Architekturstile und -muster

Ein Katalog von Konzepten

Burkhardt Renz

Fachbereich MNI
Technische Hochschule Mittelhessen

Wintersemester 2020/21

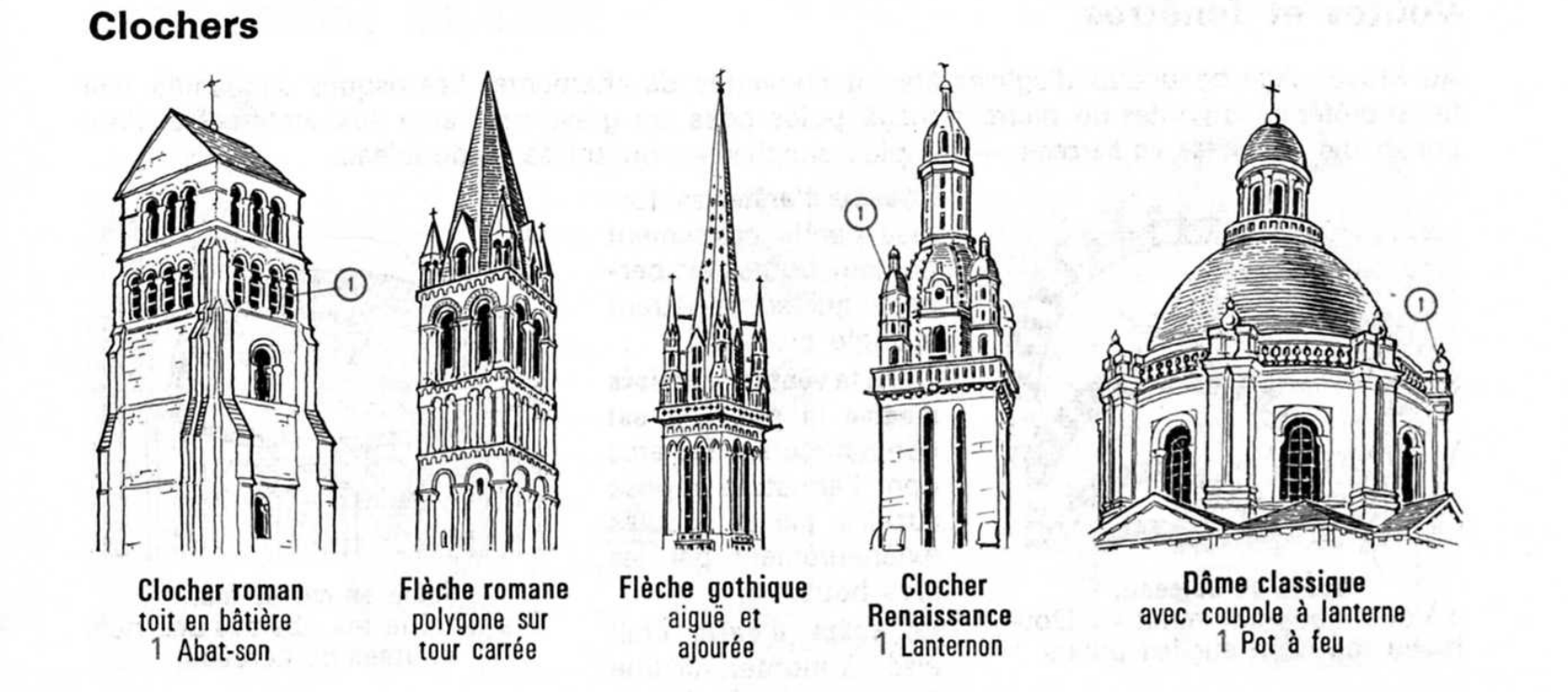
Übersicht

- Architekturstile und -muster
  - Was sind Architekturstile?
  - Arten von Komponenten und Konnektoren
  - Übersicht wichtiger Stile
  - Beschreibung wichtiger Stile
- Kombination von Stilen — am Beispiel der Architektur des Webs
- Beispiele von Architekturen
- Literatur & Links

Architekturstile und -muster

Was sind Architekturstile?

Was sind Architekturstile?



Was sind Architekturstile?

Bestandteile der Architektur

- Komponenten
- Konnektoren
- Topologie

Architekturstil

- Abstraktion von den Spezifika einer konkreten Architektur
- Leitlinie für Art der Komponenten und Konnektoren – und ihr Zusammenspiel
- Architektonische „Constraints“ bezüglich des Zusammenwirkens von Komponenten und Konnektoren
- Paradigma, Rahmen für Entwurfsentscheidungen

Abgrenzung Architekturstil und -muster

Architekturstil

Ein Architekturstil ist ein Festlegung der Rollen oder Features von Komponenten und der erlaubten Formen der Konnektoren zwischen den Komponenten.
Also: Constraints bezüglich der Komponenten und Konnektoren und ihr Zusammenspiel

Architekturmuster

Ein Architekturmuster beschreibt eine Lösung eines architektonischen Problems in einem bestimmten Kontext. Zur Beschreibung gehört auch die Begründung, wie das Problem gelöst wird.
Also: Problemstellung und die sie beeinflussenden Kräfte sollen eine Lösung bekommen.

Auswirkungen von Architekturstilen und -mustern

- prinzipielle Festlegung der Aufbaustruktur
- *zugleich:* Beschränkung der Art der Elemente und ihrer Verbindung
- *damit:* Festlegung der spezifischen Eignung und Qualitäten eines Systems

Was diese Festlegungen angeht, sehe ich Architekturstile und Architekturmuster als so verwandt an, dass ich sie gemeinsam betrachte.

Architekturstile und -muster

Arten von Komponenten und Konnektoren

Arten von Komponenten

Berechner führt eine Art Berechnung aus; nur lokaler Zustand während der Berechnung; kein Gedächtnis; z.B. Funktion, Filter

Manager verwaltet Zustand, auch über mehrere Aufrufe hinweg; bietet Methoden, um den Zustand zu beeinflussen; z.B. Server, zustandsorientierte Objekte

Controller steuert die (zeitliche) Abfolge von Aktionen, reagiert entsprechend auf Ereignisse, z.B. Scheduler, Eventhandler

Speicher enthält persistente Daten in strukturierter Form; Zugriff verschiedener anderer Komponenten möglich; z.B. Datenbanken, Dateien

Kanal Ort, an dem Daten beobachtbar sind; Bestandteile des Datenflusses, z.B. Pipe, Funktionsparameter

Arten von Konnektoren

Funktionsaufruf Gemeinsamer Adressraum; Kontrolle geht an die aufgerufene Komponente und kehrt nach „Erledigung“ zurück.

Datenfluss Prozesse interagieren durch einen Datenstrom z.B. Pipe; Komponenten sind unabhängig

Impliziter Aufruf Komponente wird durch ein Event aktiviert; sie weiß nicht, wer den Dienst benötigt

Nachrichtenaustausch Unabhängige Prozesse interagieren durch Protokoll; kann synchron sein z.B. Request/Response oder asynchron z.B. Message Queue

Gemeinsame Daten Komponenten arbeiten auf denselben Daten; Steuerung durch Datenzustand; Synchronisation; Beispiel Repository

Architekturstile und -muster

Übersicht wichtiger Stile

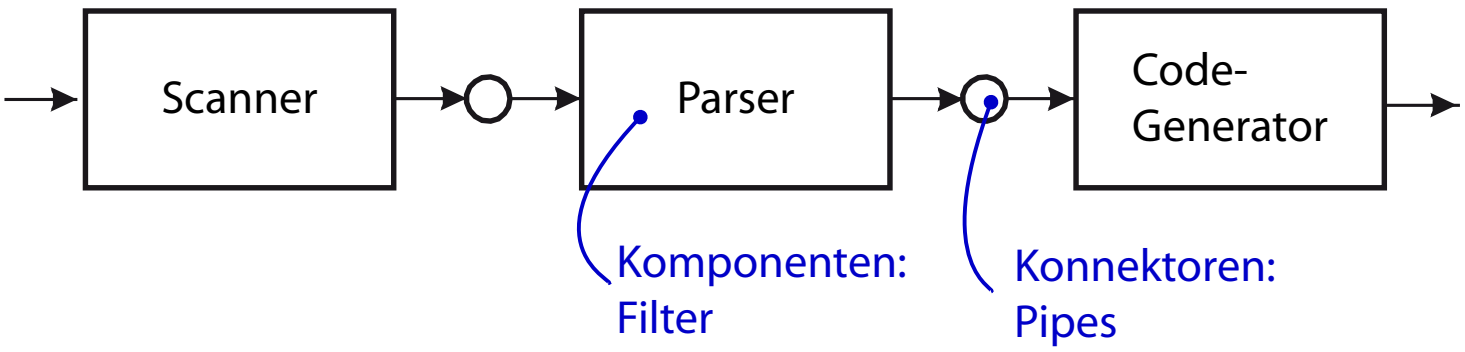
Übersicht wichtiger Architekturstile

- Datenflusssysteme
  - Batch sequenziell
  - Pipes und Filter
- Kontrollfluss
  - Funktionale Komposition
  - Objektorientierte Organisation
  - Schichten
- Benutzerinteraktion
  - MVC Model-View-Controler
  - PAC Presentation-Abstraction-Control
- (Unabhängige) Komponenten
  - Kommunizierende Prozesse
  - Ereignisgesteuerte Systeme
  - Plugin-Architektur
- Datenzentrierte Systeme
  - Repositories
  - Blackboard
- Virtuelle Maschinen
  - Interpreter
  - Regelbasierte Systeme

Architekturstile und -muster

Beschreibung wichtiger Stile

Pipes und Filter



Pipes und Filter – Charakteristik

Komponenten Filter transformieren Eingabeströme in Ausgabeströme

Konnektoren Pipes bewegen und puffern Daten zwischen den Filtern

Struktur Datenfluss zwischen unabhängigen Komponenten; Schnittstelle durch Input- und Output-Formate definiert

Systemmodell Kontinuierlicher Datenfluss zwischen Komponenten, die inkrementell die Datenströme transformieren

Pipes und Filter – Diskussion

Varianten

- Lineare Folge von Filtern: Pipeline
- Spezielle Pipes: Formatprüfung
- Push-Methode z.B. Unix-Pipes
- Pull-Methode z.B. yacc ruft lex
- Filter teilen gemeinsame Daten z.B. Symboltabelle eines Compilers

Pipes und Filter – Vor- und Nachteile

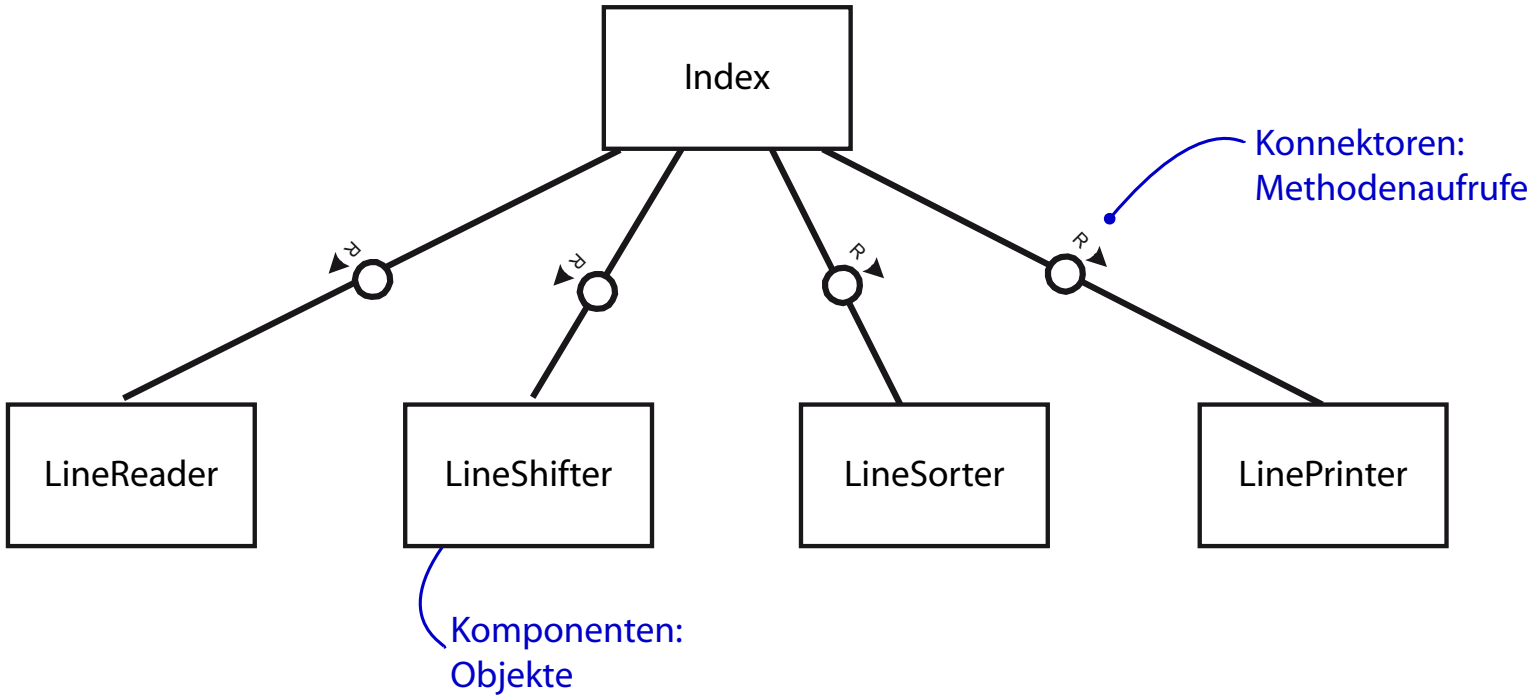
Vorteile

- Einfach zu kombinieren und wiederzuverwenden
- Einfach auszutauschen
- Effizient durch Parallelität
- Leicht zu analysieren und zu testen

