

# Architektur von Informationssystemen

Hochschule für angewandte Wissenschaften

Sommersemester 2018

Nils Löwe / [nils@loewe.io](mailto:nils@loewe.io) / @NilsLoewe

# 3. Praktikum

Praktikum 3: Architekturentwurf

Fragen?

# 4. Praktikum

Frage: Alle Präsentationen in der 12. Vorlesungswoche?

# Wiederholung Architekturmuster

Ein Architekturmuster beschreibt eine bewährte Lösung für ein wiederholt auftretendes Entwurfsproblem

*(Effektive Softwarearchitekturen)*

Chaos zu Struktur / Mud-to-structure

## Layers

Pipes und Filter

Blackboard

Domain-driven Design

## Layers

Das Layers-Muster trennt eine Architektur in verschiedene Schichten, von denen jede eine Unteraufgabe auf einer bestimmten Abstraktionsebene realisiert.

Layers

## Beispiel: ISO/OSI-Referenzmodell

Netzwerk-Protokolle sind wahrscheinlich die bekanntesten Beispiele für geschichtete Architekturen. Das ISO/OSI-Referenzmodell teilt Netzwerk-Protokolle in 7 Schichten auf, von denen jede Schicht für eine bestimmte Aufgabe zuständig ist:



Layers

## Vorteile

- Wiederverwendung und Austauschbarkeit von Schichten
- Unterstützung von Standards
- Einkapselung von Abhängigkeiten

Layers

## Nachteile

- Geringere Effizienz
- Mehrfache Arbeit (z.B. Fehlerkorrektur)
- Schwierigkeit, die richtige Anzahl Schichten zu bestimmen

## Layers

Bekannte Einsatzgebiete:

- Application Programmer Interfaces (APIs)
- Datenbanken
- Betriebssysteme
- Kommunikation...

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

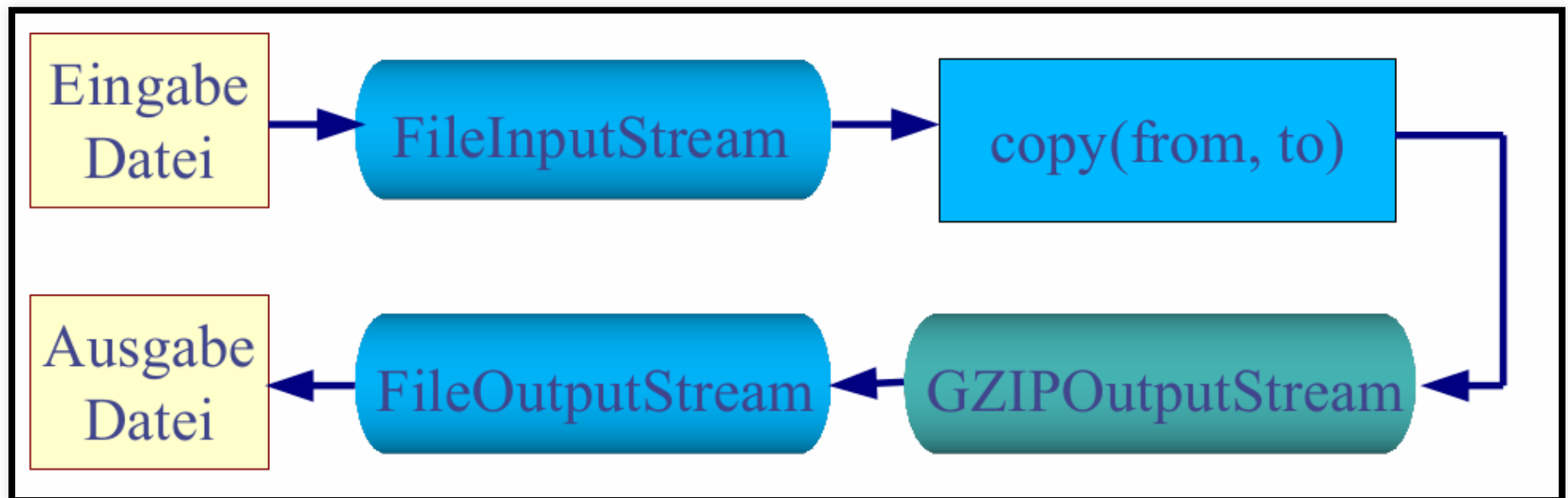
Blackboard

Domain-driven Design

## Pipes and Filters

Eine Pipes-and-Filter Architektur eignet sich für Systeme, die Datenströme verarbeiten.

## Pipes and Filters



## Pipes and Filters

Das Pipes-and-Filters Muster strukturiert Systeme, in dem Kontext „Verarbeitung von Datenströmen“. Die Verarbeitungsschritte werden in Filter eingekapselt und lassen sich so beliebig anordnen und getrennt voneinander entwickeln.

## Pipes and Filters

Der Kommandointerpreter sowie viele Werkzeuge des Unix Betriebssystems sind nach dem Pipes-and- Filter Muster gebaut. Die Ausgabe des einen dient als Eingabe für das nächste Werkzeug:



## Pipes and Filters

Anzahl Kommentarzeilen in PHP-Datei ausgeben

```
cat LocalSettings.php | grep "^ * [#|//]" | wc -l
```

## Pipes and Filters

### Vorteile

- Flexibilität durch Austausch und Hinzufügen von Filtern
- Flexibilität durch Neuordnung
- Wiederverwendung einzelner Filter
- Rapid Prototyping von Pipeline Prototypen
- Zwischendateien sind nicht notwendig aber so gewünscht möglich
- Parallel-Verarbeitung möglich

## Pipes and Filters

### Nachteile

- Die Kosten der Datenübertragung zwischen den Filtern können je nach Pipe sehr hoch sein
- Häufig überflüssige Datentransformationen zwischen den einzelnen Filterstufen
- Fehlerbehandlung über Filterstufen hinweg ist teilweise schwierig
- Gemeinsamer Zustand (z.B. Symboltabelle in Compilern) ist teuer und unflexibel
- Effizienzsteigerung durch Parallelisierung oft nicht möglich (z.B. da Filter aufeinander warten oder nur ein Prozessor arbeitet)

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

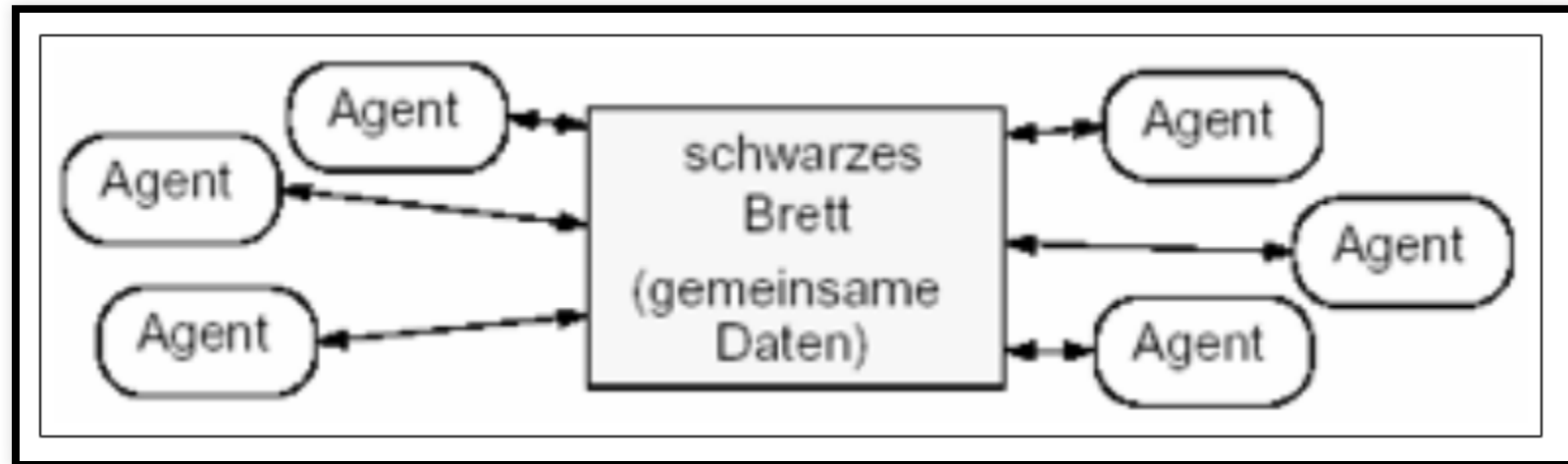
Blackboard

Domain-driven Design

## Blackboard

Das Blackboard Muster wird angewendet bei Problemen, die nicht auf eine eindeutige Lösungsstrategie hindeuten. Den Kontext für „Blackboard“ bilden somit Problembereiche, für die es noch keine festgelegten Lösungsstrategien gibt. Beispiele hierfür sind Spracherkennungs-, Bildverarbeitungs-, sowie Überwachungssysteme.

## Blackboard



## Blackboard

Das Schwarzes Brett dient als zentrale Datenstruktur. Agenten verarbeiten vorhandenes und bringen neues Wissen. Eine Steuerung entscheidet, welcher Agent die Bedingung zum Ermitteln von neuem Wissen erfüllt und somit das Programm der Lösung einen Schritt näher bringen könnte, dann aktiviert es den Agenten.

