Architektur von Informationssystemen

Hochschule für angewandte Wissenschaften

Sommersemester 2016

Nils Löwe / nils@loewe.io / @NilsLoewe

Praktikumsaufgabe

2. Praktikumsaufgabe

Fragen?

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualiät und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Wiederholung
Architekturmuster

Was sind Architekturmuster?

Ein Architekturmuster beschreibt eine bewährte Lösung für ein wiederholt auftretendes Entwurfsproblem

(Effektive Softwarearchitekturen)

Warum Architekturmuster?

Erfolg kommt von Weisheit.

Weisheit kommt von Erfahrung.

Erfahrung kommt von Fehlern.

Aus Fehlern kann man hervorragend lernen.

Leider akzeptiert kaum ein Kunde Fehler, nur weil Sie Ihre Erfahrung als Software-Architekt sammeln.

In dieser Situation helfen Heuristiken.

Heuristiken kodifizieren Erfahrungen anderer Architekten und Projekte, auch aus anderen Bereichen der Systemarchitektur.

Heuristiken bieten Orientierung im Sinne von Wegweisern, Straßenmarkierungen und Warnschildern.

Sie geben allerdings lediglich Hinweise und garantieren nichts. Es bleibt in Ihrer Verantwortung, die passen- den Heuristiken für eine bestimmte Situation auszuwählen:

Die Kunst der Architektur liegt nicht in der Weisheit der Heuristiken, sondern in der Weisheit, a priori die passenden Heuristiken für das aktuelle Projekt auszuwählen.

Architektur: Von der Idee zur Struktur

Ein klassischer und systematischer Ansatz der Beherrschung von Komplexität lautet „teile und herrsche“ (divide et impera). Das Problem wird in immer kleinere Teile zerlegt, bis diese Teilprobleme eine überschaubare Größe annehmen.

Horizontale Zerlegung

Jede Schicht stellt einige klar definierte Schnittstellen zur Verfügung und nutzt Dienste von darunter liegenden Schichten.

Vertikale Zerlegung

Jeder Teil übernimmt eine bestimmte fachliche oder technische Funktion.

Prinzipien zur Zerlegung

Kapselung (information hiding)

Wiederverwendung

Iterativer Entwurf

Dokumentation von Entscheidungen

Unabhängigkeit der Elemente

Ein Praxisbericht

Why a service oriented architecture is not the holy grail...

1. Die falsche Zerlegung erhöht die Komplexität!
2. Microservices sind keine Lösung für organisatorische Probleme
3. Neue Probleme ersetzen alte Probleme, z.B. Abhängigkeiten im Deployment

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualiät und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Überblick über Architekturmuster

Arten von Architekturmustern?

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

Mikrokernel

Reflexion

Dependency Injection

Anti-Patterns

Anti-Patterns?

Ein Anti-Pattern ist in der Softwareentwicklung ein häufig anzutreffender schlechter Lösungsansatz für ein bestimmtes Problem. Es bildet damit das Gegenstück zu den Mustern (Entwurfsmuster, Analysemuster, Architekturmuster, …), welche allgemein übliche und bewährte Problemlösungsansätze beschreiben.

Überblick über Antipatterns

Projektmanagement Anti-Patterns

Architektur Anti-Pattern

Code Smells

Organisations Anti-Pattern

Projektmanagement Anti-Patterns

Blendwerk

Aufgeblähte Software

Feature creep

Scope creep

Brooks'sches Gesetz

Death Sprint

Death March

Architektur Anti-Patterns

Big Ball of Mud

Gasfabrik

Gottobjekt

Innere-Plattform-Effekt

Spaghetticode

Sumo-Hochzeit

Code Smells

Zwiebel

Copy and Paste

Lavafluss

Magische Werte

Reservierte Wörter

Unbeabsichtigte Komplexität

Organisations Anti-Pattern

Wunderwaffe

Das Rad neu erfinden

Das quadratische Rad neu erfinden

Body ballooning

Empire building

Warme Leiche

Single head of knowledge

Organisations Anti-Pattern II

Management nach Zahlen

Vendor Lock-In

Design by Committee

Boat Anchor

Dead End

Swiss Army Knife

Blendwerk

Das Blendwerk (englisch *Smoke and mirrors*) bezeichnet nicht fertige Funktionen, welche als fertig vorgetäuscht werden.

Aufgeblähte Software

Als Bloatware (englisch *„aufblähen“*) wird Software bezeichnet, die mit Funktionen überladen ist bzw. die Anwendungen sehr unterschiedlicher Arbeitsfelder ohne gemeinsamen Nutzen bündelt. Für den Anwender macht dies das Programm unübersichtlich, für Entwickler unwartbar.