Nachteile

- Fehlerbehandlung
- Zugriff auf gemeinsame Daten?
- Kopplung durch Datenformate, Kosten für Datentransfer
- Skalierbarkeit
- kaum geeignet für interaktive Anwendungen

Objektorientierte Organisation



Objektorientierte Organisation – Charakteristik

Komponenten	Objekte, die Daten kapseln; Objekte, die als Manager und Controller agieren
Konnektoren	Methodenaufrufe
Struktur	Kollaborierende Objekte
Systemmodell	Komponenten verwalten „ihre“ Daten, bündeln Verantwortlichkeit für Daten und Funktionalität

Objektorientierte Organisation – Diskussion

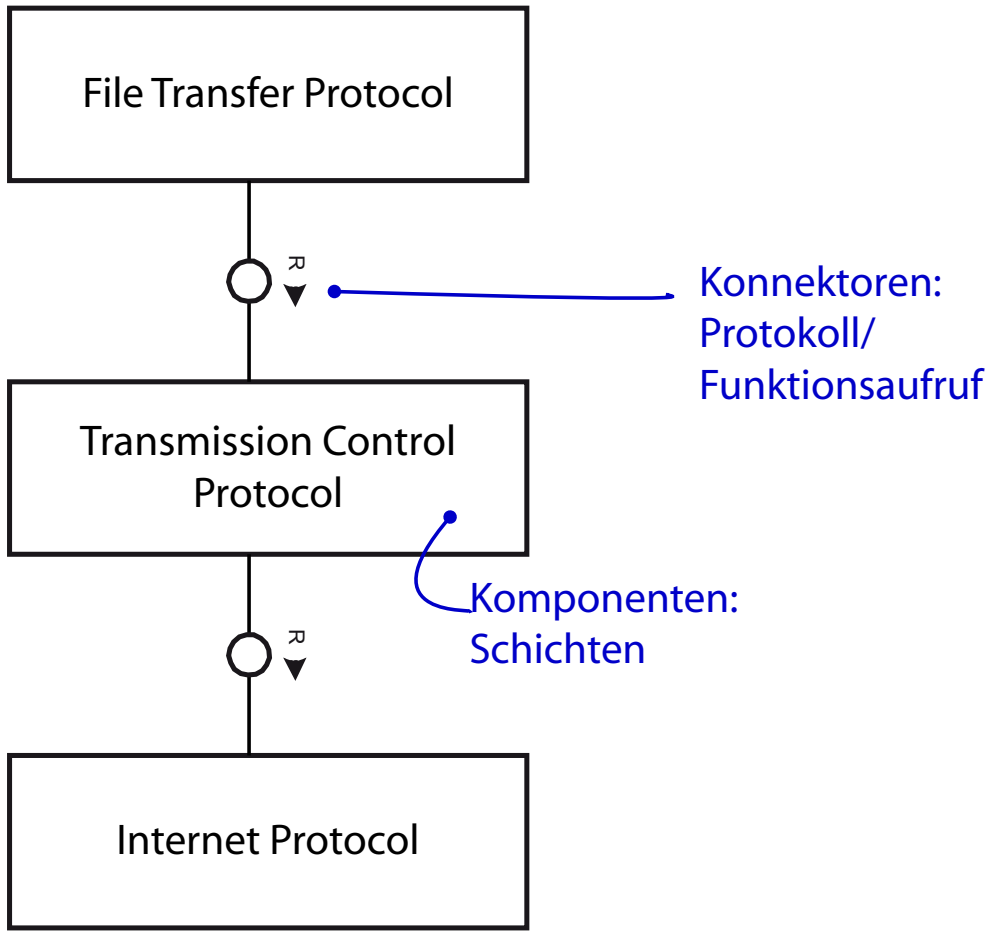
Vorteile

- Trennung Implementierung – Verwendung erleichtert Austauschbarkeit
- Wiederverwendbarkeit, etwa Klassenbibliotheken
- Wiederverwendbarkeit in neuem Kontext durch dynamischen Polymorphismus („Alter Code ruft neuen Code“)

Nachteile

- Verwender müssen Objektidentität kennen (durch Dependency Injection beherrschbar)
- Zerfaserung von Funktionalität auf viele Objekte
- Zustand der Anwendung schwer kontrollierbar

Hierarchische Schichten



Hierarchische Schichten – Charakteristik

Komponenten In Schichten organisierte Komponenten, die verschiedenen Typs sein können – virtuelle Maschinen

Konnektoren Funktionsaufrufe zu darunterliegenden Schichten – mögliche Benutzung

Struktur Hierarchie

Systemmodell Jede Schicht stellt eine Abstraktion bereit, wodurch der Verwender einer Schicht getrennt ist von Details der Implementierung.
Der Verwender nutzt den Dienst einer Schicht (einer virtuellen Maschine).

Hierarchische Schichten – Diskussion

Sprechweisen

- Schichten als Separierung von Verantwortlichkeit:
  insbesondere bei interaktiven Systemen z.B. Trennung Präsentation – Anwendungslogik – Persistenzmechanismus
- Schichten als Konzept der Modellierung: insbesondere in UML-basierte Konzepten z.B. Trennung «`boundary`» – «`controller`» – «`entity`»
- Schichten als physische Verteilung:
  2-tier, 3-tier und n-tier Architekturen
- Hierarchische Schichten (im engeren Sinne):
  etwa Hardware Abstraction Layers u.ä. – Strukturierung des Codes und der Aufrufhierarchie

Hierarchische Schichten – Diskussion

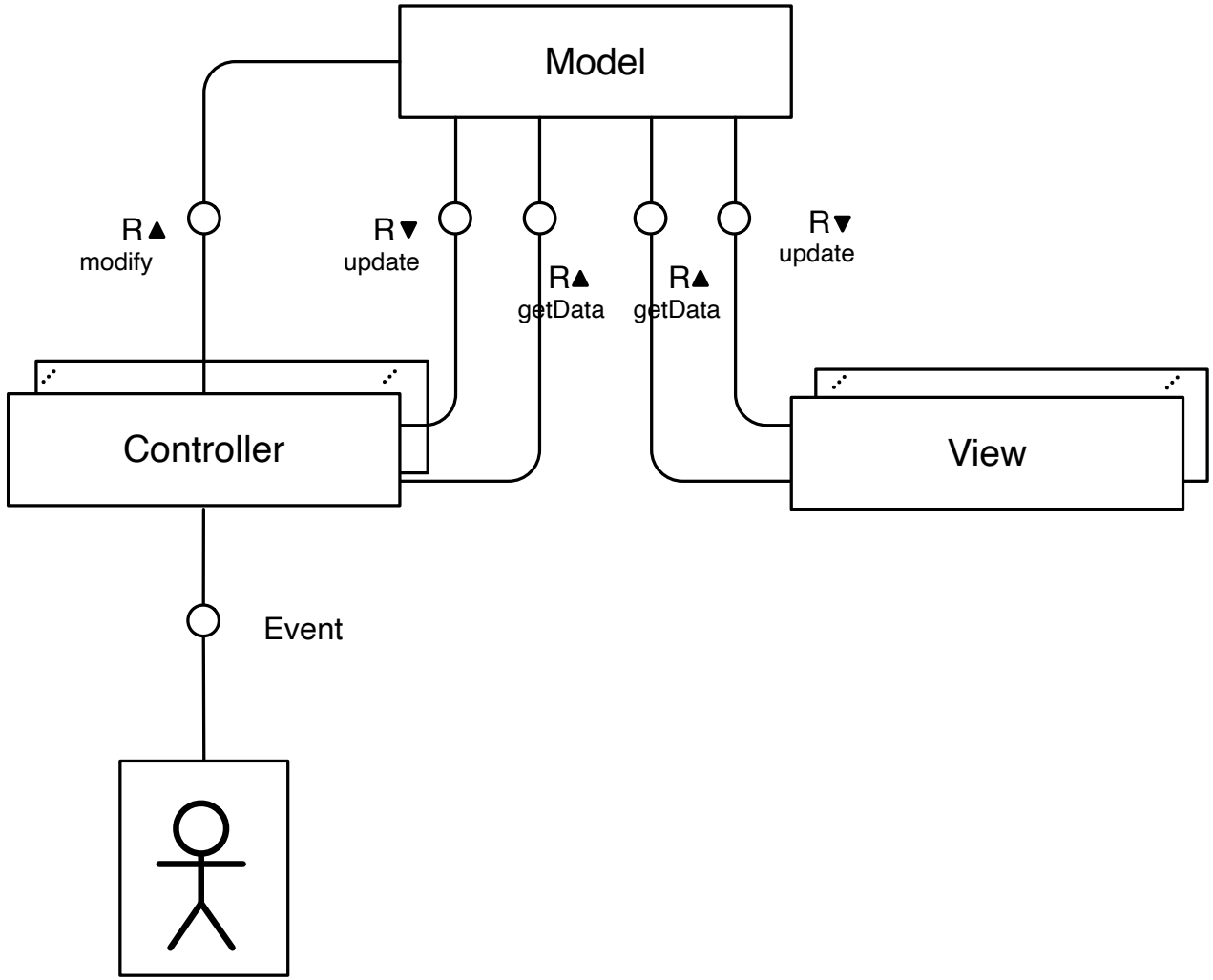
Vorteile

- Abstraktion
- Lokalisierung von Entscheidungen in Schicht – leichte Änderbarkeit
- Austauschbarkeit von Schichten

Nachteile

- Geeignete Abstraktionen schwierig zu finden
- Viele Indirektionen – Folge: *layer bridging*

MVC — Model-View-Controler



MVC — Charakteristik

Komponenten Das *Model* kapselt Anwendungsdaten unabhängig von den Views
Die *Views* präsentieren dem Benutzer Anwendungsdaten (in verschiedenen Formen)
Die *Controler* der Oberfläche erlauben es dem Benutzer die Anwendungsdaten zu verändern

Konnektoren Funktionsaufrufe

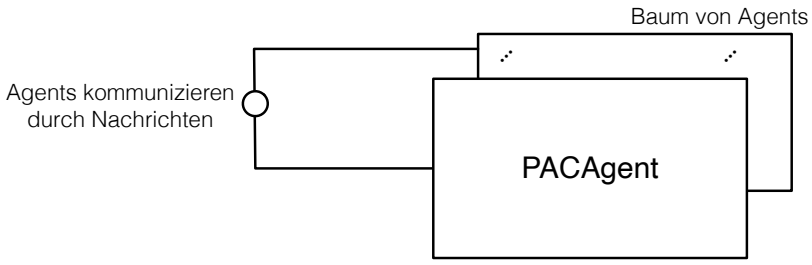
Struktur Objekt-Netz

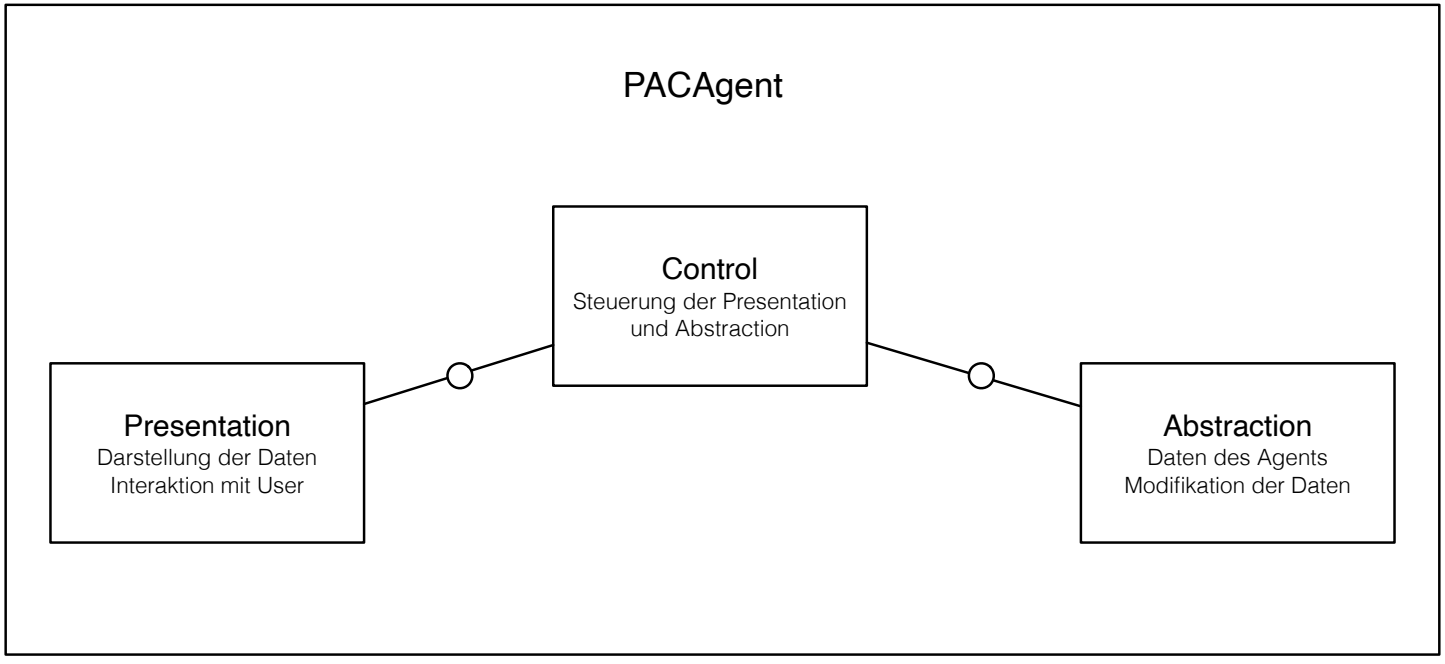
Systemmodell *Synchronisation* der Änderungen in den Views durch gemeinsame Daten im Model