## Blackboard

Die Zugriffe von Agenten auf das schwarze Brett stellen die Konnektoren da. Die Agenten sind völlig entkoppelt und können auch zur Laufzeit hinzugefügt und ausgetauscht werden, ohne dass andere Agenten betroffen sind. Die parallele Ausführung von Agenten ist ebenfalls möglich.



## Blackboard

Das Programmverhalten von Systemen, für die solch eine Architektur eingesetzt wird, ist hochgradig nichtdeterministisch und daher schwer prüfbar. Im Bereich Robotersteuerung und Mustererkennung (Bild, Ton, Sprache, Schrift) wird aufgrund der nichtdeterministischen Problemlösung die Black Board Architektur häufig verwendet.

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

## Domain-driven Design

Domain-driven Design ist nicht nur eine Technik oder Methode. Es ist viel mehr eine Denkweise und Priorisierung zur Steigerung der Produktivität von Softwareprojekten im Umfeld komplexer fachlicher Zusammenhänge

## Domain-driven Design

Domain-driven Design basiert auf folgenden zwei Annahmen:

- Der Schwerpunkt des Softwaredesigns liegt auf der Fachlichkeit und der Fachlogik.
- Der Entwurf komplexer fachlicher Zusammenhänge sollte auf einem Modell der Anwendungsdomäne, dem Domänenmodell basieren.

Domain-driven Design

Entwicklungsprozess

Domain-driven Design ist an keinen bestimmten Softwareentwicklungsprozess gebunden, orientiert sich aber an agiler Softwareentwicklung.

Insbesondere setzt es iterative Softwareentwicklung und eine enge Zusammenarbeit zwischen Entwicklern und Fachexperten voraus.

# Domain-driven Design

## Konzepte

Domain-driven Design basiert auf einer Reihe von Konzepten, welche bei der Modellierung, aber auch anderen Tätigkeiten der Softwareentwicklung, berücksichtigt werden sollten.

Der Kern ist die Einführung einer allgemein verwendeten (ubiquitous) Sprache, welche in allen Bereichen der Softwareerstellung verwendet werden sollte.

DDD - Zusammenfassung

Domain-driven Design is object oriented programming  
done right. (*Eric Evans*)

Chaos zu Struktur / Mud-to-structure

# Verteilte Systeme

Interaktive Systeme

Adaptive Systeme

Domain-spezifische Architektur



Verteilte Systeme

# Serviceorientierte Architektur (SOA)

Peer-to-Peer

Client-Server

## Serviceorientierte Architektur (SOA)

”SOA ist ein Paradigma für die Strukturierung und Nutzung verteilter Funktionalität, die von unterschiedlichen Besitzern verantwortet wird.”

## Serviceorientierte Architektur (SOA)

- SOA soll Dienste von IT-Systemen strukturieren und zugänglich machen.
- SOA orientiert sich an Geschäftsprozessen
- Geschäftsprozesse sind die Grundlage für konkrete Serviceimplementierungen

## Serviceorientierte Architektur (SOA)

- Eine technische Form der Umsetzung von SOA ist das Anbieten dieser Dienste im Internet oder in der Cloud.
- Die Kommunikation zwischen solchen angebotenen Diensten kann über SOAP, REST, XML-RPC oder ähnliche Protokolle erfolgen.

## Serviceorientierte Architektur (SOA)

- Der Nutzer dieser Dienste weiß nur, dass der Dienst angeboten wird, welche Eingaben er erfordert und welcher Art das Ergebnis ist.
- Details über die Art und Weise der Ergebnisermittlung müssen nicht bekannt sein.

## Serviceorientierte Architektur (SOA)

- Ein Dienst ist eine IT-Repräsentation von fachlicher Funktionalität.
- Ein Dienst ist in sich abgeschlossen (autark) und kann eigenständig genutzt werden.
- Ein Dienst ist in einem Netzwerk verfügbar.
- Ein Dienst hat eine wohldefinierte veröffentlichte Schnittstelle (Vertrag). Für die Nutzung reicht es, die Schnittstelle zu kennen. Kenntnisse über die Details der Implementierung sind hingegen nicht erforderlich.

## Serviceorientierte Architektur (SOA)

- Ein Dienst ist plattformunabhängig, d. h. Anbieter und Nutzer eines Dienstes können in unterschiedlichen Programmiersprachen auf verschiedenen Plattformen realisiert sein.
- Ein Dienst ist in einem Verzeichnis registriert.
- Ein Dienst ist dynamisch gebunden, d. h. bei der Erstellung einer Anwendung, die einen Dienst nutzt, braucht der Dienst nicht vorhanden zu sein. Er wird erst bei der Ausführung lokalisiert und eingebunden.
- Ein Dienst sollte grobgranular sein, um die Abhängigkeit zwischen verteilten Systemen zu senken.

## Serviceorientierte Architektur (SOA)

### Vorteile

- Eine agile IT-Umgebung, die schnell auf geschäftliche Veränderungen reagieren kann
- Niedrigere Gesamtbetriebskosten durch die Wiederverwendung von Services
- Höhere Leistung, größere Skalierbarkeit und Transparenz
- Dienstleistungen und Produkte können schneller auf den Markt gebracht werden



## Serviceorientierte Architektur (SOA)

### Nachteile

- SOA wird von Marketingabteilungen gehyped: Einführung von SOA ist die Lösung aller bisherigen Probleme
- SOA generiert einen höheren Aufwand als bisherige monolithische Programmstrukturen.
- SOA erzeugt im Code wesentlich komplexere Abläufe
- SOA setzt für die beteiligten Entwickler ein erhebliches Know-how voraus.
- *Somit sind Entwickler auch nicht so einfach ersetzbar, und die Abhängigkeit der Unternehmen von einzelnen Entwicklern steigt deutlich.*

Verteilte Systeme

Serviceorientierte Architektur (SOA)

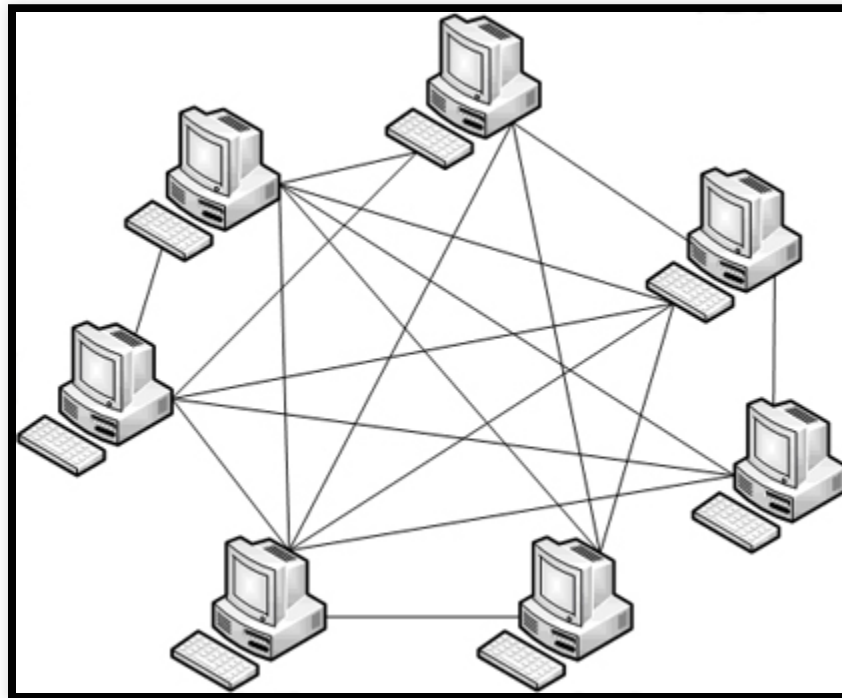
Peer-to-Peer

Client-Server

## Peer-to-Peer

In einem reinen Peer-to-Peer-Netz sind alle Computer gleichberechtigt und können sowohl Dienste in Anspruch nehmen, als auch zur Verfügung stellen.

# Peer-to-Peer



## Peer-to-Peer

### Typische Eigenschaften:

- Hohe Heterogenität bezüglich der Bandbreite, Rechenkraft, Online-Zeit
- Die Verfügbarkeit und Verbindungsqualität der Peers kann nicht vorausgesetzt werden
- Peers bieten Dienste und Ressourcen an und nehmen Dienste anderer Peers in Anspruch
- Dienste und Ressourcen können zwischen allen teilnehmenden Peers ausgetauscht werden.
- Peers haben eine signifikante Autonomie (über die Ressourcenbereitstellung).
- Das P2P-System ist selbstorganisierend.