Feature creep

bezeichnet es, wenn der Umfang der zu entwickelnden Funktionalität in einem Projektplan festgehalten wird, diese aber dauernd erweitert wird.

Der Kunde versucht, nach der Erstellung des Projektplanes weitere Funktionalität mit unterzubringen. Dies führt zu Problemen, wenn die in Arbeit befindliche Version nicht das notwendige Design aufweist, Termine nicht eingehalten werden können oder die realen Kosten über die planmäßigen Kosten wachsen.

Scope creep

ist ähnlich wie der Feature creep, jedoch nicht auf Funktionalität bezogen, sondern auf den Anwendungsbereich. Auch hier zeichnet sich der Auftraggeber dadurch aus, dass er geschickt und versteckt den Umfang der Software nachträglich erweitern möchte, ohne dass er dies explizit zugibt.

Death Sprint

Bei einem Death Sprint (Überhitzter Projektplan) wird Software iterativ bereitgestellt. Die Bereitstellung erfolgt hierbei in einer viel zu kurzen Zeitspanne. Nach außen sieht das Projekt zunächst sehr erfolgreich aus, da immer wieder neue Versionen mit neuen Eigenschaften abgeschlossen werden. Allerdings leidet die Qualität des Produktes sowohl nach außen sichtbar, wie auch technisch, was allerdings nur der Entwickler erkennt. Die Qualität nimmt ab mit jeder „erfolgreichen“ neuen Iteration.

Death March

Ein Death March (Todesmarsch; gelegentlich auch Himmelfahrtskommando) ist das Gegenteil von einem Überhitzten Projektplan. Ein Todesmarschprojekt zieht sich ewig hin.

Ein Todesmarschprojekt kann auch bewusst in Kauf genommen werden, um von Defiziten in der Organisation abzulenken und Entwicklungen zu verschleppen, d. h. so lange an etwas zu entwickeln, bis eine nicht genau spezifizierte Eigenschaft in irgendeiner Form subjektiv funktioniert.

Brooks'sches Gesetz

"Adding manpower to a late software project makes it later."

Big Ball of Mud

bezeichnet ein Programm, das keine erkennbare Softwarearchitektur besitzt. Die häufigsten Ursachen dafür sind ungenügende Erfahrung, fehlendes Bewusstsein für Softwarearchitektur, Fluktuation der Mitarbeiter sowie Druck auf die Umsetzungsmannschaft. Obwohl derartige Systeme aus Wartbarkeitsgründen unerwünscht sind, sind sie dennoch häufig anzutreffen.

Gasfabrik

Als Gasfabrik (englisch *Gas factory*) werden unnötig komplexe Systementwürfe für relativ simple Probleme bezeichnet.

The Blob / Gottobjekt

Ein Objekt (“Blob“) enthält den Großteil der Verantwortlichkeiten, während die meisten anderen Objekte nur elementare Daten speichern oder elementare Dienste anbieten.

Innere-Plattform-Effekt

tritt auf wenn ein System derartig weitreichende Konfigurationsmöglichkeiten besitzt, dass es letztlich zu einer schwachen Kopie der Plattform wird, mittels derer es gebaut wurde.

Ein Beispiel sind Datenmodelle, die auf konkrete (anwendungsbezogene) Datenbanktabellen verzichten und stattdessen mittels allgemeiner Tabellen eine eigene Verwaltungsschicht für die Datenstruktur implementieren mit dem eigentlichen Ziel, die Flexibilität zu erhöhen.

Spaghetti Code

Der Code ist weitgehend unstrukturiert; keine Objektorientierung oder Modularisierung: undurchsichtiger Kontrollfluss.

Sumo-Hochzeit

Als Sumo-Hochzeit bezeichnet man es, wenn ein Fat Client unnatürlich stark abhängig von der Datenbank ist.

In der Datenbank ist hierbei sehr viel Logik in Form der datenbankeigenen Programmiersprache positioniert. Beispielsweise in Oracle mit der Programmiersprache PL/SQL. Die ganze Architektur ist dadurch sehr unflexibel.

Zwiebel

Als Zwiebel (engl. Onion) bezeichnet man Programmcode, bei dem neue Funktionalität um (oder über) die alte gelegt wird.

Häufig entstehen Zwiebeln, wenn ein Entwickler ein Programm erweitern soll, das er nicht geschrieben hat. Der Entwickler möchte oder kann die bereits existente Lösung nicht komplett verstehen und setzt seine neue Lösung einfach drüber. Dies führt mit einer Vielzahl von Versionen und unterschiedlichen Entwicklern über die Jahre zu einem Zwiebel-System.