MVC — Diskussion

- Meist implementiert mittels des Entwurfsmusters *Observer*
- Gibt es in vielen Varianten
- Vorsicht: Inflationärer Gebrauch des Etiketts „MVC“ führt zu Verwirrung – Jeder versteht etwas anderes darunter

PAC — Presentation-Abstraction-Control





PAC — Charakteristik

Komponenten Hierarchisch organisierte Agents, die jweils eine spezielle Funktionalität haben oder gruppieren.
Jeder Agent besteht aus

- Die *Presentation* ist das User-Interface
- Die *Abstraction* enthält die Daten des Agents
- Die *Control* steuert die Interaktion zwischen P und A, sowie mit den anderen Agents der Hierarchie

Konnektoren Funktionsaufrufe innerhalb der Agents, Nachrichten zwischen den Agents

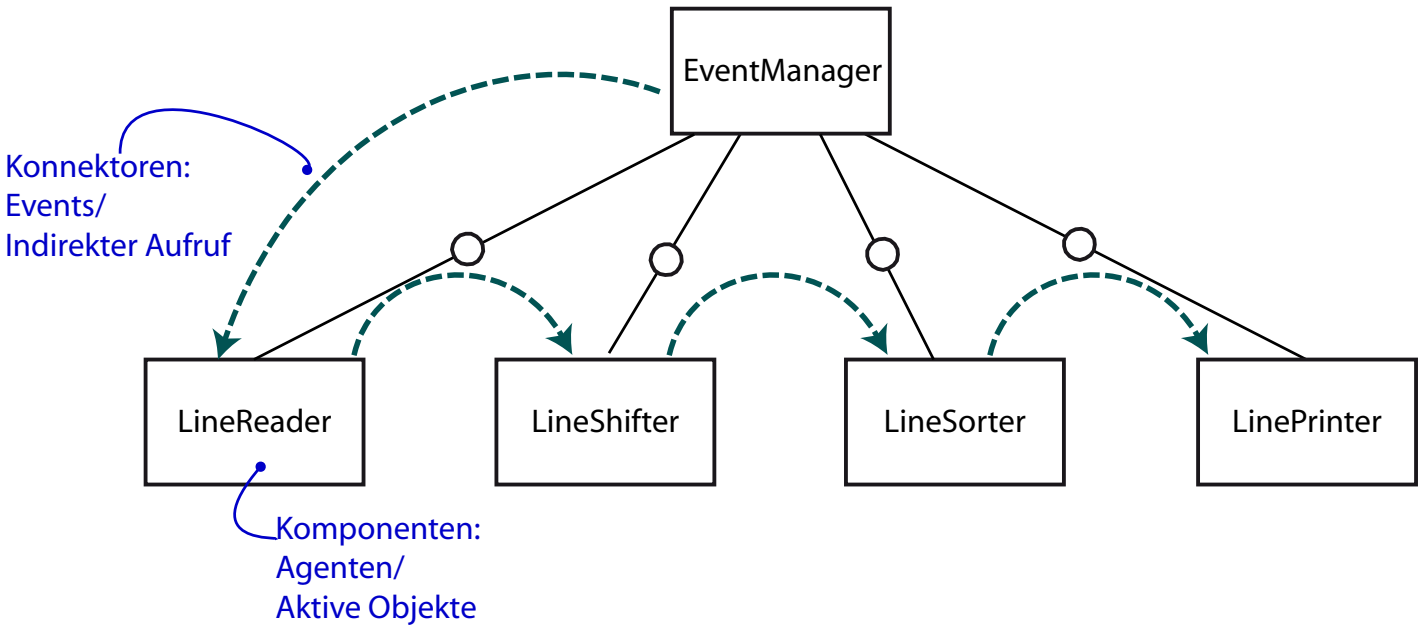
Struktur Hierarchie von Agents

Systemmodell Jeder Agent ist für eine spezielle Funktionalität zuständig
Übergeordnete Agents kombinieren die Funktionalität ihrer Kinder
Änderungen von Anwendungsdaten werden durch die Hierarchie propagiert

PAC — Diskussion

- Jeder Agent kann als eine Variante von MVC aufgefasst werden
- Es geht aber nicht wie bei MVC in erster Linie um die Synchronisation verschiedener Sichten auf dieselben Daten, sondern um die gleichartige Organisation und Präsentation unterschiedlicher Funktionalität
- Trennung der Belange – verschiedene Konzepte im Anwendungsgebiet können durch getrennte Agents repräsentiert werden
- Neue Agents können leicht eingebaut werden – wegen des Nachrichtenkonzepts zwischen den Agents
- Aber: Komplexes System

Ereignisgesteuerte Systeme



Ereignisgesteuerte Systeme – Charakteristik

Komponenten Prozesse, Threads, aktive Objekte, die Events erzeugen und auf Events reagieren.

Konnektoren Die Komponenten registrieren sich für die Benachrichtigung, wenn bestimmte Events auftreten. Das Event führt so zum impliziten Aufruf der Komponente

Struktur Unabhängige, nicht zentral gesteuerte Komponenten - benötigt ein Framework für den Eventmechanismus

Systemmodell Die Komponenten sind nicht durch direkte Referenzen gekoppelt, sondern reagieren auf Events. Häufig sind die Komponenten nebenläufig. Reihenfolge der Aufrufe nicht deterministisch.

Ereignisgesteuerte Systeme – Diskussion

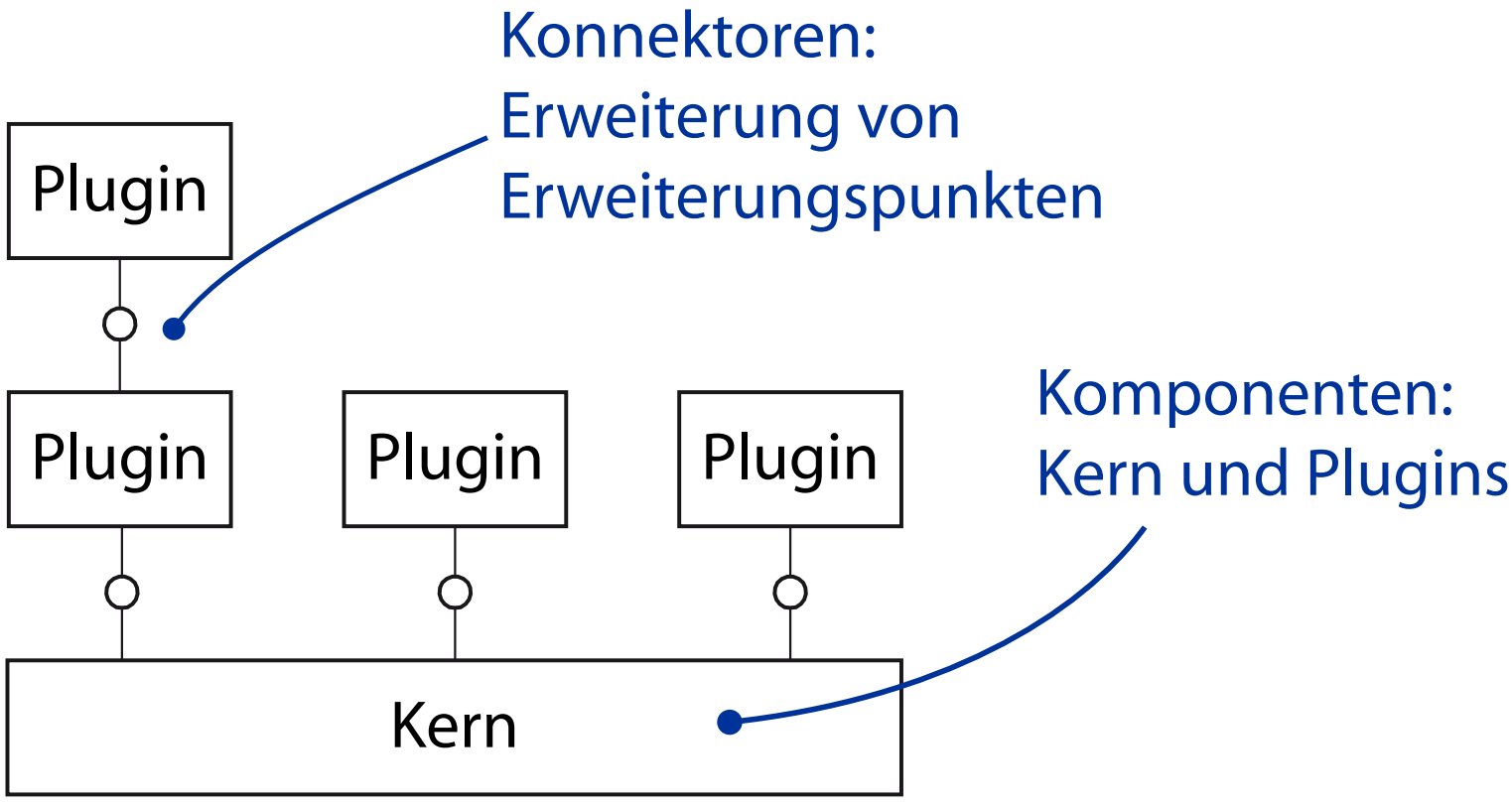
Vorteile

- Austauschbarkeit der Komponenten zur Laufzeit
- Evolution des Systems durch Veränderung und Erweiterung einzelner Komponenten, ohne andere zu tangieren

Nachteile

- Kontrollfluss nur schwer zu verfolgen
- Komponenten müssen Infrastruktur und Protokolle gemeinsam haben
- Fehlerbehandlung in Bezug auf die Interaktion der Komponenten
- Zugriff der Komponenten auf gemeinsam verwendete Daten?

Plugin-Architektur



Plugin-Architektur – Charakteristik

Komponenten der Kern,
die Plugins

Konnektoren Erweiterung von Erweiterungspunkten des Kerns oder von anderen Plugins

Struktur Baumartige Struktur von sich erweiternden Plugins

Systemmodell Vertrag zwischen Angebot zur Erweiterung und Nutzung des Angebots.

Plugin-Architektur – Diskussion

Beispiele

- OSGi Java-Bundles und Services
- Eclipse und Eclipse Rich Client Platform

Arten

- Spezieller Vertrag zwischen Wirt und Plugin, Wirt sucht Plugin – z.B. .NET Assemblies
- Plugin–Registry wie bei Eclipse
- Zeitpunkt der Plugin-Aktivierung: Startzeitpunkt, zur Laufzeit

Plugin-Architektur – Diskussion

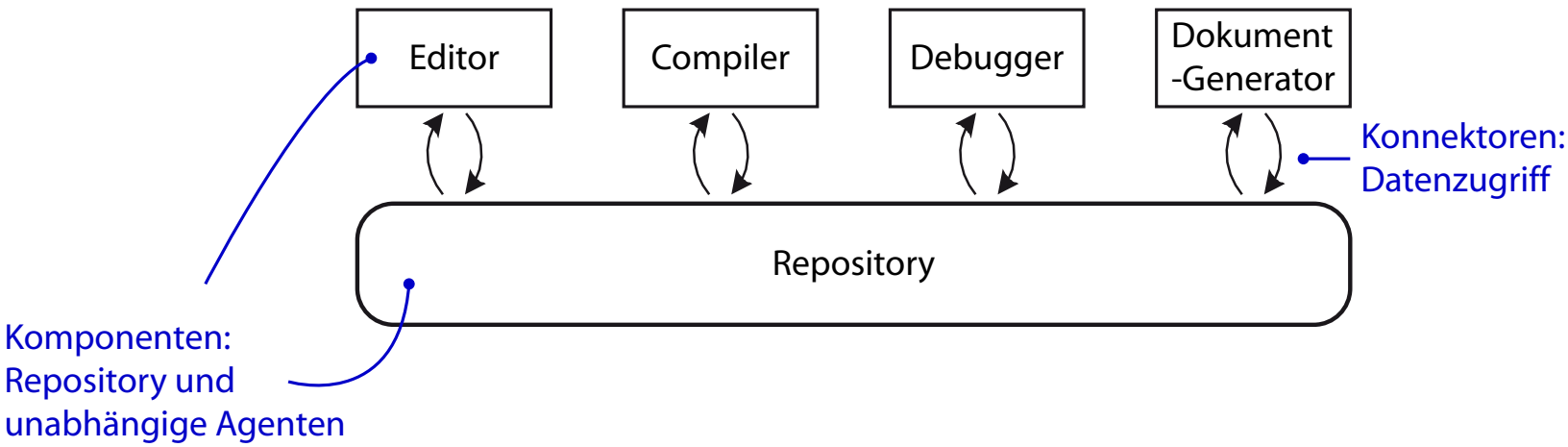
Vorteile

- Erweiterung – ohne Modifikation des vorhandenen Systems
- Verbesserte Wartbarkeit durch Austausch von Plugins
- Wiederverwendung von Plugins (auch aus anderen Anwendungen)

Nachteile

- Starke Kopplung zwischen Komponenten kann leicht entstehen
- Fehlerhaftes Plugin kann Stabilität des Systems gefährden
- Subtile Abhängigkeiten zwischen Plugins können entstehen

Repository


Editor
Compiler
Debugger
Dokument
-Generator
Konnektoren:
Datenzugriff
Repository
Komponenten:
Repository und
unabhängige Agenten

