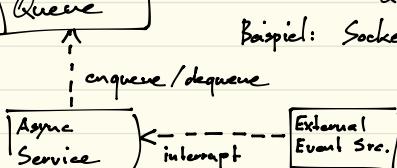
Half-Sync / Half-Async



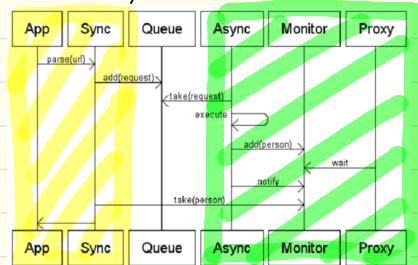
Idee: Wie kann das Beste aus der synchronen und asynchronen Welt verwendet werden



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



Beispiel: Sockets auf Unix auslesen



Fault Treatment Patterns



21.05.2013

Idee

Zugrundeliegender Fehler korrigieren (oder eben nicht)

- ① Let Sleeping Dogs lie
Bewusst bekannte Fehler nicht korrigieren \Rightarrow Kosten/Ertrag abwägen, 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 zuverlässig verfolgen, statt nur lokale Symptombehandlung vornehmen
- ⑤ Revise Procedure
Troubleshooting-Guide für Benutzer überarbeiten "Wenn A, versuch B. Ok? Nein, dann C. Ok?"

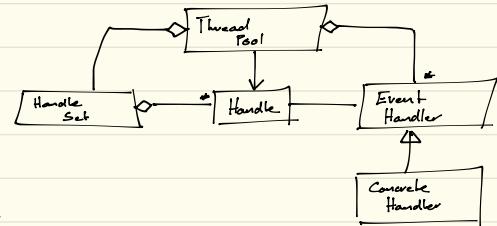
Leader/Follower

Ablauf

Leader nimmt Request entgegen, bestimmt neuen Leader und verarbeitet Request.

Sobald fertig, reicht er sich wieder in Queue/Pool ein. Weniger Context-Switches da Thread direkt weiterleiten kann statt Task weiterzugeben.

Vorteil



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

Soll ein Thread nach Processing "hinten oder vorne" eintreten?

\hookrightarrow vorne, da wieder weniger Context-Switches nötig sind