Cut-and-Paste Programming

Code wird an zahlreichen Stellen wiederverwendet, indem er kopiert und verändert wird. Dies sorgt für Wartungsprobleme

Lösung: Black-Box-Wiederverwendung, Refaktorisierung

Lava Flow

Ein Lavafluss (englisch Lava flow oder Dead Code) beschreibt den Umstand, dass in einer Anwendung immer mehr „toter Quelltext“ herumliegt. Dieser wird nicht mehr genutzt. Statt ihn zu löschen, werden im Programm immer mehr Verzweigungen eingebaut, die um den besagten Quelltext herumlaufen oder auf ihm aufbauen. Redundanter Code ist der Überbegriff zu totem Code.

The Golden Hammer / Wunderwaffe

Ein bekanntes Verfahren (Golden ”Hammer“) wird auf alle moglichen Probleme angewandt Wer als einziges Werkzeug nur einen Hammer kennt, lebt in einer Welt voller Nägel.

Lösung: Ausbildung verbessern

Reinvent the Wheel

Da es an Wissen uber vorhandene Produkte und Lösungen (auch innerhalb der Firma) fehlt,wird das Rad stets neu erfunden. Erhöhte Entwicklungskosten und Terminprobleme.

Lösung: Wissensmanagement verbessern

Das quadratische Rad neu erfinden

Mit das quadratische Rad neu erfinden (englisch Reinventing the square wheel) bezeichnet man die Bereitstellung einer schlechten Lösung, wenn eine gute Lösung bereits existiert.

Body ballooning

Beim Body ballooning handelt der Vorgesetzte ausschließlich aus der Bestrebung heraus, seine Machtposition auszubauen, welche sich entweder aus der Unternehmensstruktur oder auch rein subjektiv aus der Anzahl der Mitarbeiter unter sich definiert. Dies kann dazu führen, dass der Vorgesetzte bewusst arbeitsintensivere Lösungen und Arbeitstechniken den effizienten vorzieht.

Empire building

Durch sachlich nicht nachvollziehbare, nicht konstruktive Maßnahmen versucht ein einzelner, seine Macht auszubauen bzw. zu erhalten. Dies kann Body ballooning sein, aber auch das ständige Beschuldigen anderer, gerade derer, die nicht mehr für die Unternehmung arbeiten, die Ausführung von pathologischer Politik, Diskreditierung, Mobbing und sonstige Facetten, die nur darauf abzielen, die eigene Position zu stärken bzw. den eigenen Status zu halten.

Warme Leiche

Eine warme Leiche (englisch warm body) bezeichnet eine Person, die einen zweifelhaften oder keinen Beitrag zu einem Projekt leistet.

Single head of knowledge

Ein Single head of knowledge ist ein Individuum, welches zu einer Software, einem Werkzeug oder einem anderen eingesetzten Medium, als einziges unternehmensweit das Wissen besitzt. Dies zeugt häufig von fehlendem Wissensmanagement, mangelndem Austausch zwischen den Kollegen oder Defiziten in der Organisation, kann aber auch von dem Individuum bewusst angestrebt worden sein.

Management nach Zahlen

(englisch *Management by numbers*) ist eine Anspielung auf Malen nach Zahlen. Beim Management nach Zahlen wird ein übermäßiger Schwerpunkt auf das quantitative Management gelegt. Insbesondere wenn Fokus auf Kosten gelegt wird, während andere Faktoren wie Qualität vernachlässigt werden.

Vendor Lock-In

Ein System ist weitgehend abhangig von einer proprietaren Architektur oder proprietaren Datenformaten

Lösung: Portabilität erhöhen

Lösung: Abstraktionen einführen

Design by Committee

Das typische Anti-Muster von Standardisierungsgremien, die dazu neigen, es jedem Teilnehmer recht zu machen und übermäßig komplexe Entwürfe abzuliefern

Lösung: Gruppendynamik und Treffen verbessern

Boat Anchor

Eine Komponente ohne erkennbaren Nutzen

Dead End

eingekaufte Komponente, die nicht mehr unterstützt wird

Swiss Army Knife

Eine Komponente, die vorgibt, alles tun zu können

Hilfreiches Wissen um Anti-Patterns vorzubeugen

”The Pragmatic Programmer”

(Andy Hunt, Dave Thomas)

”Clean Code”

(Uncle Bob Martin)

”The Developers Code”

(Ka Wai Cheung)

Warum sind wir nochmal hier?