Repository – Charakteristik

Komponenten Eine Speicherkomponente und viele Agenten, die auf diese Daten zugreifen

Konnektoren Der Zugriff auf die Speicherkomponente verbindet die verschiedenen Agenten

Struktur Die Steuerung ergibt sich durch die Entscheidungen der Agenten, die sie auf Grund der Datenlage in der Speicherkomponente treffen

Systemmodell Zentralisierte Daten sind die Basis der autonomen Entscheidung von Agenten, die diese Daten lesen und verändern. Datenintegration als wichtigstes Ziel

Repository – Diskussion

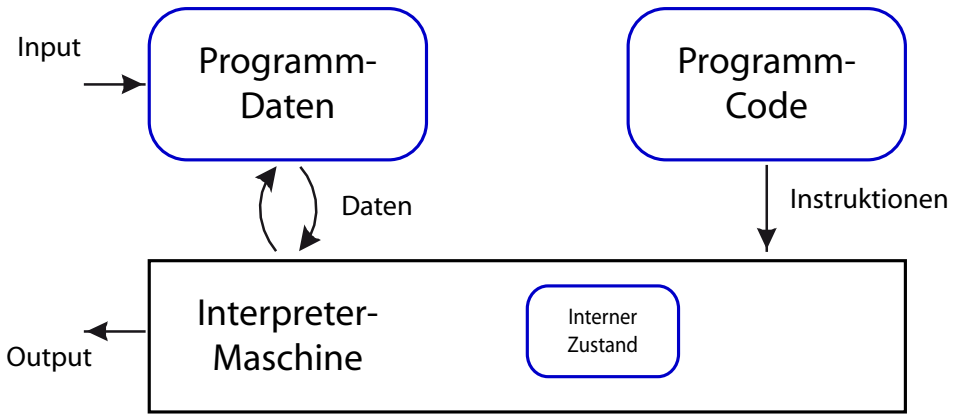
Vorteile

- Austauschbarkeit von Agenten
- Kontrolle der Daten durch zentrale Datenhaltung

Nachteile

- Repository kann zum Engpass werden
- Fehlerbehandlung durch Unabhängigkeit der Agenten erschwert
- Kontrollfluss evtl. schwer zu überblicken

Interpreter



Komponenten der virtuellen Maschine

Konnektoren: Zugriff auf Programmdaten und -instruktionen

Interpreter – Charakteristik

Komponenten Komponenten sind:
die Maschine,
der Programmcode (beinhaltet den Kontrollfluss) und
der Speicher der Maschine (enthält die Daten)

Konnektoren Interner Ablauf des Interpreters

Struktur Metalevel: hohe Dynamik durch Steuerung des Verhaltens zur Laufzeit

Systemmodell Steuerung des Interpreters durch eine *Sprache*

Interpreter – Diskussion

Beispiele

- Java Virtual Machine
- SQL-Interpreter in Datenbanksystemen
- Dynamische interaktive Oberflächen erzeugt aus Beschreibungen
- ABAP in SAP R/3 - interpretierte Sprache mit spezialisierten Konstrukten im Zugriff auf Repository

Arten

- Datensteuerung – einfach Formen von Interpretern durch Konfiguration zur Laufzeit
- Domänenspezifische Sprachen
- Alternative: Codegenerierung

Interpreter – Diskussion

Vorteile

- Sprache - gut verstanden, gut wartbar, gut erweiterbar
- Sprache - mächtig
- Testbarkeit wegen Einsatz einer Sprache
- Domänenspezifika können direkt formuliert werden

Nachteile

- Leistungsfähigkeit – kompensierbar durch Objektserialisierung, Zwischencode, JIT Compiler
- Sicherheitsprobleme durch Interpreter häufig: SQL Injection, Windows Scripting
- aufwändig, einmal konstruierte Sprache nur schwer später zu revidieren

Literatur

Mary Shaw, David Garlan
Software Architecture: Perspectives on an Emerging Discipline
Upper Saddle River, NJ: Prentice-Hall, 1996.

Paris Avgeriou, Uwe Zdun
Architectural Patterns Revisited – A Pattern Language
EuroPLoP 2005.

Übersicht

- Architekturstile und -muster
- Kombination von Stilen — am Beispiel der Architektur des Webs
- Beispiele von Architekturen
- Literatur & Links

Das Konzept der Herleitung einer Architektur

Architekturstil nach Fielding

„An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.“ [Fielding 1.5]

Kombination von Stilen

„New architectures can be defined as instances of specific styles. Since architectural styles may address different aspects of software architecture, a given architecture may be composed of multiple styles. Likewise, a hybrid style can be formed by combining multiple basic styles into a single coordinated style.“ [Fielding 1.5]

Die Architektur des Webs

Komponenten Server mit Ressourcen (z.B. Apache), Gateways, Proxies, Klienten wie Browser, Search Bots

Konnektoren APIs zum Zugriff auf http-Server wie libwww, Serverseitige APIs wie Apache API, Servlet API ...

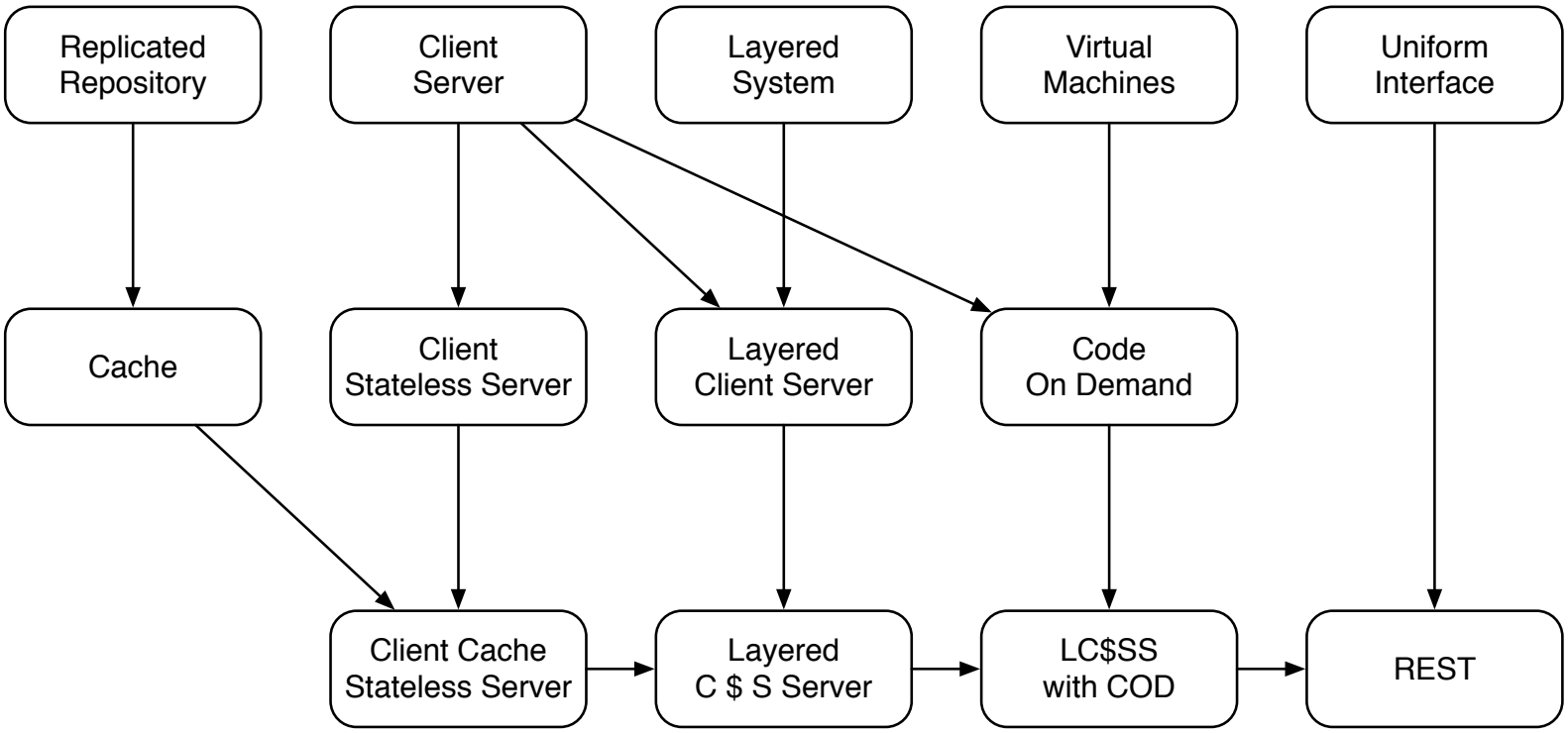
Daten Ressourcen, Identifizierer (URL), Repräsentation der Ressource (z.B. als HTML, XML, JPG etc), Metadaten, Steuerdaten.

Topologie Viele Klienten, viele Server, Intermediäre/Proxies auf beiden Seiten.

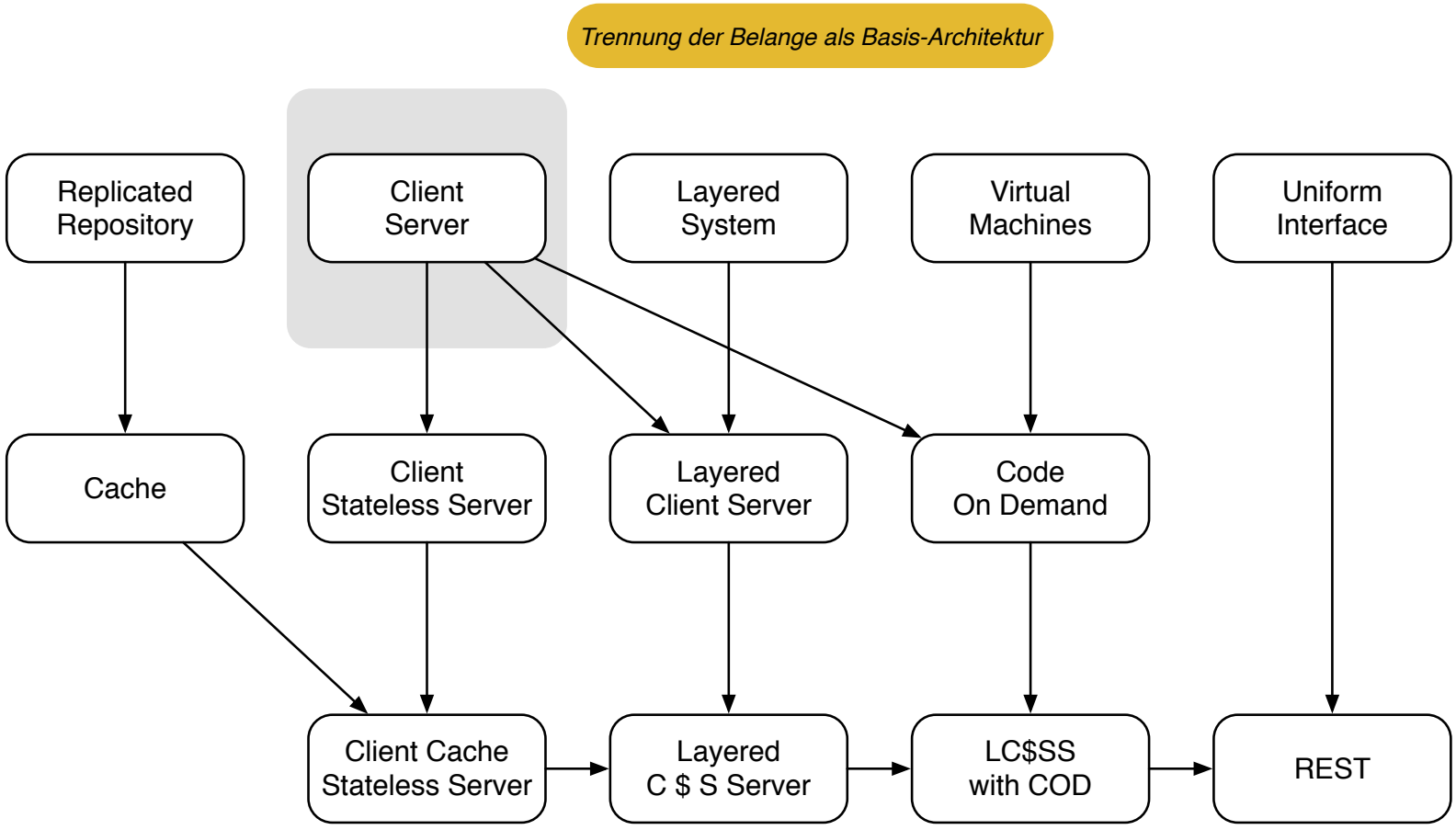
REST: Representational State Transfer

- Die Schlüsselabstraktion ist die *Ressource* – eine identifizierbare Informationseinheit.
- Die Repräsentation einer Ressource ist eine *Bytefolge* zusammen mit *Metadaten* zu ihrer Beschreibung; sie hat einen bestimmten Typ: etwa HTML, JPG o.ä..
- Alle Interaktionen sind kontextfrei.
- Komponenten verwenden eine kleine Menge wohldefinierter Methoden: get, put, post, delete ...
- Operationen sollen möglichst idempotent sein.
- Intermediäre und Caches sind erwünscht.