Peer-to-Peer

## Vorteile

- alle Computer sind gleichberechtigt
- Kostengünstiger als Servernetzwerke
- Kein leistungsstarker zentraler Server erforderlich
- Keine spezielle Netzwerksoftware erforderlich
- Benutzer verwalten sich selbst
- Keine hierarchische Netzwerkstruktur

Peer-to-Peer

## Nachteile

- Zentrale Sicherheitsaspekte sind nicht von Bedeutung
- Sehr schwer zu administrieren
- kein einziges Glied im System ist verlässlich

Verteilte Systeme

Serviceorientierte Architektur (SOA)

Peer-to-Peer

Client-Server



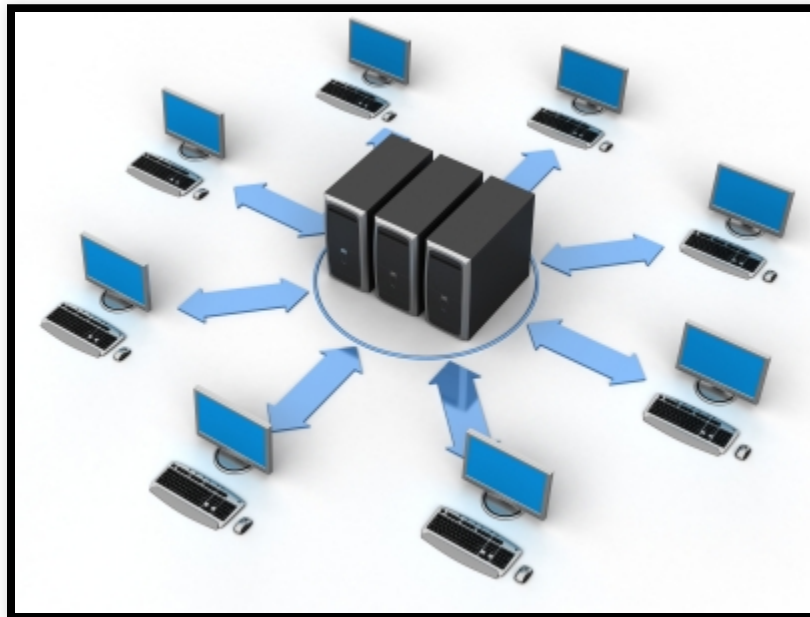
## Client-Server

Das Client-Server-Modell verteilt Aufgaben und Dienstleistungen innerhalb eines Netzwerkes.

## Client-Server

- Der Client kann auf Wunsch einen Dienst vom Server anfordern
- Der Server beantwortet die Anforderung.
- Üblicherweise kann ein Server gleichzeitig für mehrere Clients arbeiten.

# Client-Server



## Client-Server

- Ein Server ist ein Programm, das einen Dienst (Service) anbietet.
- Ein anderes Programm, der Client, kann diesen Dienst nutzen.
- Die Kommunikation zwischen Client und Server ist abhängig vom Dienst
- Der Dienst bestimmt, welche Daten zwischen beiden ausgetauscht werden.
- Der Server ist in Bereitschaft, um jederzeit auf die Kontaktaufnahme eines Clients reagieren zu können.
- Der Server ist passiv und wartet auf Anforderungen.
- Die Regeln der Kommunikation für einen Dienst werden durch ein für den jeweiligen Dienst spezifisches Protokoll festgelegt.

## Client-Server

- Clients und Server können als Programme auf verschiedenen Rechnern oder auf demselben Rechner ablaufen.
- Das Konzept kann zu einer Gruppe von Servern ausgebaut werden, die eine Gruppe von Diensten anbietet.
- In der Praxis laufen Server-Dienste meist gesammelt auf bestimmten Rechnern, die dann selber "Server" genannt werden

Client-Server

## Vorteile

- Gute Skalierbarkeit
- Einheitliches Auffinden von Objekten

Client-Server

## Nachteile

- Der Server muss immer in Betrieb sein
- Der Server muss gegen Ausfall und Datenverlust gesichert werden

Chaos zu Struktur / Mud-to-structure

Verteilte Systeme

Interaktive Systeme

Adaptive Systeme

Domain-spezifische Architektur



Interaktive Systeme

# Model View Controller (MVC)

Model View Presenter

Presentation-Abstraction-Control (PAC)

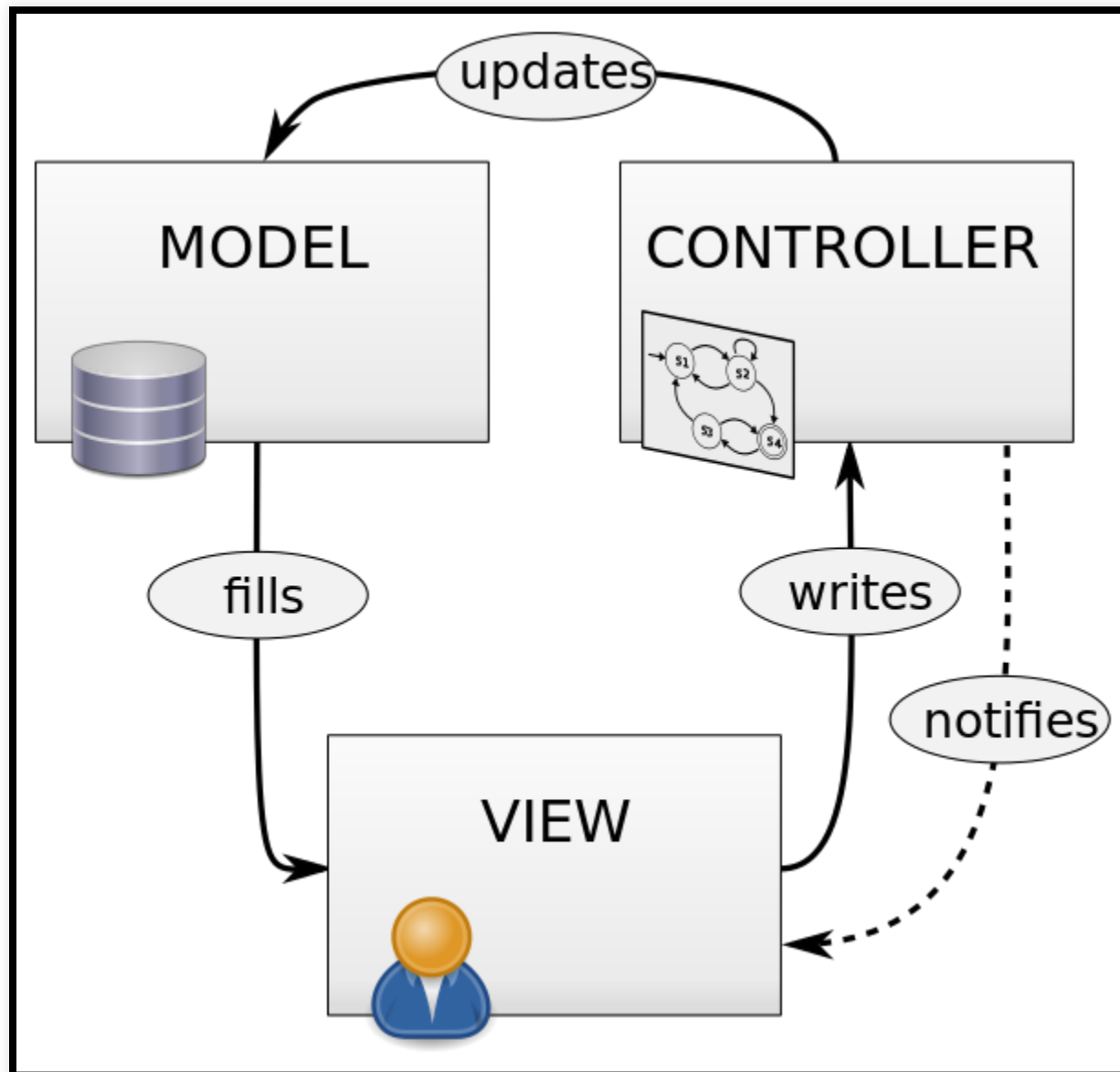
## Model View Controller (MVC)

Das MVC-Pattern ist eine spezielle Variante des Layers-Pattern, die sich aus den drei Schichten Datenhaltung (Model), Programmlogik (Controller) und Präsentation (View) zusammensetzt.

## Model View Controller (MVC)

- Model: Speicherung und Zugriffskontrolle von Daten
- View: Darstellung der Daten für die Anwender
- Controller: Vermittlung zwischen View und Model

# Model View Controller (MVC)



## Model View Controller (MVC)

### Einsatzgebiete / Gründe

- Benutzerschnittstellen sind besonders häufig von Änderungen betroffen
- Information auf verschiedene Weise darstellen
- Änderungen an den Daten sofort in unterschiedlichen Darstellungen sichtbar machen
- Verschiedene Benutzerschnittstellen unterstützen, ohne den Kern der Anwendung zu verändern?

## Model View Controller (MVC)

### Vorteile

- Mehrere Sichten desselben Modells
- Automatische Synchronisation aller Views
- Austauschbarkeit von Views und Controllern
- Gute Trennung von Modell und View
- Potential für vorgefertigte Frameworks

## Model View Controller (MVC)

### Nachteile

- Erhöhte Komplexität
- Starke Kopplung zwischen Modell und View
- Starke Kopplung zwischen Modell und Controller
- Potential für unnötig häufige Aktualisierungen
- Häufig ineffizienter Datenzugriff auf das Modell.
- View und Controller sind schwer zu portieren.