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

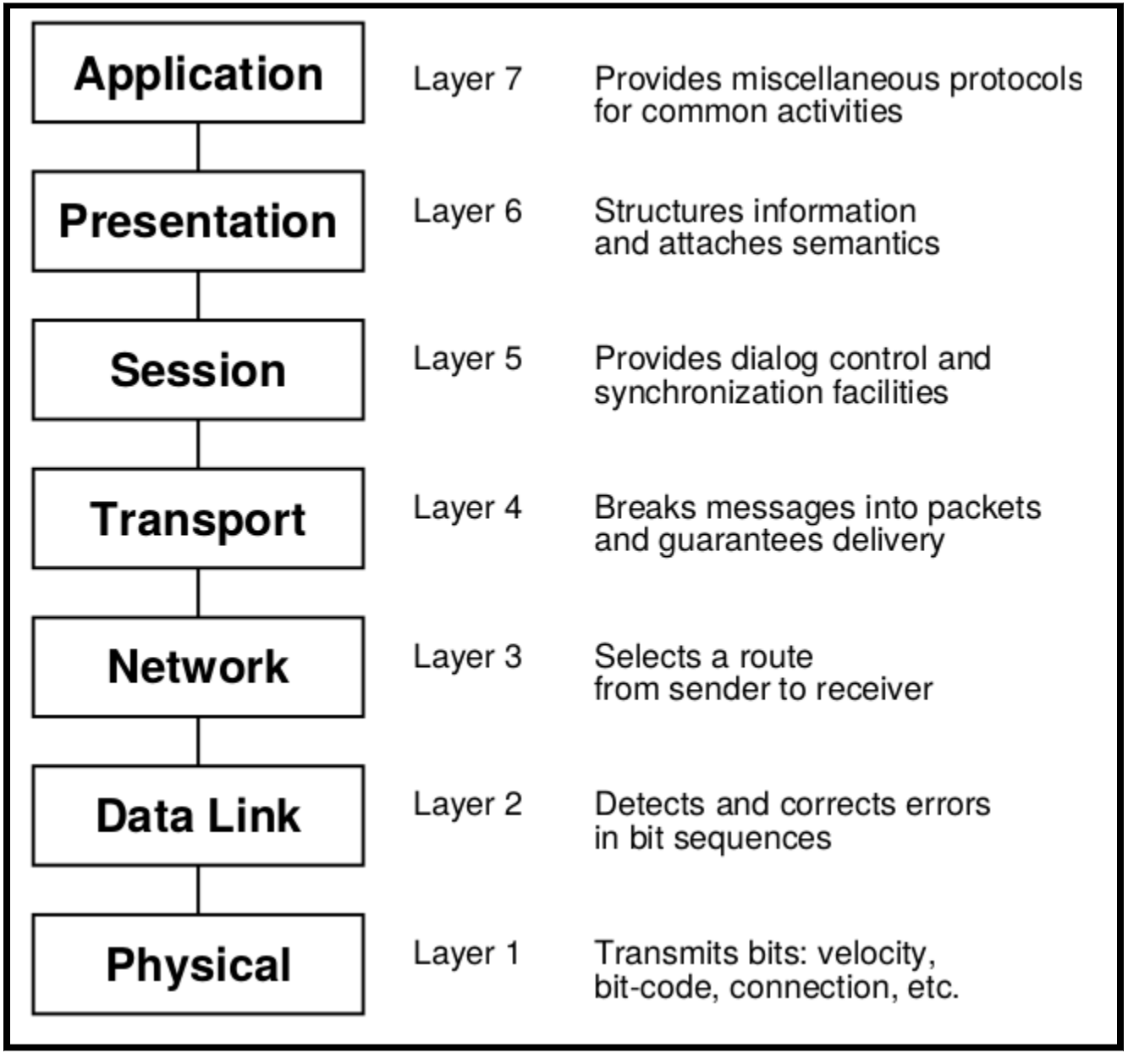
Layers

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

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:

Beispiel: ISO/OSI-Referenzmodell



Layers

Aufgabe: Folgendes System bauen:

- Aktivitäten auf niederer Ebene wie Hardware-Ansteuerung, Sensoren, Bitverarbeitung
- Aktivitäten auf hoher Ebene wie Planung, Strategien und Anwenderfunktionalität
- Die Aktivitäten auf hoher Ebene werden durch Aktivitäten der niederen Ebenen realisiert