Herleitung von REST



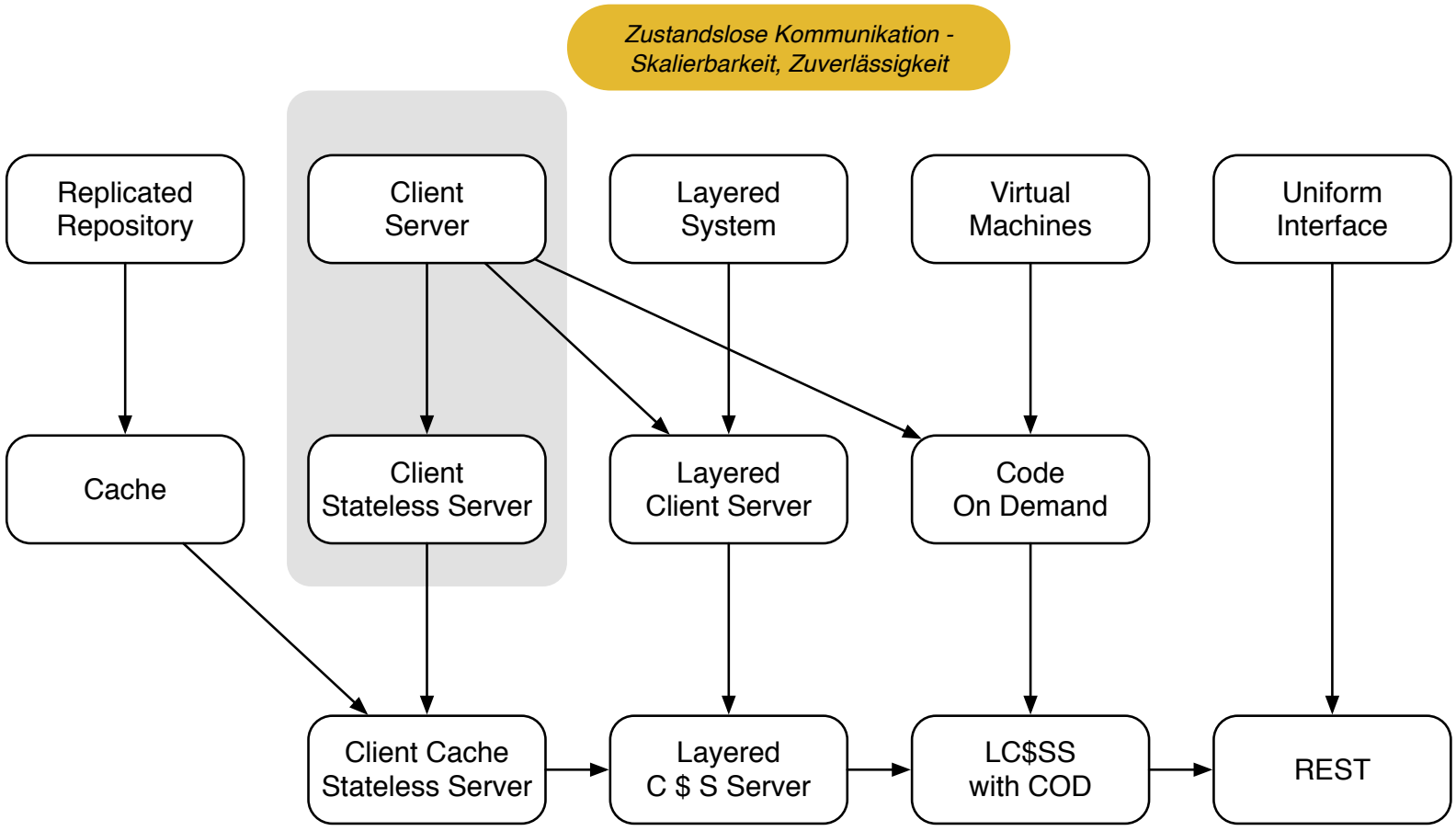
Herleitung von REST



Client-Server

- Trennung der Belange: Trennung der Verwendung von Daten von ihrer Speicherung
- Portabilität der Benutzerschnittstelle im Client über verschiedene Plattformen hinweg
- Skalierbarkeit durch die Vereinfachung der Server-Komponenten
- Insbesondere: Möglichkeit der unabhängigen Entwicklung der Komponenten in Organisationen, die sich nicht absprechen müssen oder können

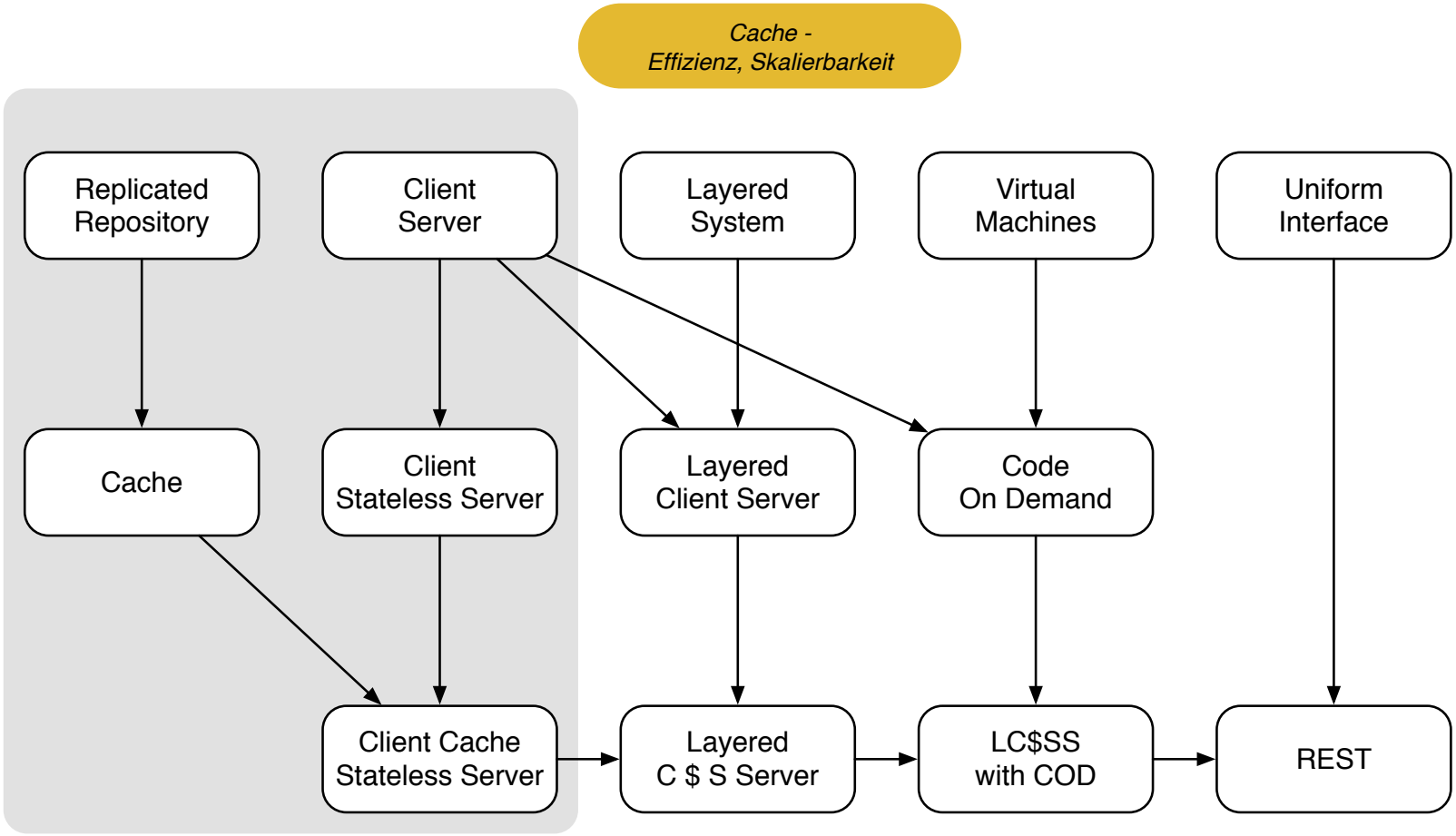
Herleitung von REST



Stateless

- Jeder Request des Clients soll alle Informationen enthalten, die der Server für das Erstellen der Response benötigt – Zustand ist vollständig auf der Client-Seite
- Testbarkeit/Überprüfbarkeit: Jeder Request/Response kann für sich betrachtet werden
- Zuverlässigkeit: Wenn ein Server ausfällt, kann ein anderer die Aufgaben übernehmen ohne dass Zustandsinformation verloren gegangen wäre
- Skalierbarkeit: Man kann serverseitig Ressourcen frei managen, da keine Zustandsinformation benötigt wird
- Allerdings: mehr Daten müssen im Netz übermittelt werden
- Allerdings: Server kann den Zustand des Clients nicht kontrollieren

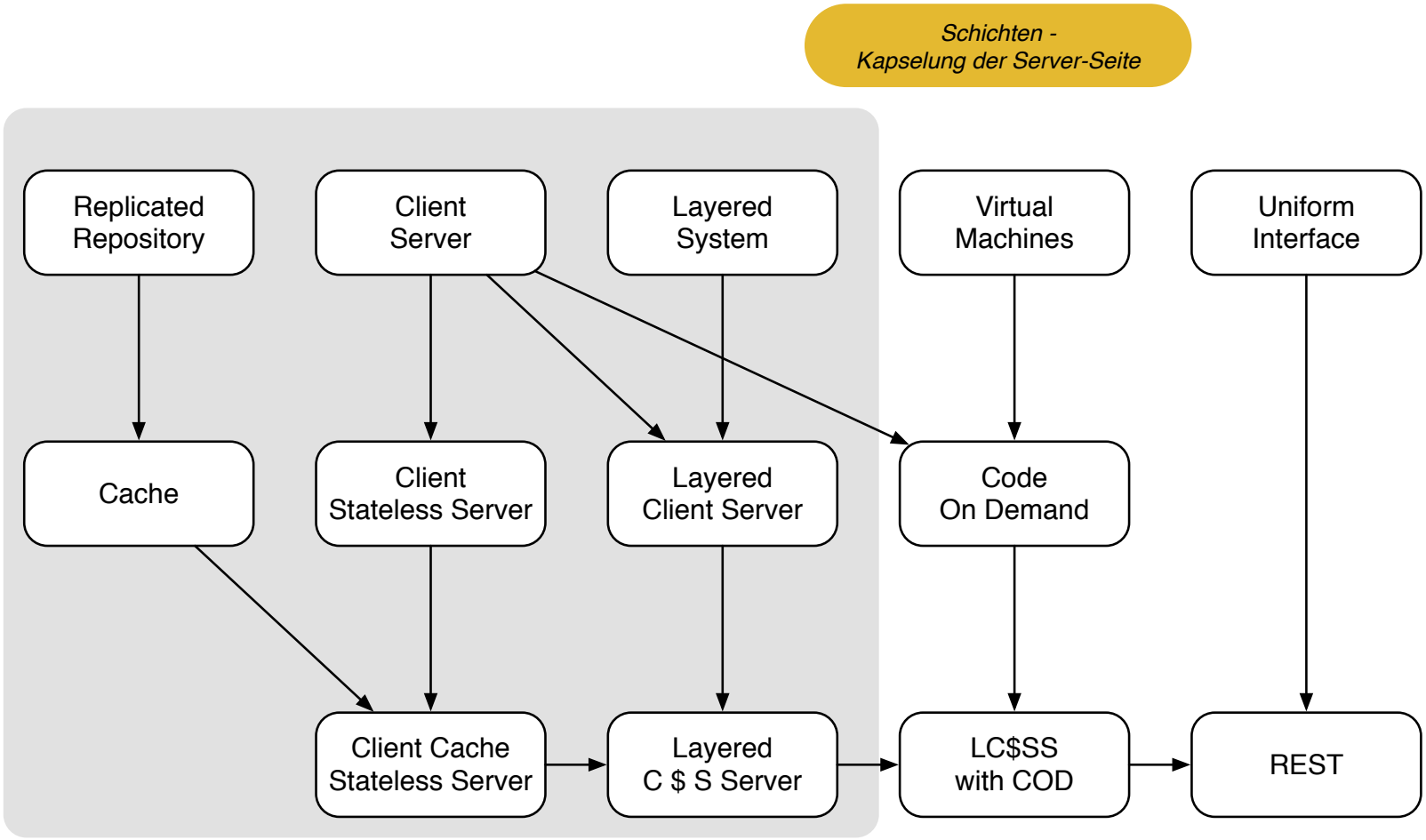
Herleitung von REST



Cache

- Daten können als „cacheable“ gekennzeichnet werden können und deshalb zwischengespeichert werden können
- Cache kann client-seitig sein oder bei Intermediären
- Vorteil: Verringerung des Datentransfers im Netz
- Vorteil: Beschleunigung des Antwortverhaltens beim Client, Verringerung der Latenzzeit
- Nachteil: Daten können veraltet sein