Interaktive Systeme

Model View Controller (MVC)

**Model View Presenter**

Presentation-Abstraction-Control (PAC)



Model View Presenter

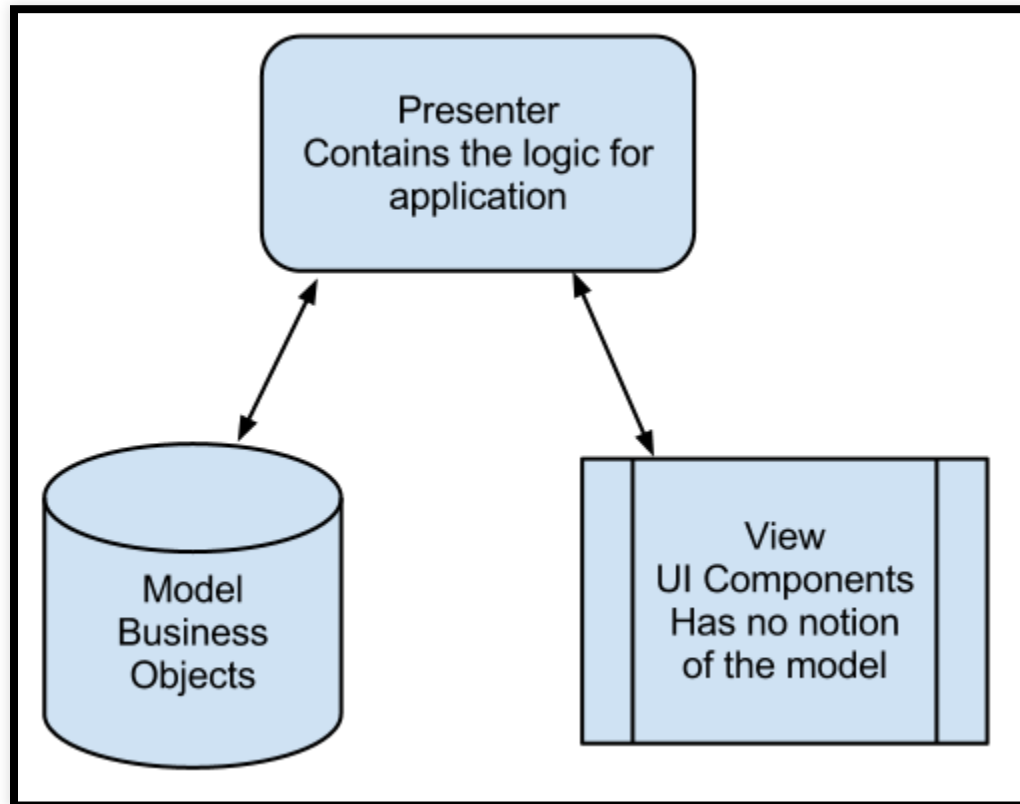
Hervorgegangen aus dem Model-View-Controller  
(MVC) Architekturmuster.

Vollständige Trennung von Model und View, Verbindung über einen  
Presenter.

## Model View Presenter

- Vollständige Trennung von Model und View
- Deutlich verbesserte Testbarkeit
- Strenge Trennung der einzelnen Komponenten

## Model View Presenter (MVP)



## Model View Presenter (MVP)

MVP basiert wie MVC auch auf drei Komponenten:

- Model
- View
- Presenter

## Model View Presenter (MVP)

### Bedingungen

- Model und View verwenden jeweils eigene Schnittstellen
- Die Schnittstellen definieren den genauen Aufbau beider Schichten
- Der Präsentator verknüpft lediglich die Schnittstellen miteinander

## Model View Controller (MVP)

### Vorteile

- Mehrere Sichten desselben Modells
- Automatische Synchronisation aller Views
- Vollständige Austausch- und Wiederverwertbarkeit des Modells und der Ansicht
- Gute Testbarkeit der View durch 'doubles'

## Model View Controller (MVP)

### Nachteile

- Erhöhte Komplexität
- Potential für unnötig häufige Aktualisierungen
- Häufig ineffizienter Datenzugriff auf das Modell.
- Kompliziert umsetzbar

Interaktive Systeme

Model View Controller (MVC)

Model View Presenter

Presentation-Abstraction-Control (PAC)



## Presentation-Abstraction-Control (PAC)

Hohe Flexibilität für ein System, das aus vielen autarken Einzelsystemen zusammengesetzt ist.

## Presentation-Abstraction-Control (PAC)

- Große Systeme, für die das Model-View-Controller-Muster nicht ausreicht
- Aufteilung des Systems in zwei Richtungen
- --> In die drei Einheiten Presentation, Control und Abstraction (ähnlich dem MVC)
- --> Hierarchisch in verschiedene Teile („Agenten“), die jeweils einen Teil der Aufgaben des Systems anbieten

## Presentation-Abstraction-Control (PAC)

### Agenten

- Stellen die erste Stufe der Strukturierung während des Architekturentwurfes dar
- Aufteilung der gesamten Anforderungen auf einzelne Agenten
- Aufbau der hierarchischen Struktur
- Für jeden Agenten erfolgt dann eine Aufteilung in Presentation, Abstraction und Control

Presentation-Abstraction-Control (PAC)

## Hierarchie: Drei Schichten

- Top-Level-Agent: Globale Aufgaben
- Intermediate-Level-Agenten: Strukturierung der Bottom-Level-Agenten
- Bottom-Level-Agenten: Konkrete, abgeschlossene Aufgaben

## Presentation-Abstraction-Control (PAC)

### Vorteile

- Zerlegung der Funktionen des Gesamtsystems in einzelne semantisch getrennte Teile
- Gute Erweiterbarkeit durch neue Agenten
- Gute Wartbarkeit ist durch die interne Struktur der Agenten

## Presentation-Abstraction-Control (PAC)

### Nachteile

- Erhöhte Systemkomplexität
- Erhöhter Koordinations- und Kommunikationsaufwand zwischen den Agenten
- Die Steuerungskomponenten können eine hohe Komplexität erreichen

Fragen?

Chaos zu Struktur / Mud-to-structure

Verteilte Systeme

Interaktive Systeme

Adaptive Systeme



Adaptive Systeme

**Mikrokernel**

Reflexion

Dependency Injection

Mikrokernel

Ziel: Änderung von Systemanforderungen zur Laufzeit  
dynamisch begegnen.

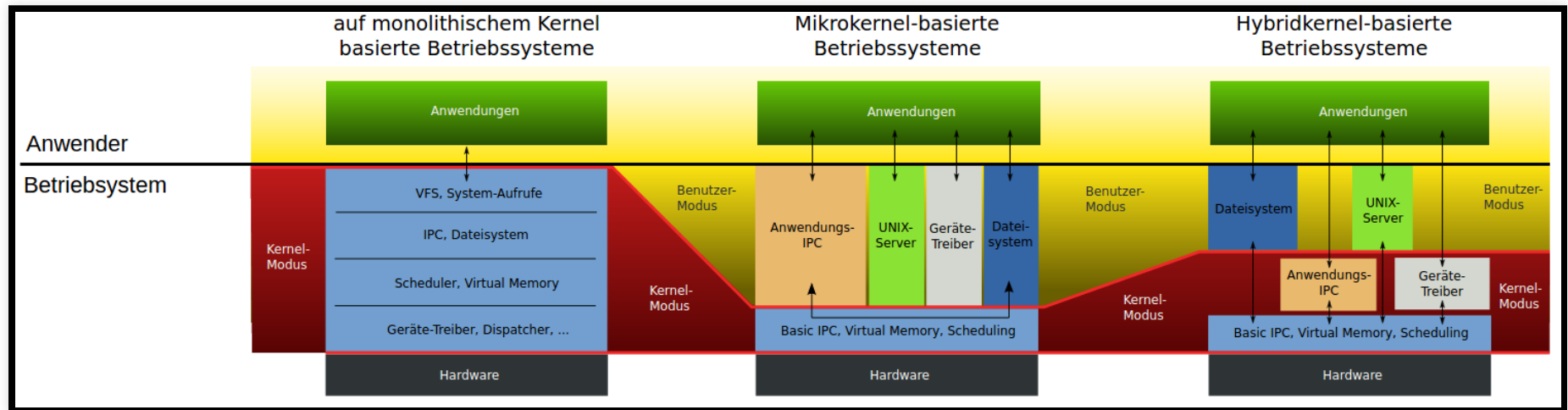
Mikrokernel

## Aufgaben

- Der Mikrokernel bietet die Basis für mögliche Erweiterungen
- Der Mikrokernel koordiniert die Zusammenarbeit.

# Microkernel

## Herausforderung



Mikrokernel

## Beispiele: Mikrokernel

- Minix Kernel
- GNU Mach
- AmigaOS
- SymbianOS

Mikrokernel

## Beispiele: Monolithische Kernels

- Linux
- Android
- Windows bis Win98 (DOS Kernel)

Mikrokernel

## Beispiele: Hybrid-Kernels

- MacOS X (Darwin)
- Windows NT (oft als Mikrokernel bezeichnet)

## Mikrokernel

### Vorteile

- Separierte Komponenten: Austauschbarkeit
- Treiber im Benutzer-Modus: Sicherheit
- kleine Trusted Computing Base
- Skalierbarkeit
- Zuverlässigkeit
- Transparenz



Mikrokernel

## Nachteile

- Leistung
- Komplexität

Fragen?

Rückblick auf Architekturmuster

Chaos zu Struktur / Mud-to-structure

Verteilte Systeme

Interaktive Systeme

Adaptive Systeme

Domain-spezifische Architektur

## Chaos zu Struktur / Mud-to-structure

- Organisation der Komponenten und Objekte eines Softwaresystems
- Die Funktionalität des Gesamtsystems wird in kooperierende Subsysteme aufgeteilt
- Zu Beginn des Softwareentwurfs werden Anforderungen analysiert und spezifiziert
- Integrierbarkeit, Wartbarkeit, Änderbarkeit, Portierbarkeit und Skalierbarkeit sollen berücksichtigt werden

# Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

# Verteilte Systeme

- Verteilung von Ressourcen und Dienste in Netzwerken
- Kein "zentrales System" mehr
- Basiert auf guter Infrastruktur lokaler Datennetze

# Verteilte Systeme

Serviceorientierte Architektur (SOA)

Peer-to-Peer

Client-Server



# Interaktive Systeme

- Strukturierung von Mensch-Computer-Interaktionen
- Möglichst gute Schnittstellen für die Benutzer schaffen
- Der eigentliche Systemkern bleibt von der Benutzerschnittstelle unangetastet.