Layers

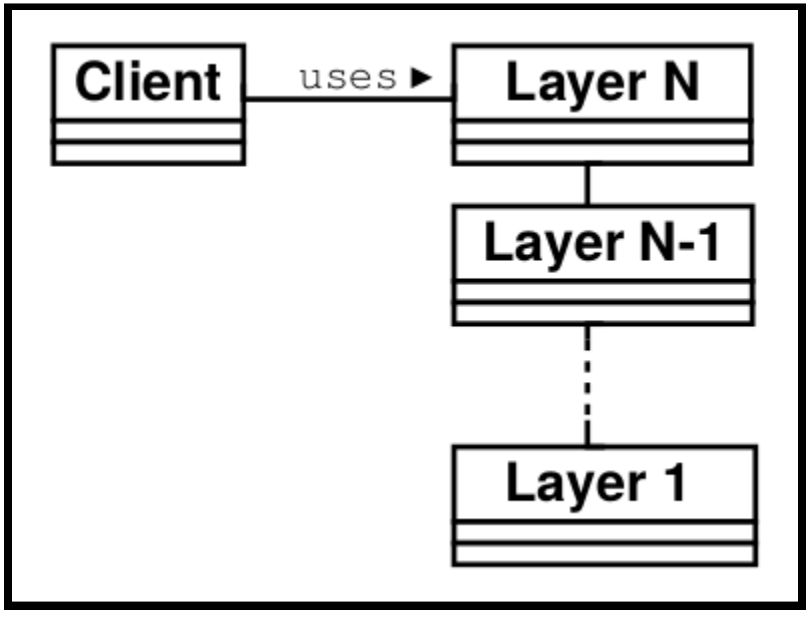
Dabei sollen folgende Ziele berücksichtigt werden:

- Änderungen am Quellcode sollten möglichst wenige Ebenen betreffen
- Schnittstellen sollten stabil (und möglicherweise standardisiert) sein
- Teile (= Ebenen) sollten austauschbar sein
- Jede Ebene soll separat realisierbar sein

Layers

Das Layers-Muster gliedert ein System in zahlreiche Schichten. Jede Schicht schützt die unteren Schichten vor direktem Zugriff durch höhere Schichten.

Layers



Layers

Dynamisches Verhalten

Top-Down Anforderung

Eine Anforderung des Benutzers wird von der obersten Schicht entgegengenommen. Diese resultiert in Anforderungen der unteren Schichten bis hinunter auf die unterste Ebene. Ggf. werden die Ergebnisse der unteren Schichten wieder nach oben weitergeleitet, bis das letzte Ergebnis an den Benutzer zurückgegeben wird.

Layers

Dynamisches Verhalten

Bottom-Up Anforderung

Hier empfängt die unterste Schicht ein Signal, das an die oberen Schichten weitergeleitet wird. Schließlich benachrichtigt die oberste Schicht den Benutzer.