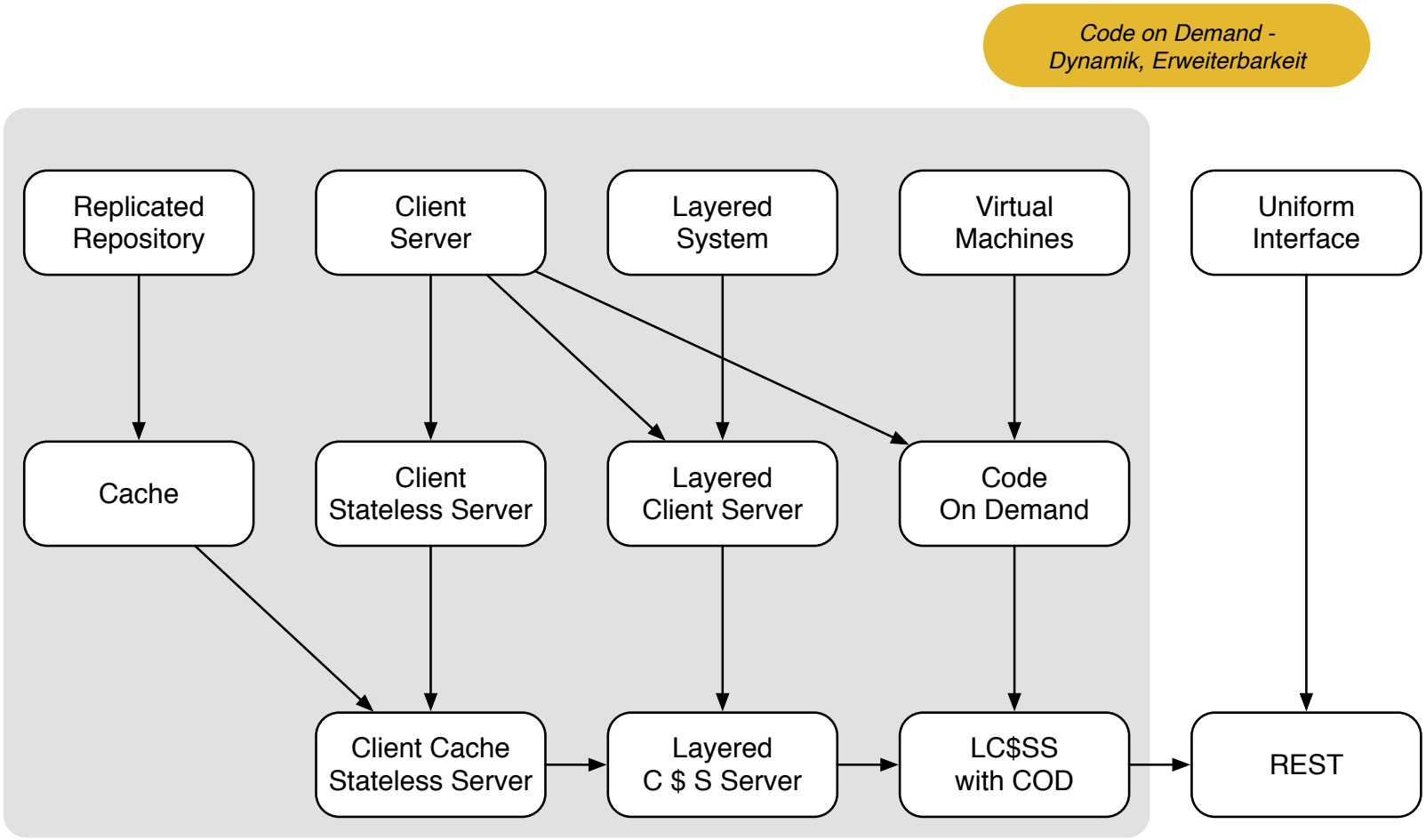
Herleitung von REST



Schichten

- Bessere Verteilung der Auslastung, Load Balancing
- Skalierbarkeit durch Intermediäre
- Leichte Möglichkeit Systeme in das Netz zu integrieren
- Verarbeitung der Daten kann ähnlich wie bei Pipes & Filters auf Komponenten aufgeteilt werden
- Aber: Erhöhung der Latenzzeit

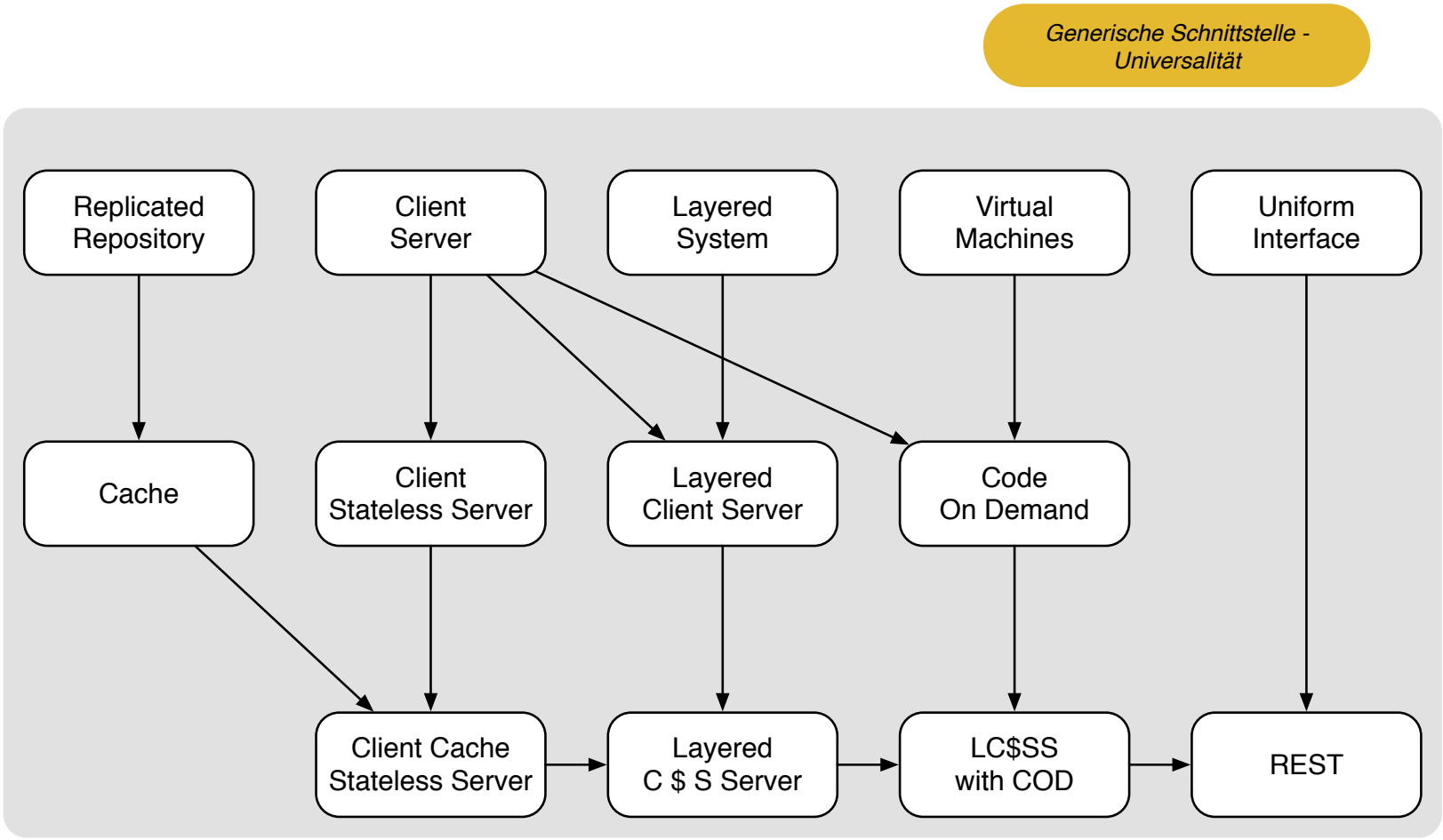
Herleitung von REST



Code-On-Demand

- Erweiterung der Funktionalität des Clients durch Code der vom Server geladen und auf dem Client ausgeführt wird: Interpreter-Stil
- Erweiterbarkeit des Clients, ohne dass sein Code selbst erweitert werden muss
- Performance: Client kann lokal Daten verarbeiten ohne Verwendung des Servers
- Aber: Client muss den Code ausführen können – komplexeres System
- Aber: Sicherheit – schadhafter Code kann eventuell auf dem Client ausgeführt werden

Herleitung von REST



Uniform Interface

„REST is defined by four interface constraints:

- identification of resources;
- manipulation of resourcen through representations;
- self-descriptive messages; and,
- hypermedia as the engine of application state.“

[Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures]

Literatur

Roy Thomas Fielding

Architectural Styles and the Design of Network-based Software Architectures

PhD Thesis Irvine, CA; 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

Roy T. Fielding, Richard N. Taylor

Principled Design of the Modern Web Architecture

ACM Transactions on Internet Technology, Vol. 2, No. 2, May 2002, Pages 115–150.

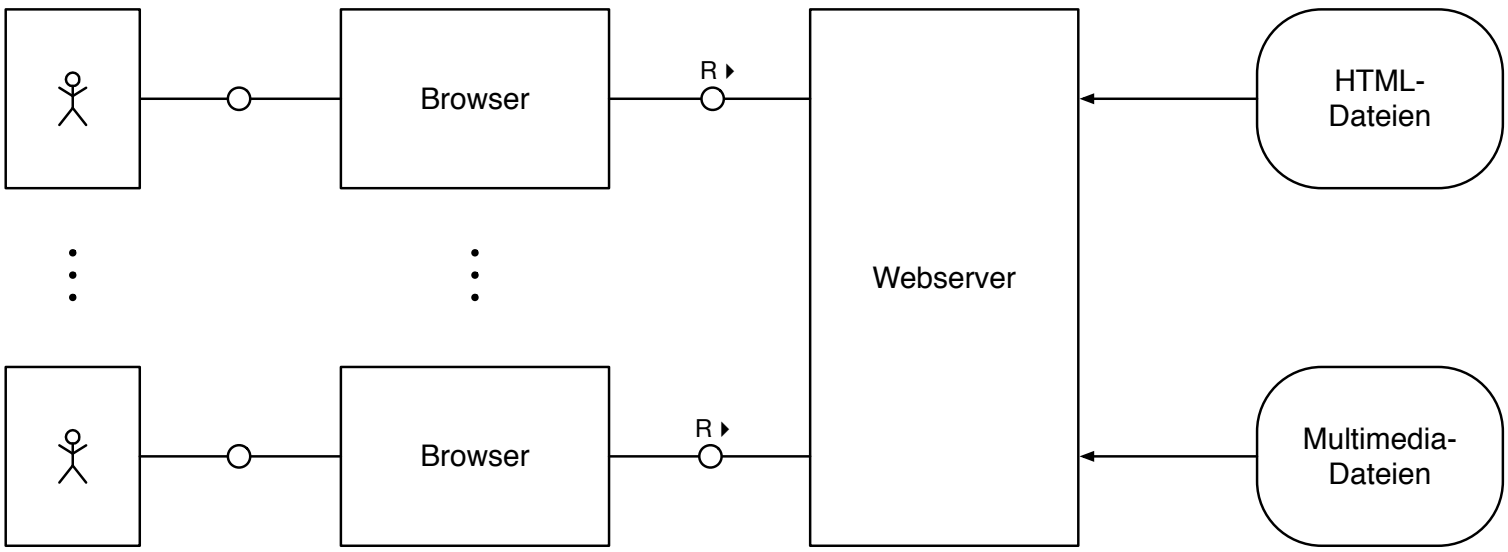
Übersicht

- Architekturstile und -muster
- Kombination von Stilen — am Beispiel der Architektur des Webs
- Beispiele von Architekturen
  - Entwicklung der Architektur von Web-Anwendungen
  - Metalevel-Architektur und domänenspezifische Sprachen
  - MapReduce
- Literatur & Links