# Interaktive Systeme

Model View Controller (MVC)

Model View Presenter

Presentation-Abstraction-Control (PAC)

# Adaptive Systeme

- Unterstützung der Erweiterungs- und Anpassungsfähigkeit von Softwaresystemen.
- Das System sollte von vornherein mögliche Erweiterungen unterstützen
- Die Kernfunktionalität sollte davon unberührt bleiben kann.

# Adaptive Systeme

Mikrokern

Reflexion

Dependency Injection

Wie wähle ich denn nun das passende Muster aus?

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Dokumentation von Architekturen

## Nutzen von Templates

Beispiele:

- arc42
- Normen
- Software Guidebook

# ARC42

(Dr. Gernot Starke / Dr. Peter Hruschka)

<http://www.arc42.de/>

*arc42 unterstützt Software- und Systemarchitekten. Es kommt aus der Praxis und basiert auf Erfahrungen internationaler Architekturprojekte und Rückmeldungen vieler Anwender.*

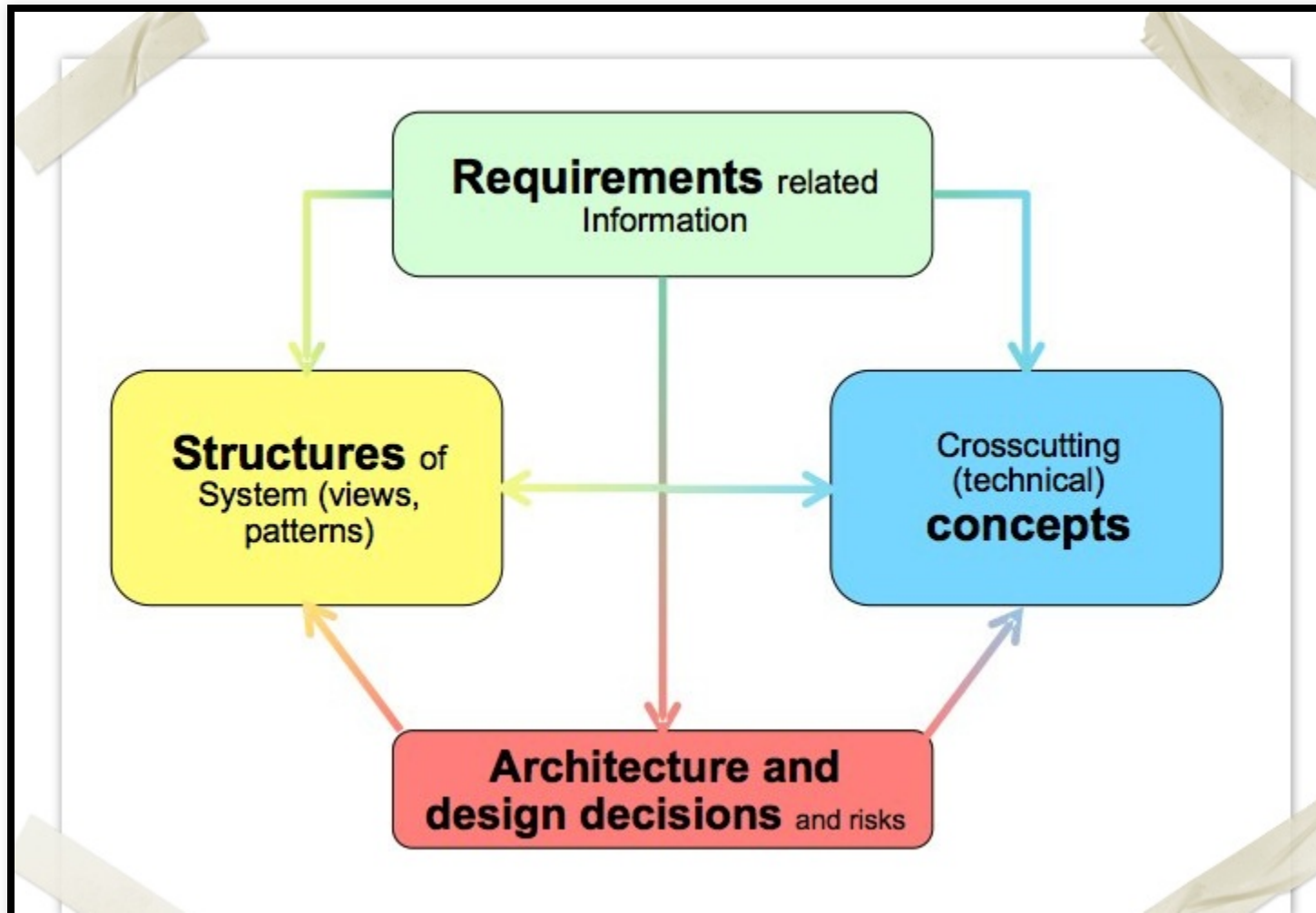


# Dokumentation von Architekturen

## ARC42

1. Einführung und Ziele
2. Randbedingungen
3. Kontextabgrenzung
4. Lösungsstrategie
5. Bausteinsicht
6. Laufzeitsicht
7. Verteilungssicht
8. Querschnittliche Konzepte/Muster
9. Entwurfsentscheidungen
10. Qualitätsszenarien
11. Risiken
12. Glossar

# ARC42



# ARC42

## 1. Einführung und Ziele

- Aufgabenstellung
- Qualitätsziele
- eine Kurzfassung der architekturelevanten Anforderungen (insb. die nichtfunktionalen)
- Stakeholder

# ARC42

## 2. Randbedingungen

Welche Leitplanken schränken die Entwurfsentscheidungen ein?

- Technische Randbedingungen
- Organisatorische Randbedingungen
- Konventionen

ARC42

### 3. Kontextabgrenzung

- Fachlicher Kontext
- Technischer- oder Verteilungskontext

ARC42

#### 4. Lösungsstrategie

Wie funktioniert die Lösung? Was sind die fundamentalen Lösungsansätze?

ARC42

## 5. Bausteinsicht

Die statische Struktur des Systems, der Aufbau aus Implementierungsteilen.

ARC42

## 6. Laufzeitsicht

Zusammenwirken der Bausteine zur Laufzeit, gezeigt an exemplarischen Abläufen ("Szenarien")



ARC42

7. Verteilungssicht

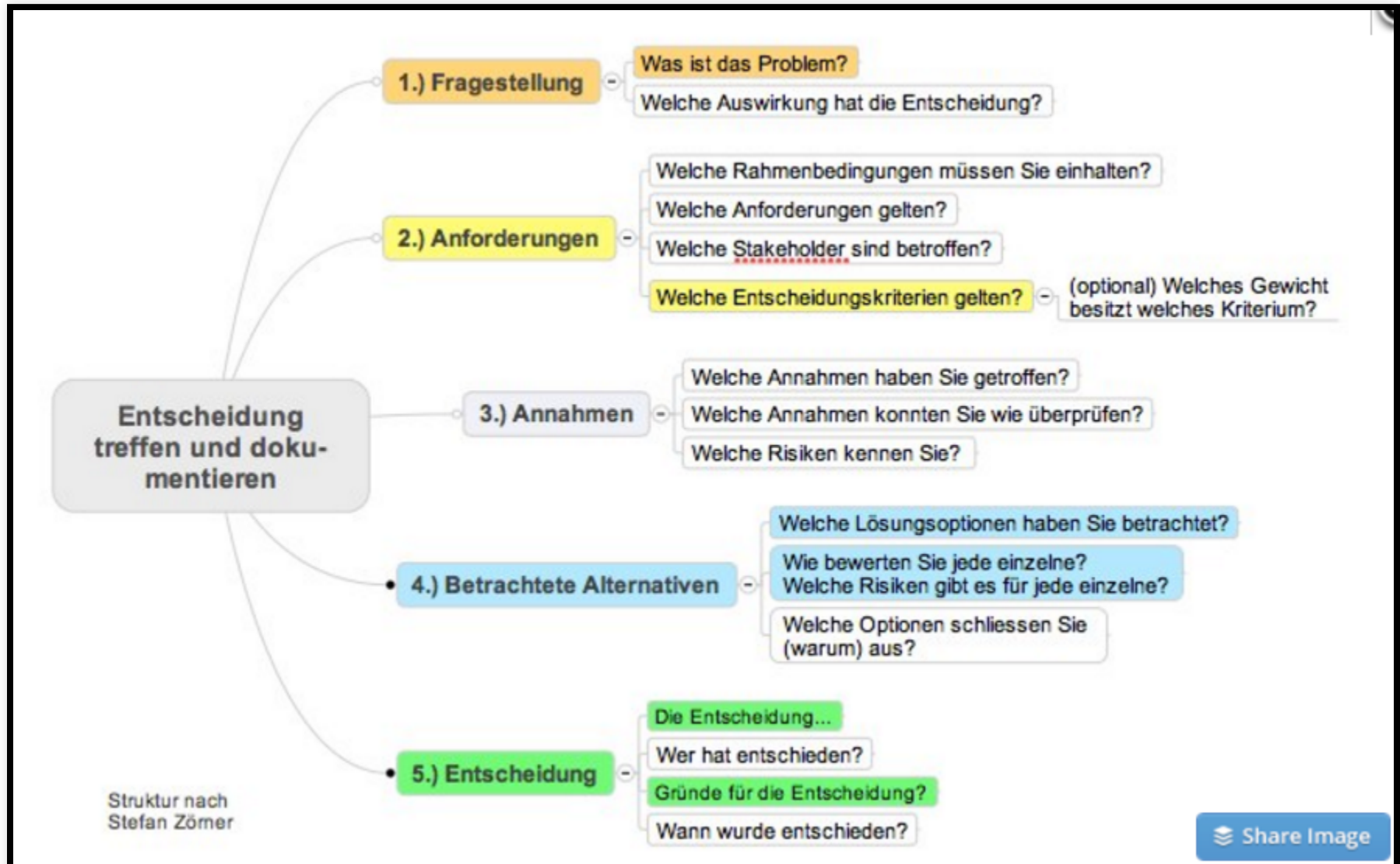
Deployment: Auf welcher Hardware werden die Bausteine betrieben?