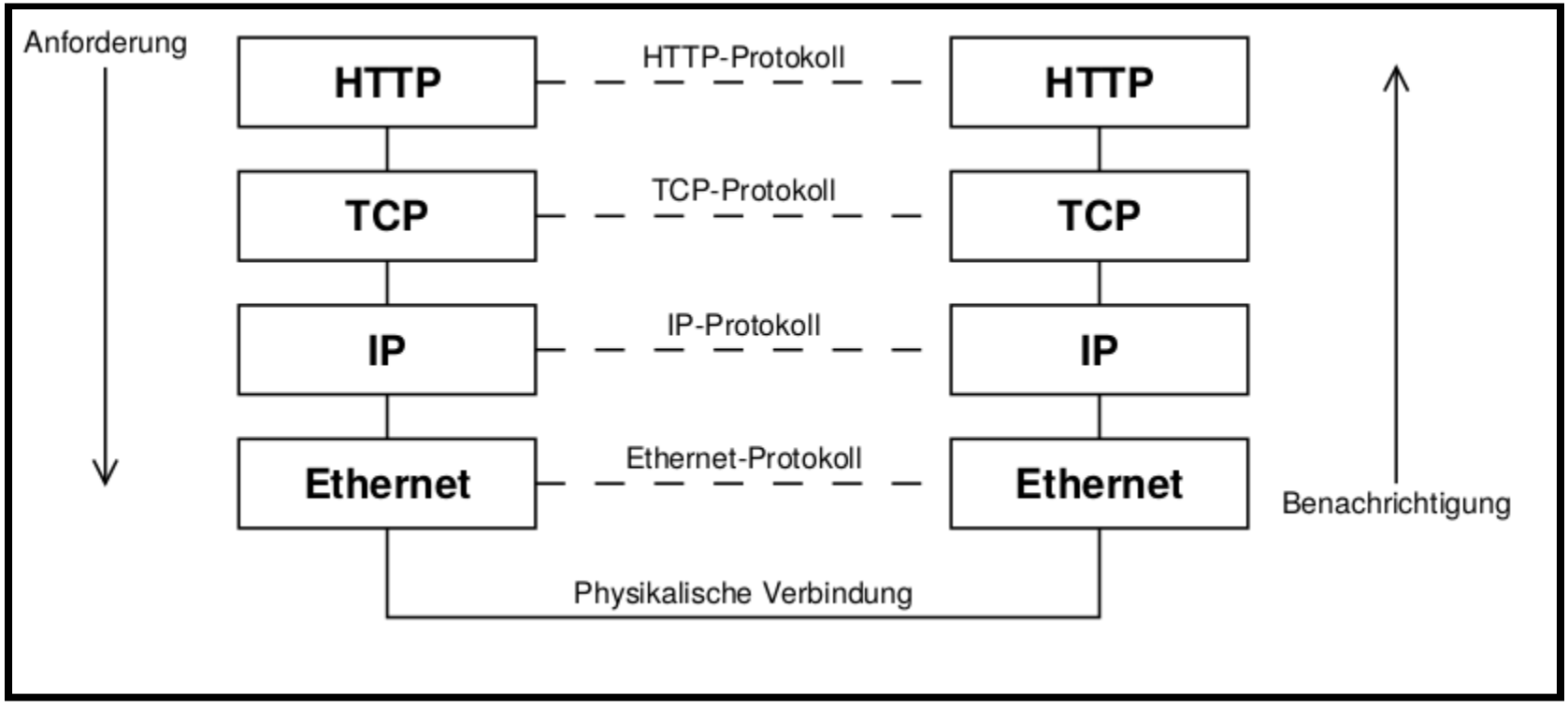
Layers

Dynamisches Verhalten: Protokoll Stack

In diesem Szenario kommunizieren zwei n-Schichten-Stacks miteinander. Eine Anforderung wandert durch den ersten Stack hinunter, wird übertragen und schließlich als Signal vom zweiten Stack empfangen. Jede Schicht verwaltet dabei ihr eigenes Protokoll.