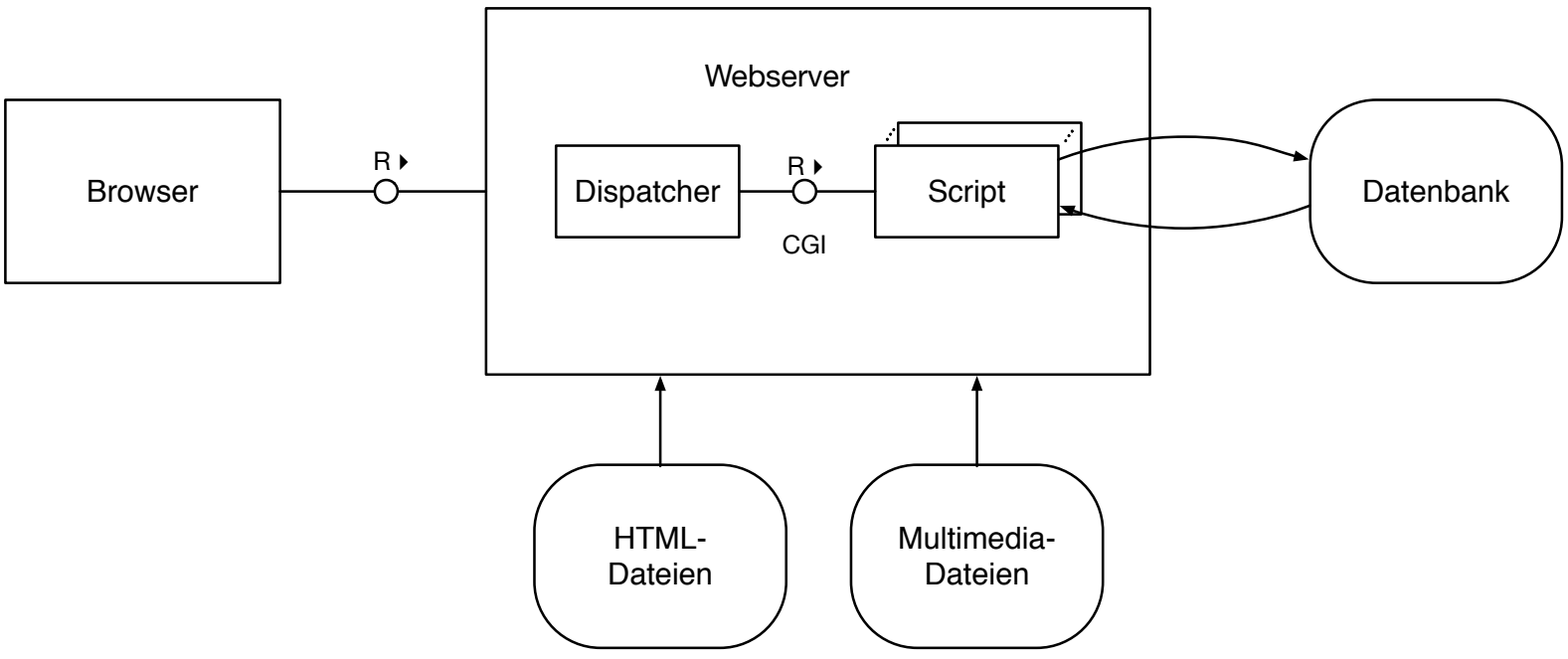
Beispiele von Architekturen

Entwicklung der Architektur von Web-Anwendungen

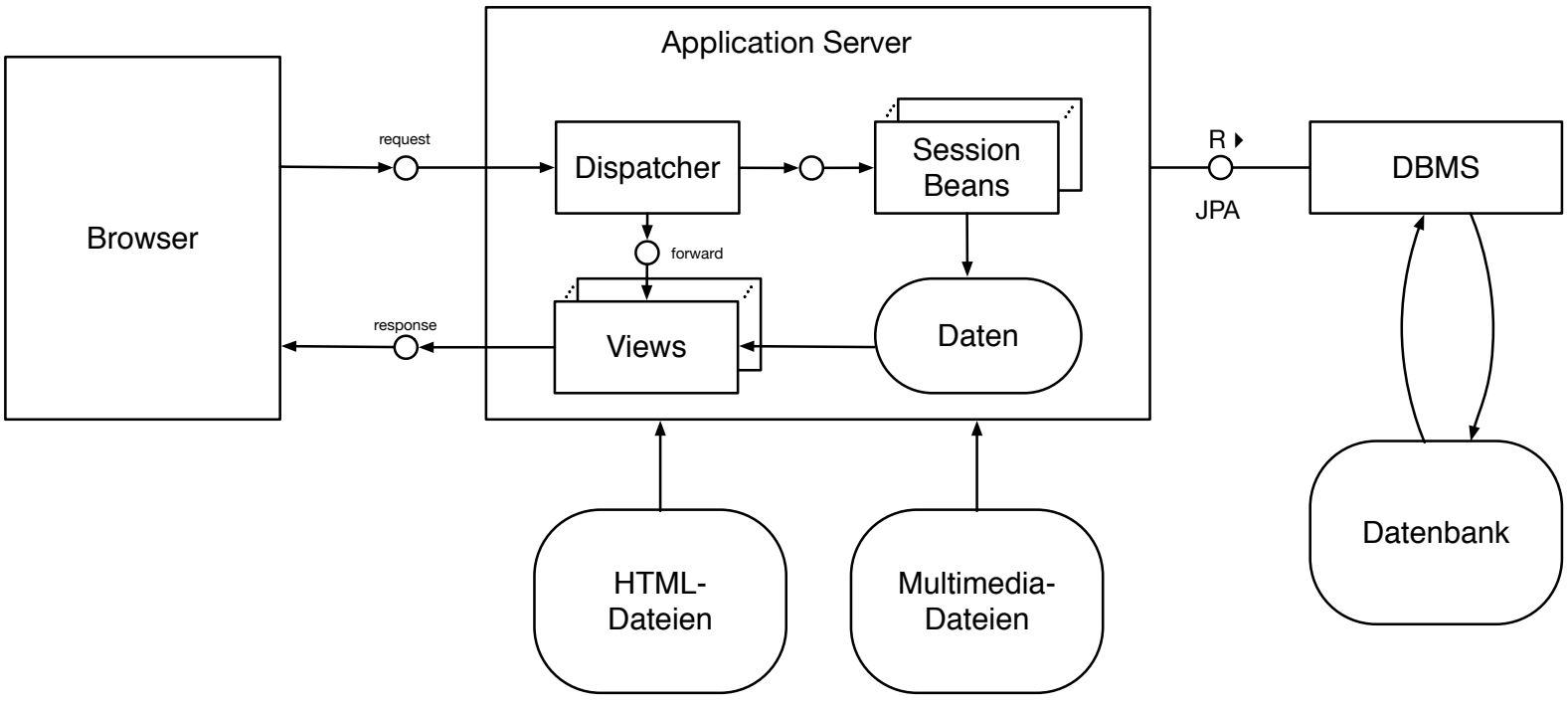
Statische, verlinkte Webseiten



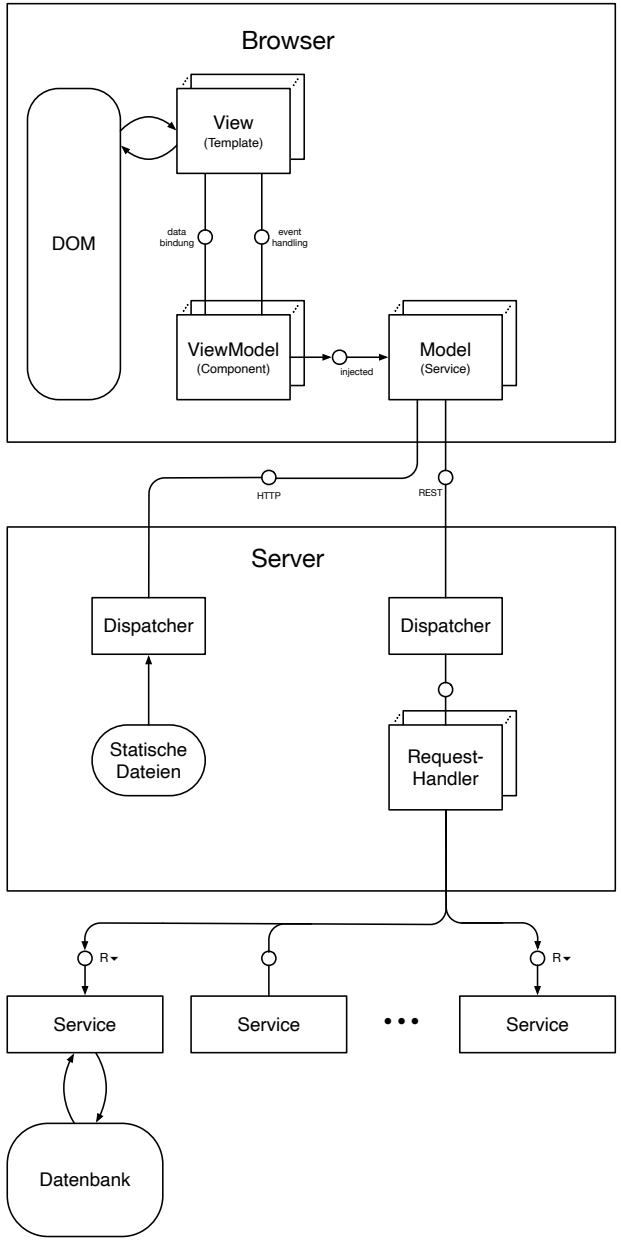
Verlinkte Webseiten mit Scripting (CGI)



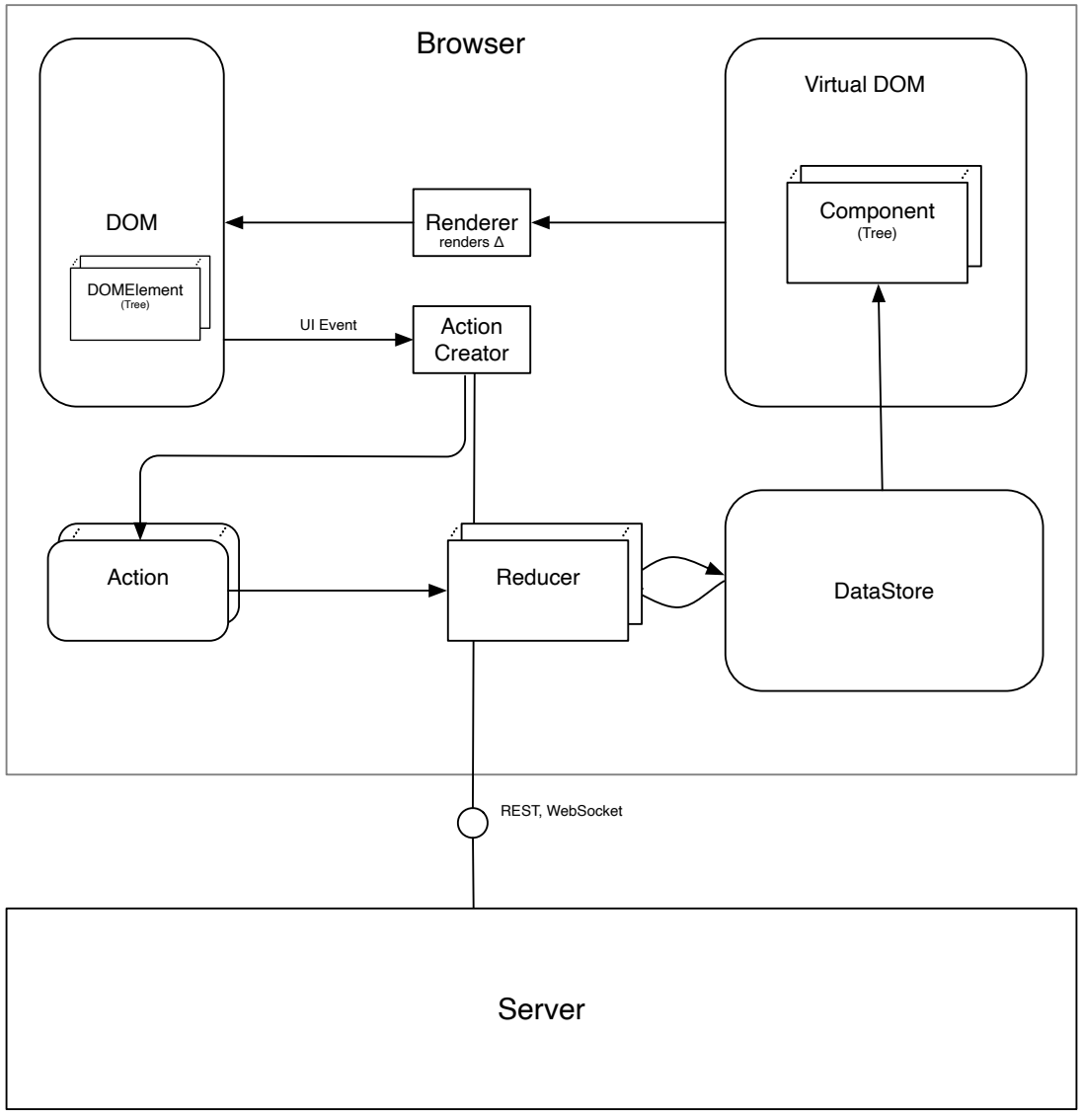
Geschäftsanwendungen im Web



Single-Page-Anwendungen im Web



Single-Page-Anwendungen mit zentralem Zustand



Beispiele von Architekturen

Metalevel-Architektur und domänenspezifische Sprachen

Metalevel-Architektur

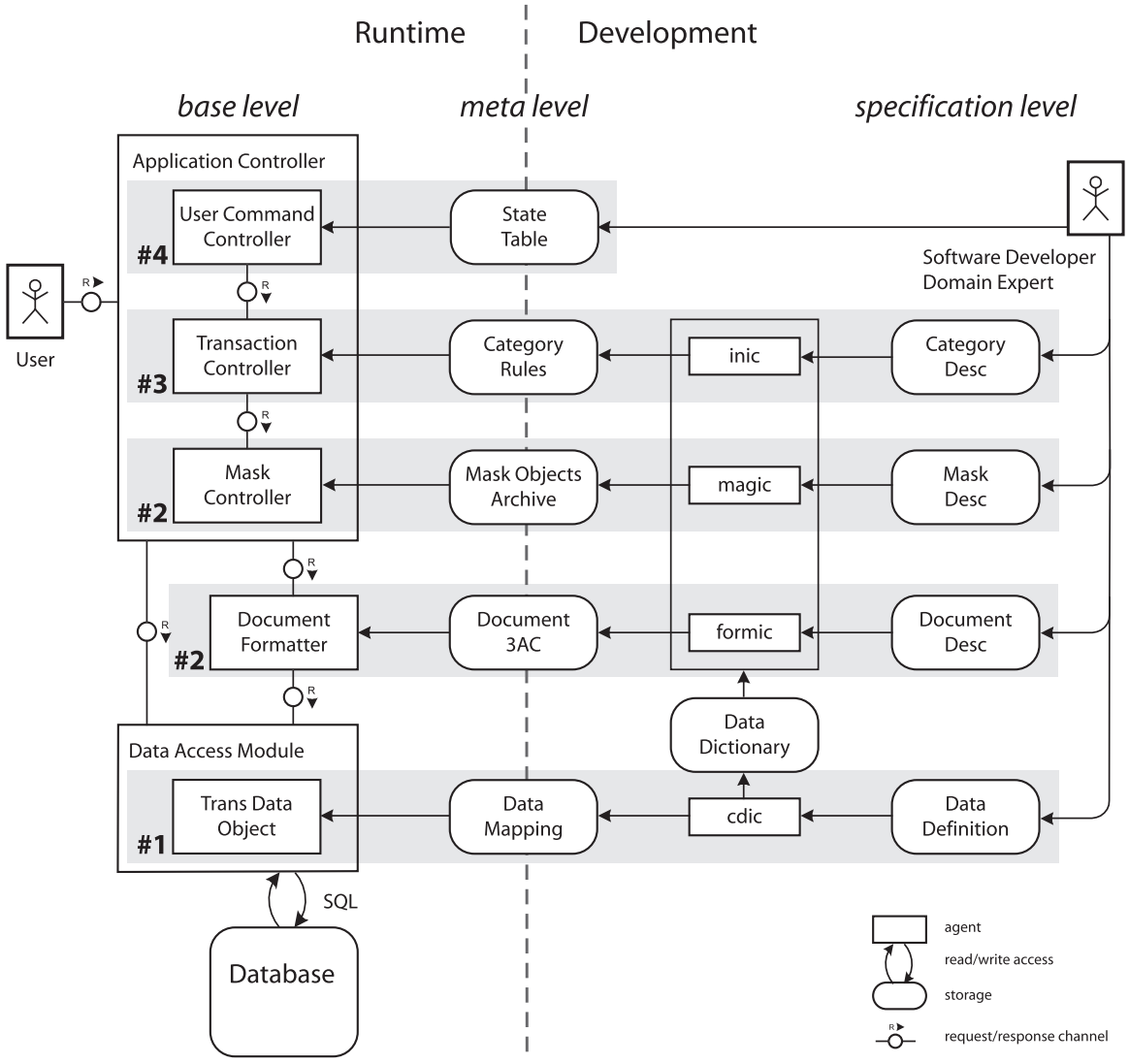
Metalevel und DSL

- Metaebene: beschreibt Verhalten der Anwendung
- Basisebene: tatsächliches Verhalten zur Laufzeit
- Domänenspezifische Sprache (DSL) beschreibt Metaobjekte – gerne mit fachlicher Expertise eingebaut