## ARC42

### 8. Querschnittliche Konzepte und Muster

- Wiederkehrende Muster und Strukturen
- Fachliche Strukturen
- Querschnittliche, übergreifende Konzepte
- Nutzungs- oder Einsatzanleitungen für Technologien
- Oftmals projekt-/systemübergreifend verwendbar!

## 9. Entwurfsentscheidungen



ARC42

## 10. Qualitätsszenarien

Qualitätsbaum sowie dessen Konkretisierung durch Szenarien

ARC42

## 11. Risiken

Eine nach Prioritäten geordnete Liste der erkannten Risiken

”Risikomanagement ist Projektmanagement für Erwachsene”

ARC42

12. Glossar

Die wichtigsten Begriffe der Software-Architektur in alphabetischer  
Reihenfolge

ARC42

<http://confluence.arc42.org/>

Fragen?

# IEEE Standards

**IEEE Standards** documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards or implementations thereof.



# IEEE Standards

- IEEE 802: LAN
- IEEE 802.3: Carrier sense multiple access with collision detection (CSMA/CD)
- IEEE 802.11: Wireless LAN
- IEEE 830: Recommended Practice for Software Requirements Specifications
- IEEE 1394: FireWire/i.Link Bussysteme
- IEEE 1471: Recommended Practice for Architectural Description of Software-Intensive Systems
- IEEE 9945: Portable Operating System Interface (POSIX®)

IEEE 830-1998

IEEE

Fragen?

# Software Guidebook

- Template von Simon Brown aus dem Buch "*Software Architecture for Developers*"
- Buch: <https://leanpub.com/software-architecture-for-developers>
- Beispiel: <https://leanpub.com/techtribesje> (kostenlos)

# Software Guidebook

Welche Informationen wünsche ich mir, wenn ich in ein neues Projekt komme?

- Karten
- Sichten
- Geschichte
- Praktische Informationen!

Software Guidebook

# Product vs project documentation

# Software Guidebook

1. Context
2. Functional Overview
3. Quality Attributes
4. Constraints
5. Principles
6. Software Architecture
7. External Interfaces
8. Code
9. Data
10. Infrastructure Architecture
11. Deployment
12. Operation and Support
13. Development Environment

## Software Guidebook

# Context

A context section should be one of the first sections of the software guidebook and simply used to set the scene for the remainder of the document.



## Software Guidebook

# Context

A context section should answer the following types of questions:

- What is this software project/product/system all about?
- What is it that's being built?
- How does it fit into the existing environment? (e.g. systems, business processes, etc)
- Who is using it? (users, roles, actors, personas, etc)

## Software Guidebook

# Functional Overview

Even though the purpose of a software guidebook isn't to explain what the software does in detail, it can be useful to expand on the context and summarise what the major functions of the software are.

## Software Guidebook

# Functional Overview

- Is it clear what the system actually does?
- Is it clear which features, functions, use cases, user stories, etc are significant to the architecture and why?
- Is it clear who the important users are (roles, actors, personas, etc) and how the system caters for their needs?
- It is clear that the above has been used to shape and define the architecture? Alternatively, if your software automates a business process or workflow, a functional view should answer questions like the following:
  - Is it clear what the system does from a process perspective?
  - What are the major processes and flows of information through the system?

Software Guidebook

## Quality Attributes

With the functional overview section summarising the functionality, it's also worth including a separate section to summarise the quality attributes/non-functional requirements.

## Software Guidebook

# Quality Attributes

- Is there a clear understanding of the quality attributes that the architecture must satisfy?
- Are the quality attributes SMART (specific, measurable, achievable, relevant and timely)?
- Have quality attributes that are usually taken for granted been explicitly marked as out of scope if they are not needed? (e.g. “user interface elements will only be presented in English” to indicate that multi-language support is not explicitly catered for)
- Are any of the quality attributes unrealistic? (e.g. true 24x7 availability is typically very costly to implement inside many organisations)

## Software Guidebook

# Quality Attributes

Simply listing out each of the quality attributes is a good starting point. Examples include:

- Performance (e.g. latency and throughput)
- Scalability (e.g. data and traffic volumes)
- Availability (e.g. uptime, downtime, scheduled maintenance, 24x7, 99.9%, etc)
- Security (e.g. authentication, authorisation, data confidentiality, etc)
- Extensibility
- Flexibility
- Auditing

## Software Guidebook

# Quality Attributes

- Monitoring and management
- Reliability
- Failover/disaster recovery targets (e.g. manual vs automatic, how long will this take?)
- Business continuity
- Interoperability
- Legal, compliance and regulatory requirements (e.g. data protection act)
- Internationalisation (i18n) and localisation (L10n)
- Accessibility
- Usability

## Software Guidebook

# Constraints

Software lives within the context of the real-world, and the real-world has constraints. This section allows you to state these constraints so it's clear that you are working within them and obvious how they affect your architecture decisions.



## Software Guidebook

# Constraints

Constraints are typically imposed upon you but they aren't necessarily "bad", as reducing the number of available options often makes your job designing software easier. This section allows you to explicitly summarise the constraints that you're working within and the decisions that have already been made for you.

## Software Guidebook

# Constraints

- Time, budget and resources.
- Approved technology lists and technology constraints.
- Target deployment platform.
- Existing systems and integration standards.
- Local standards (e.g. development, coding, etc).
- Public standards (e.g. HTTP, SOAP, XML, XML Schema, WSDL, etc).
- Standard protocols.
- Standard message formats.
- Size of the software development team.
- Skill profile of the software development team.
- Nature of the software being built (e.g. tactical or strategic).
- Political constraints.
- Use of internal intellectual property.
- etc

## Software Guidebook

# Principles

The principles section allows you to summarise those principles that have been used (or you are using) to design and build the software.

## Software Guidebook

# Principles

The purpose of this section is to simply make it explicit which principles you are following. These could have been explicitly asked for by a stakeholder or they could be principles that you (i.e. the software development team) want to adopt and follow.

## Software Guidebook

# Principles

If you have an existing set of software development principles (e.g. on a development wiki), by all means simply reference it. Otherwise, list out the principles that you are following and accompany each with a short explanation or link to further information.

## Software Guidebook

# Principles

Example principles include:

- Architectural layering strategy.
- No business logic in views.
- No database access in views.
- Use of interfaces.
- Always use an ORM.
- Dependency injection.
- The Hollywood principle (don't call us, we'll call you).
- High cohesion, low coupling.

## Software Guidebook

# Principles

- Follow SOLID (Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle).
- DRY (don't repeat yourself).
- Ensure all components are stateless (e.g. to ease scaling).
- Prefer a rich domain model.
- Prefer an anaemic domain model.
- Always prefer stored procedures.
- Never use stored procedures.
- Don't reinvent the wheel.
- Common approaches for error handling, logging, etc.
- Buy rather than build.

## Software Guidebook

# Software Architecture

The software architecture section is your “big picture” view and allows you to present the structure of the software. Traditional software architecture documents typically refer to this as a “conceptual view” or “logical view”, and there is often confusion about whether such views should refer to implementation details such as technology choices.



## Software Guidebook

# Software Architecture

- What does the “big picture” look like?
- Is there are clear structure?
- Is it clear how the system works from the “30,000 foot view”?
- Does it show the major containers and technology choices?
- Does it show the major components and their interactions?
- What are the key internal interfaces? (e.g. a web service between your web and business tiers)

Software Guidebook

## External Interfaces

Interfaces, particularly those that are external to your software system, are one of the riskiest parts of any software system so it's very useful to summarise what the interfaces are and how they work.

## Software Guidebook

# External Interfaces

- What are the key external interfaces?
- e.g. between your system and other systems (whether they are internal or external to your environment)
- e.g. any APIs that you are exposing for consumption
- e.g. any files that your are exporting from your system
- Has each interface been thought about from a technical perspective?
- What is the technical definition of the interface?

## Software Guidebook

# External Interfaces

- If messaging is being used, which queues (point-to-point) and topics (pub-sub) are components using to communicate?
- What format are the messages (e.g. plain text or XML defined by a DTD/ Schema)?
- Are they synchronous or asynchronous?
- Are asynchronous messaging links guaranteed?
- Are subscribers durable where necessary?
- Can messages be received out of order and is this a problem?