Layers

Beispiel: TCP/IP



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

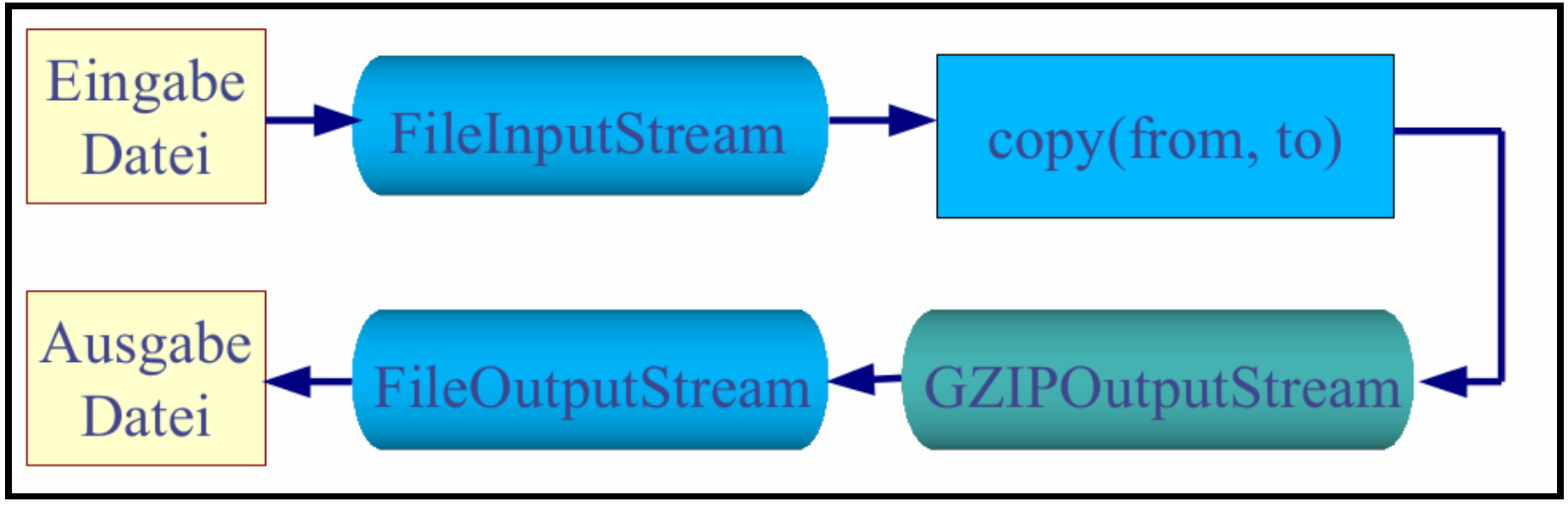
Blackboard

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

Die fünf größten Dateien ausgeben

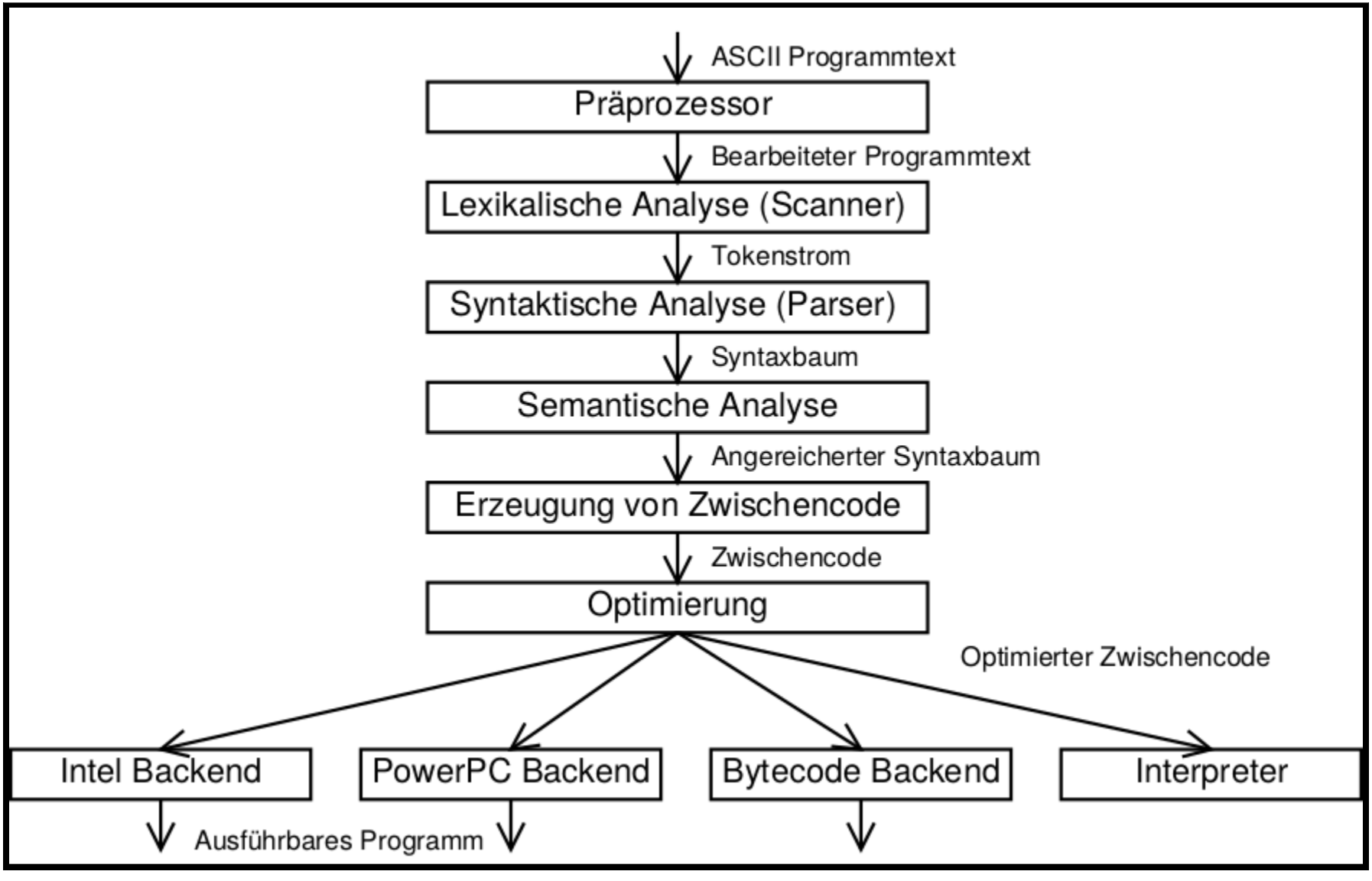
```
du | sort -rn | head -5
```

Pipes and Filters

Alle Prozesse stoppen, die mit p beginnen

```
ps - e | grep " p" | awk '{ print $1 }' | xargs kill
```

Beispiel: Compiler



Pipes and Filters

Teilnehmer: Filter

zuständig für Zusammenarbeit mit Pipe

- Holt Eingabedaten.
- Wendet eine Funktion auf seine Eingabedaten an.
- Liefert Ausgabedaten.