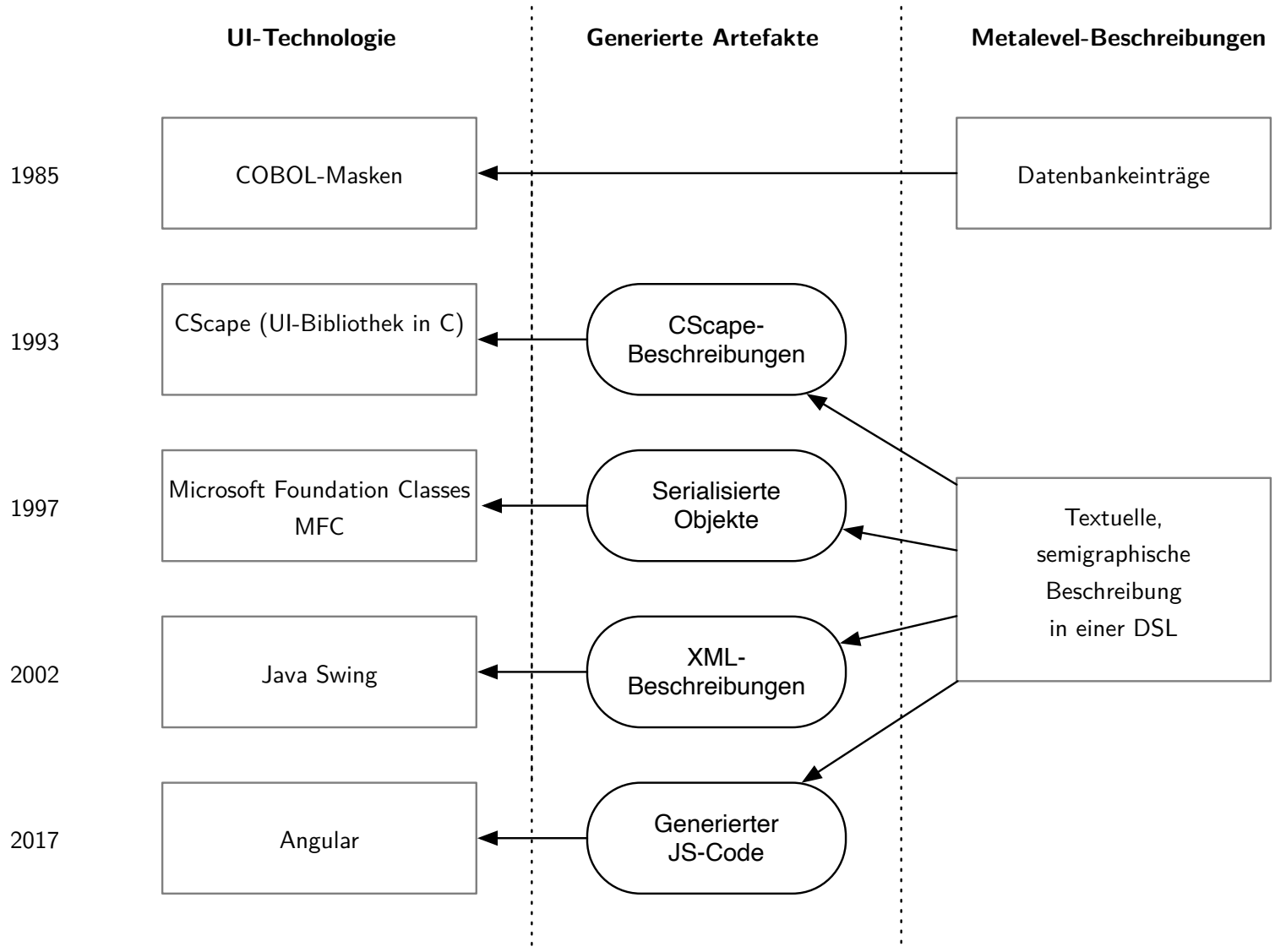
Verwendung der Artefakte der Metaebene zur ...

- ... Compilierzeit: Generierung von Code oder Laufzeitobjekten
- ... Initialisierungszeit: Konfiguration der Anwendung
- ... Laufzeit: Interpreter
- kombiniert: Zur Compilezeit wird Zwischencode erzeugt, der zur Laufzeit interpretiert wird

Metalevel-Architektur – Beispiel



Metalevel-Architektur – Evolution



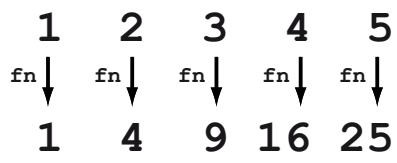
Beispiele von Architekturen

MapReduce

map und reduce in funktionalen Sprachen

```
(map fn coll)

(map square [1 2 3 4 5])
```

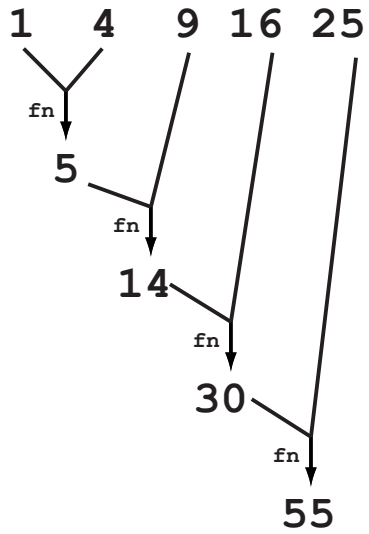


unäre Map-Funktion

map und reduce in funktionalen Sprachen

```
(reduce fn coll)

(reduce + [1 4 9 16 25])
```



binäre Reduce-Funktion

map und reduce in funktionalen Sprachen

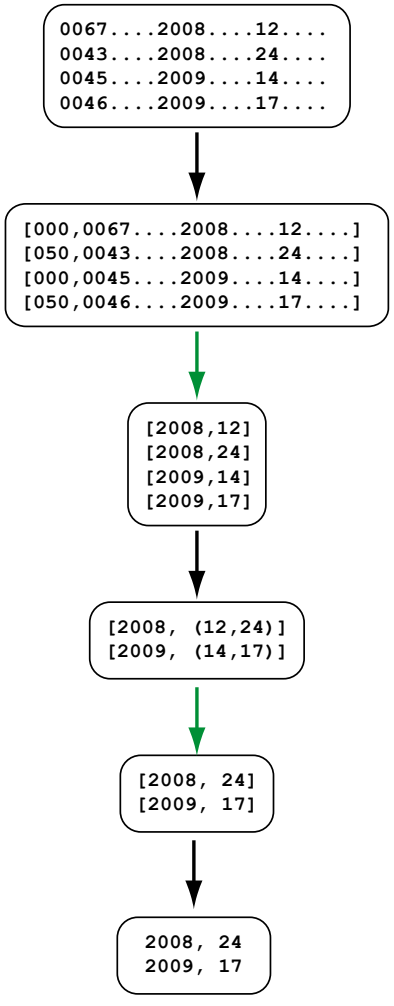
```
(mr map-fn reduce-fn coll)
```

unäre Map-Funktion

binäre Reduce-Funktion

```
(mr square + [1 2 3 4 5])
```

Grundidee von MapReduce



Datensätze in Rohform

record reader

[Key, Value]-Paare

map([k,v]) -> [k‘,v‘]
optional: combiner

[Key, Value]-Paare

partitioner
shuffle and sort

[Key, List]-Paare

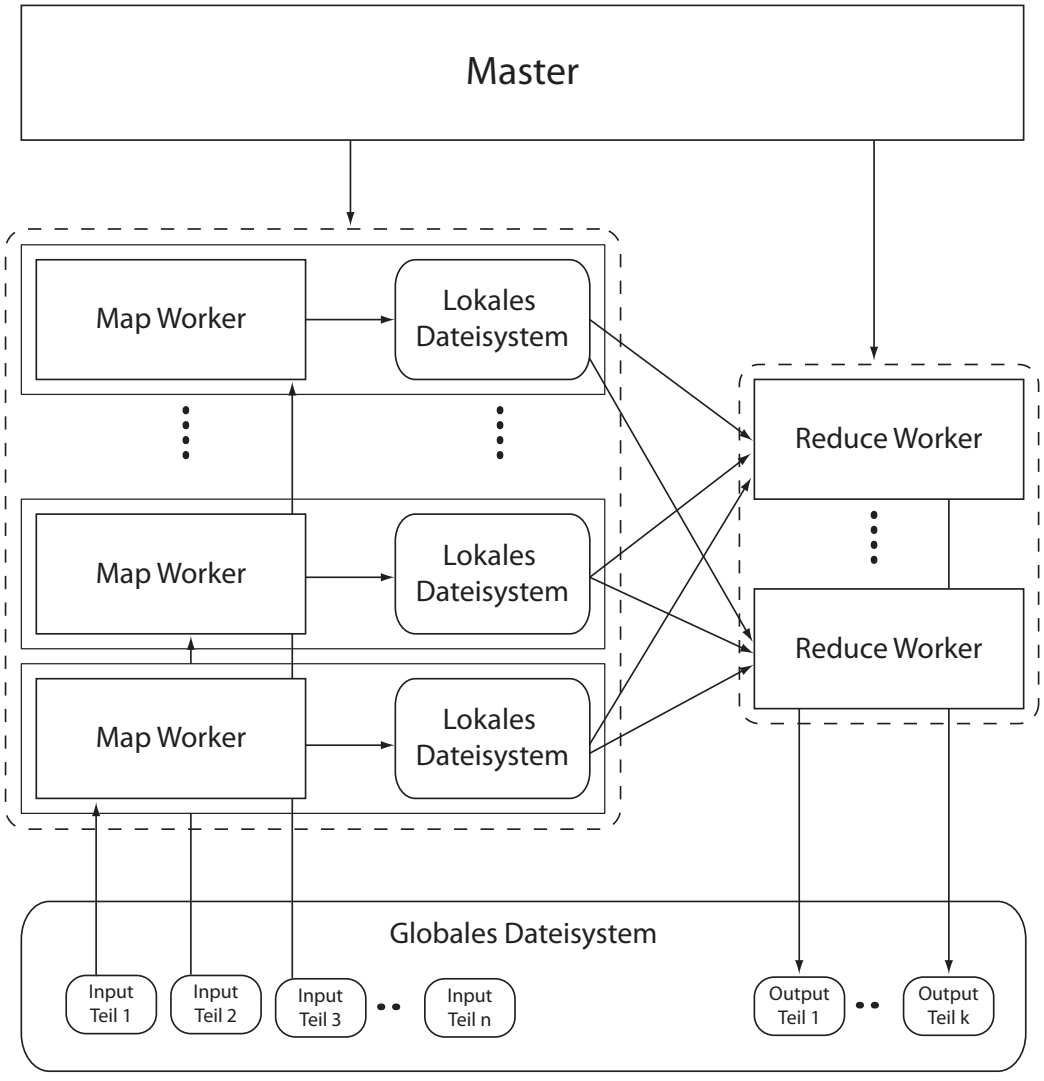
reduce([k‘,l]) -> [k‘,v“]

[Key, Value]-Paare

output format

Ergebnis

Architektur von MapReduce



Übersicht

Literatur

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal
Pattern-orientierte Software-Architektur: Ein Pattern-System
Bonn: Addison-Wesley, 1998.

Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann
Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects
J. Wiley & Sons, 2000.

Martin Fowler
Patterns of Enterprise Application Architecture
Addison-Wesley, 2003.

Literatur

Markus Voelter
DSL Engineering: Designing, Implementing and Using Domain-Specific Languages
dslbook.org, 2013. http://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf

Claudia Fritsch, Burkhardt Renz
Four Mechanisms for Adaptable Systems: A Meta-Level Approach to Building a Software Product Line.
Software Process Improvement and Practice Volume 10, 103 - 124, John Wiley & Sons 2005.
https://esb-dev.github.io/mat/4ma+.pdf

Literatur


Jeffrey Dean, Sanjay Ghemawat
MapReduce: Simplified Data Processing on Large Clusters
OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.


Tom White
Hadoop: The Definitive Guide
O'Reilly, 2012.


Donald Miner, Adam Shook
MapReduce Design Patterns
O'Reilly, 2013.