## Software Guidebook

# External Interfaces

- Are interfaces idempotent?
- Is the interface always available or do you, for example, need to cache data locally?
- How is performance/scalability/security/etc catered for?
- Has each interface been thought about from a non-technical perspective?
- Who has ownership of the interface?
- How often does the interface change and how is versioning handled?
- Are there any service-level agreements in place?

## Software Guidebook

### Code

Although other sections of the software guidebook describe the overall architecture of the software, often you'll want to present lower level details to explain how things work. This is what the code section is for. Some software architecture documentation templates call this the “implementation view” or the “development view”.

## Software Guidebook

### Code

- Generating/rendering HTML: a short description of an in-house framework that was created for generating HTML, including the major classes and concepts.
- Data binding: our approach to updating business objects as the result of HTTP POST requests.
- Multi-page data collection: a short description of an in-house framework we used for building forms that spanned multiple web pages.
- Web MVC: an example usage of the web MVC framework that was being used.
- Security: our approach to using Windows Identity Foundation (WIF) for authentication and authorisation.

## Software Guidebook

# Code

- Domain model: an overview of the important parts of the domain model.
- Component framework: a short description of the framework that we built to allow components to be reconfigured at runtime.
- Configuration: a short description of the standard component configuration mechanism in use across the codebase.
- Architectural layering: an overview of the layering strategy and the patterns in use to implement it.
- Exceptions and logging: a summary of our approach to exception handling and logging across the various architectural layers.
- Patterns and principles: an explanation of how patterns and principles are implemented.



## Software Guidebook

# Data

The data associated with a software system is usually not the primary point of focus yet it's arguably more important than the software itself, so often it's useful to document something about it.

## Software Guidebook

# Data

- What does the data model look like?
- Where is data stored?
- Who owns the data?
- How much storage space is needed for the data? (e.g. especially if you're dealing with "big data")
- What are the archiving and back-up strategies?
- Are there any regulatory requirements for the long term archival of business data?
- Likewise for log files and audit trails?
- Are flat files being used for storage? If so, what format is being used?

Software Guidebook

## Infrastructure Architecture

While most of the software guidebook is focussed on the software itself, we do also need to consider the infrastructure because software architecture is about software and infrastructure.

## Software Guidebook

# Infrastructure Architecture

This section is used to describe the physical/virtual hardware and networks on which the software will be deployed. Although, as a software architect, you may not be involved in designing the infrastructure, you do need to understand that it's sufficient to enable you to satisfy your goals. The purpose of this section is to answer the following types of questions:

## Software Guidebook

# Infrastructure Architecture

- Is there a clear physical architecture?
- What hardware (virtual or physical) does this include across all tiers?
- Does it cater for redundancy, failover and disaster recovery if applicable?
- Is it clear how the chosen hardware components have been sized and selected?

## Software Guidebook

# Infrastructure Architecture

- If multiple servers and sites are used, what are the network links between them?
- Who is responsible for support and maintenance of the infrastructure?
- Are there central teams to look after common infrastructure (e.g. databases, message buses, application servers, networks, routers, switches, load balancers, reverse proxies, internet connections, etc)?
- Who owns the resources?
- Are there sufficient environments for development, testing, acceptance, pre-production, production, etc?

Software Guidebook

## Deployment

The deployment section is simply the mapping between the software and the infrastructure.

## Software Guidebook

# Deployment

This section is used to describe the mapping between the software (e.g. containers) and the infrastructure. Sometimes this will be a simple one-to-one mapping (e.g. deploy a web application to a single web server) and at other times it will be more complex (e.g. deploy a web application across a number of servers in a server farm). This section answers the following types of questions:



## Software Guidebook

# Deployment

- How and where is the software installed and configured?
- Is it clear how the software will be deployed across the infrastructure elements described in the infrastructure architecture section? (e.g. one-to-one mapping, multiple containers per server, etc)
- If this is still to be decided, what are the options and have they been documented?
- Is it understood how memory and CPU will be partitioned between the processes running on a single piece of infrastructure?
- Are any containers and/or components running in an active-active, active-passive, hot-standby, cold-standby, etc formation?
- Has the deployment and rollback strategy been defined?
- What happens in the event of a software or infrastructure failure?
- Is it clear how data is replicated across sites?

Software Guidebook

## Operation and Support

The operations and support section allows you to describe how people will run, monitor and manage your software.

## Software Guidebook

# Operation and Support

Most systems will be subject to support and operational requirements, particularly around how they are monitored, managed and administered. Including a dedicated section in the software guidebook lets you be explicit about how your software will or does support those requirements. This section should address the following types of questions:

## Software Guidebook

# Operation and Support

- Is it clear how the software provides the ability for operation/support teams to monitor and manage the system?
- How is this achieved across all tiers of the architecture?
- How can operational staff diagnose problems?
- Where are errors and information logged? (e.g. log files, Windows Event Log, SMNP, JMX, WMI, custom diagnostics, etc)
- Do configuration changes require a restart?
- Are there any manual housekeeping tasks that need to be performed on a regular basis?
- Does old data need to be periodically archived?

## Software Guidebook

# Development Environment

The development environment section allows you to summarise how people new to your team install tools and setup a development environment in order to work on the software.

## Software Guidebook

# Development Environment

The purpose of this section is to provide instructions that take somebody from a blank operating system installation to a fully-fledged development environment.

## Software Guidebook

# Development Environment

- Pre-requisite versions of software needed.
- Links to software downloads (either on the Internet or locally stored).
- Links to virtual machine images.
- Environment variables, Windows registry settings, etc.
- Host name entries.
- IDE configuration.
- Build and test instructions.
- Database population scripts.
- Usernames, passwords and certificates for connecting to development and test services.
- Links to build servers.

## Software Guidebook

# Development Environment

If you're using automated solutions (such as Vagrant, Docker, Puppet, Chef, Rundeck, etc), it's still worth including some brief information about how these solutions work, where to find the scripts and how to run them.



# Software Guidebook: Ein Beispiel

- Beispiel: <https://leanpub.com/techtribesje>
- Website: <https://techtribes.je>

Fragen?

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Frameworks

Was ist ein Framework?

# Frameworks

Ein Framework ist kein fertiges Programm, es stellt einen Rahmen zur Verfügung.

# Frameworks

- Ein Framework ist eine semi-vollständige Applikation.
- Es stellt für Applikationen eine wiederverwendbare, gemeinsame Struktur zur Verfügung.
- Entwickler integrieren das Framework in ihre eigene Applikation ein, und erweitern es um die Applikationslogik.
- Frameworks stellen eine kohärente Struktur zur Verfügung, anstatt eine einfache Menge von Hilfsklassen anzubieten.

# Frameworks

- Ein Framework gibt in der Regel die Anwendungsarchitektur vor.
- Ein Framework definiert den Kontrollfluss der Anwendung
- Ein Framework definierte die Schnittstellen für die Applikation.

# Frameworks

Eine allgemeingültige Definition von Frameworks gibt es aufgrund der hohen Anzahl von Diversitäten nicht.



# Frameworks

## Vorteile

- Wiederverwendung von Code
- Grundfunktionalität muss nicht immer wieder implementiert werden
- Es existieren genormte Schnittstellen z.B. zu Datenbanken
- Frameworks erleichtern die Programmierarbeit und sparen Entwicklungszeit
- Frameworks können den Stil entscheidend verbessern

# Frameworks

## Nachteile

- Frameworks erhöhen die Komplexität der Anwendung
- Frameworks stecken voller Know-How und eine effiziente Anwendung erfordert Profiwissen
- Frameworks nehmen nicht das Verständnis der Grundlagen ab, auch wenn oft so gearbeitet wird
- Dokumentationen sind größtenteils unzureichend

# Frameworks

Wie wähle ich ein Framework aus?

# Popularität und Community

*Wie wahrscheinlich finde ich Hilfe und Entwickler?*

# Philosophie

*A tool developed by professionals for their own needs will obviously meet the demands of other professionals.*

# Sustainability / Nachhaltigkeit

*Kann das Framework "mitwachsen"?*

# Support

*Gibt es professionelle Hilfe neben der Community?*

# Technik

*Wie gut ist das Framework implementiert?*



# Security

*Wie schnell werden Sicherheitslücken reportet und geschlossen?*

# Dokumentation

*Wie gut, ausführlich und verständlich ist das Framework dokumentiert?*

*Wie aktuell ist die Doku?*

# Lizenz

*Ein Framework unter GPL Lizenz verlangt z.B., dass die Anwendung auch unter der GPL steht. MIT dagegen nicht.*

# Entwickler-Kapazität

*Wie wahrscheinlich werde ich Entwickler finden?*

# Hosting Requirements

*Wie einfach kann ich die Anwendung deployen?*

# Einfache Installation?

*Wie schnell ist ein neues Projekt eingerichtet?*

# Lernkurve

*Wie komplex ist das Framework?*

# Inhalte / Funktionen?

- AJAX
- Authentication
- Authorization
- Caching
- Data Validation
- Templating engine
- URL mapping / rewriting
- ...?



# DB Abstraktion / ORM

*Wie einfach/mächtig ist das Object Relational Mapping?*

# JS Library

*Welche JS Bibliothek ist per default dabei?*

# Unit Testing

*Wie sehr ist TDD Teil der Philosophie, wie ist der Tool-Support?*

# Skalierbarkeit?

*Wie einfach lässt sich die Anwendung bei Bedarf skalieren?*

Ausprobieren!

*Reviews lesen reicht nicht, Erfahrungen und das look&feel zählen!*

## Wann brauche ich ein Framework?

- Die Anwendung basiert im Wesentlichen auf CRUD Operationen
- Die Anwendung wird relativ groß
- UI und Anwendungslogik sollen getrennt werden
- Authentication und andere Grundfunktionen werden intensiv genutzt
- Zeitdruck + Das Framework ist bereits bekannt

## Wann brauche ich KEIN Framework?

- Ich brauche nur einen kleinen Teil des Frameworks (z.B. ORM)
- Zeitdruck + Das Framework ist nicht bekannt
- "Frameworks lösen jedes Problem"

# Überblick über aktuelle Frameworks

[https://en.wikipedia.org/wiki/Comparison\\_of\\_web\\_frameworks](https://en.wikipedia.org/wiki/Comparison_of_web_frameworks)



# Vorstellung konkreter Technologien & Frameworks

- Ruby on Rails
- NodeJS
- AngularJS
- Twitter Bootstrap

Fragen?

# Ruby on Rails

Ruby on Rails

## Ruby: Überblick

- Erste Version 1993
- Verbreitet seit 2006 (durch Rails)
- Objektorientiert
- Interpretiert
- Dynamisch getypt
- "Script Sprache"

Ruby on Rails

## Ruby: Tradeoffs

- Flexibilität vs. Sicherheit
- Laufzeit-Effizienz vs. Produktivität

## Ruby on Rails

# Ruby: Beispielcode

```
>> properties = ['object oriented', 'duck typed', 'productive', 'fun']  
=> ["object oriented", "duck typed", "productive", "fun"]  
>> properties.each {|property| puts "Ruby is #{property}."}  
Ruby is object oriented.  
Ruby is duck typed.  
Ruby is productive.  
Ruby is fun.
```

Ruby on Rails

# Ruby: Installation

Installation via

- OS Paketmanager
- rvm
- rbenv

Ruby on Rails

# Ruby: Programming Model

```
>> 4
=> 4
>> 4.class
=> Fixnum
>> 4 + 4
=> 8
>> 4.methods
=> ["inspect", "%", "<<", "singleton_method_added", "numerator", ...
    "!", "+", "to_i", "methods", ...
    ]
```

Alles ist ein Objekt



Ruby on Rails

# Ruby: Programming Model

```
>> x = 4
=> 4
>> x < 5
=> true
>> x <= 4
=> true
>> x > 4
=> false
>> false.class
=> FalseClass
>> true.class
=> TrueClass
```

Alles ist ein Objekt

## Ruby on Rails

# Ruby: Programming Model

```
>> 4 + 'four'
TypeError: String can't be coerced into Fixnum
from (irb):51:in `+'
from (irb):51
>>
=>
>>
=>
4.class
Fixnum
(4.0).class
Float
>> 4 + 4.0
=> 8.0
```

## Duck Typing

# Ruby on Rails

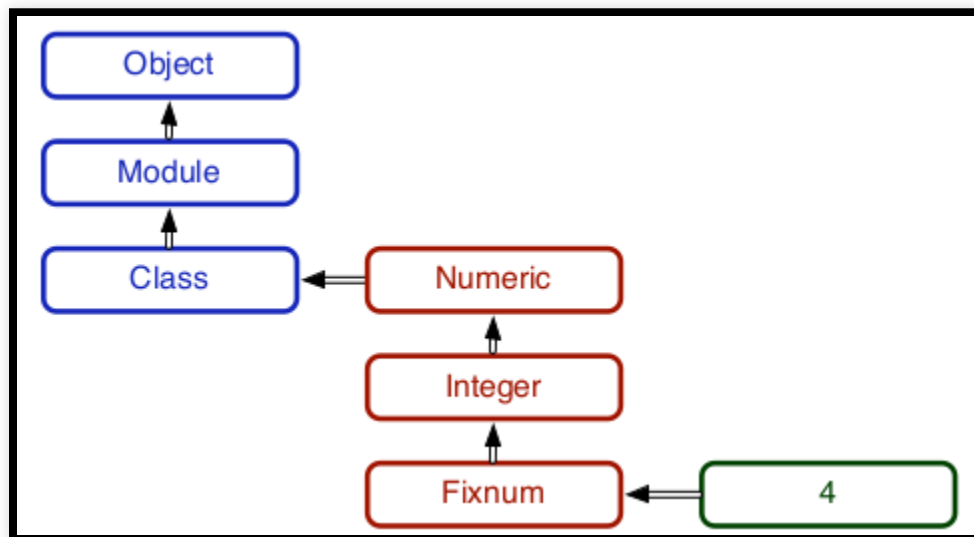
## Ruby: Programming Model

```
If it walks like a duck and quacks like a duck, it's a duck.
```

- Dynamische Typisierung: Zur Ausführungszeit interpretiert
- Starke Typisierung: Typsicherheit

Ruby on Rails

# Ruby: Metamodel



## Ruby: Literaturempfehlungen

- Programming Ruby (Dave Thomas / Andy Hunt)
- Practical Object-Oriented Design in Ruby (Sandy Metz)
- Confident Ruby: 32 Patterns for Joyful Coding (Avdi Grimm)

## Ruby on Rails: Geschichte

- 2004: Entwickelt als Basis für *Basecamp*
- Version 1.0 (2005)
- Version 1.1 (2006) - Scripting Engines, Performance
- Version 1.2 (2007) - REST Support, MIME-type routing, UTF-8
- Version 2.0 (2007) - REST als Default
- Version 2.1 (2008) - Named Scopes, Migrationen mit Timestamp
- Version 2.2 (2008) - Internationalisierung, Threadsicherheit
- Version 2.3 (2009) - Template Engine
- Version 3.0 (2010) - Modularisierung: Einzelne Teile verwendbar
- Version 3.1 (2011) - Asset Pipeline
- Version 4.0 (2013)
- Version 5.0 (2016)

## Ruby on Rails: Bestandteile

- Active Support: Ruby-Erweiterungen von Rails
- Active Record: Objektabstraktionsschicht (ORM)
- Action Pack: Request-Behandlung und Response-Ausgabe.
- Action View: Templates
- Action Mailer: E-Mail-Versand und -Empfang
- Active Resource: Routing, REST, XML-RPC

## Ruby on Rails: Grundlagen

- Basiert auf Ruby
- Model-View-Controller Architektur
- „Don't repeat yourself“
- „Convention over Configuration“
- Scaffolding
- Datenbankmigrationen



## Ruby on Rails: Grundlagen

### „Don't repeat yourself“

- Jede Information sollte nur ein einziges Mal vorhanden sein
- z.B. ActiveRecord liest das DB-Scheme direkt aus der DB
- z.B. Rails erstellt für das Model automatisch Getter- und Setter-Methoden
- Vorteil: Informationen werden nicht inkonsistent wenn eine Stelle verändert wird

## Ruby on Rails: Grundlagen

### „Convention over configuration“

- Rails erwartet sinnvolle Standardwerte
- --> z.B. Primärschlüssel einer Tabelle ist ID vom Typ Integer
- --> ein Modell mit dem Namen Customer liegt in der Datei #  
{Rails.root}/app/models/customer.rb
- --> Die zugehörige Tabelle heißt customers

## Ruby on Rails: Grundlagen

# Scaffolding

- Es gibt Generatoren für alle Standardfälle
- Models, Controller, Views, Mailer, Migrationen, ...
- Konventionen werden eingehalten
- Web-Anwendungen lassen sich sehr schnell entwickeln
- Wenn in der Datenbank etwa ein Feld hinzugefügt wird, erscheint es auch sofort in der entsprechenden View/New/Edit-Ansicht.
- Scaffolding ist vor allem für Prototyping gedacht

Ruby on Rails: Grundlagen

## Webserver-Unterstützung

- Integrierter Application-Server: WEBrick
- Für Produktion: Apache, Nginx, Lighttpd, ... z.B. mit Phusion Passenger

## Ruby on Rails: Grundlagen

### Schichten: Model

- Typischerweise basierend auf einer relationalen Datenbank
- Zugriff mit Hilfe von ActiveRecord hergestellt (ORM-Schicht)
- --> Klasse <> Tabelle
- --> Attribut <> Spalte
- --> Objekt <> Zeile
- Support für: SQLite, DB2, Informix, MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Sybase, ...
- Alternativen: Sequel, Datamapper, XML, ...

Ruby on Rails: Grundlagen

## Schichten: Controller

- Basiert auf *ActionController*, Bestandteil von *ActionPack*
- Ein Controller kapselt die Geschäftslogik im Model von der Darstellung der View
- Ruby-Code

## Ruby on Rails: Grundlagen

### Schichten: View

- Präsentationsschicht *ActionView*, Bestandteil von *ActionPack*
- Diverse Ausgabeformate/Template-Engines werden unterstützt:
- -> HTML. ERB, Slim, Haml, Sass, CoffeeScript, ...
- -> XML – z. B. für XHTML und Web Services
- -> JSON
- -> JavaScript – RJS-Templates
- -> Binärdaten

## Ruby on Rails: Ein Beispiel

- [https://github.com/railstutorial/sample\\_app\\_rails\\_4](https://github.com/railstutorial/sample_app_rails_4)
- <https://www.railstutorial.org/>
- <https://www.railstutorial.org/book>



# Fragen?

Unterlagen: [ai2018.nils-loewe.de](http://ai2018.nils-loewe.de)