Pipes and Filters

Teilnehmer: Filter

Ein Filter kann auf dreierlei Weise mit den Daten umgehen:

- Er kann die Daten anreichern, indem er weitere Informationen berechnet und hinzufügt,
- Er kann die Daten verfeinern, indem er Information konzentriert oder extrahiert
- Er kann die Daten verändern, indem er sie in eine andere Darstellung überführt.

Pipes and Filters

Teilnehmer: Filter

Ein Filter kann auf verschiedene Weise aktiv werden:

- Die folgende Pipeline holt Daten aus dem Filter
- Die vorhergehende Pipeline schickt Daten in den Filter
- Meistens ist der Filter jedoch selbst aktiv – er holt Daten aus der vorhergehenden Pipeline und schickt Daten in die folgende Pipeline.

Pipes and Filters

Teilnehmer: Pipe

Eine Pipe verbindet Filter miteinander. Sie verbindet auch die Datenquelle mit dem ersten Filter und den letzten Filter mit der Datensenke.

Zuständig für:

- Übermittelt Daten
- Puffert Daten
- Synchronisiert aktive Nachbarn

Verbindet eine Pipe zwei aktive Komponenten, sichert sie die Synchronisation der Filter.

Pipes and Filters

Teilnehmer: Datenquelle, Datensenke

Diese Komponenten sind die Endstücke der Pipeline und somit die Verbindung zur Aussenwelt.

Zuständig für:

- Übermittelt Daten an/aus Pipeline.

Eine Datenquelle kann entweder aktiv sein (dann reicht sie von sich aus Daten in die Pipeline) oder passiv (dann wartet sie, bis der nächste Filter Daten anfordert). Analog kann die Datensenke aktiv Daten anfordern oder passiv auf Daten warten.

Pipes and Filters

Dynamisches Verhalten

Zwei aktive Filter sind durch eine Pipe verbunden; beide Filter arbeiten parallel.

Pipes and Filters

Vorteile

- Flexibilität durch Austausch und Hinzufügen von Filtern
- Flexibilität durch Neuanordnung
- 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

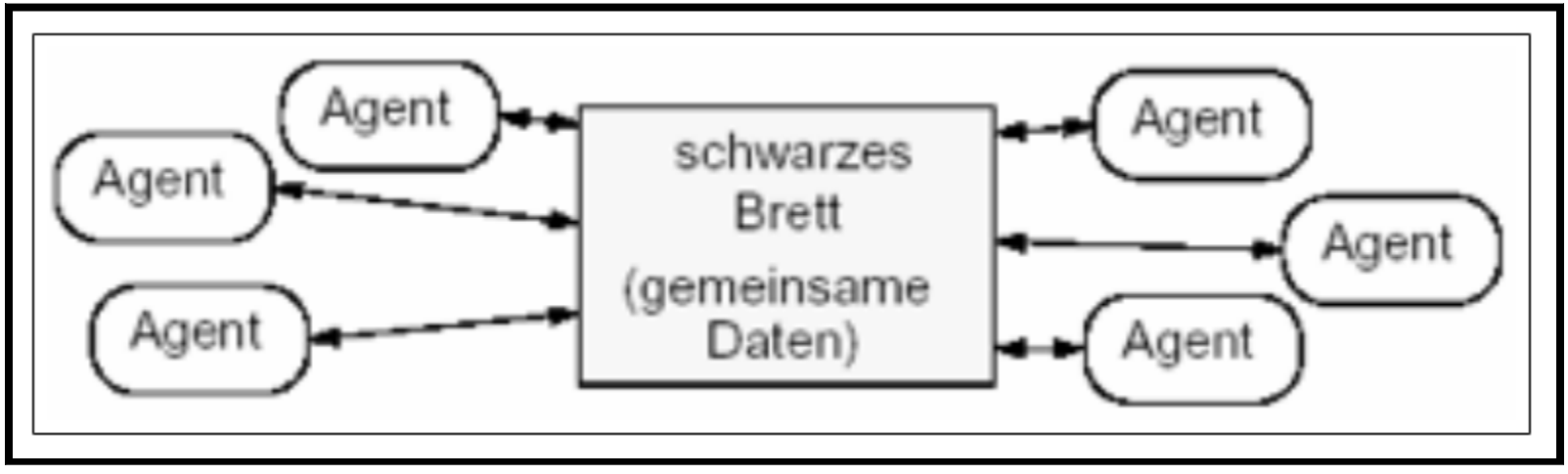
Blackboard

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


Agent
Agent
Agent
schwarzes
Brett
(gemeinsame
Daten)
Agent
Agent
Agent

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 betroen 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.

Domain-driven Design

Sprache

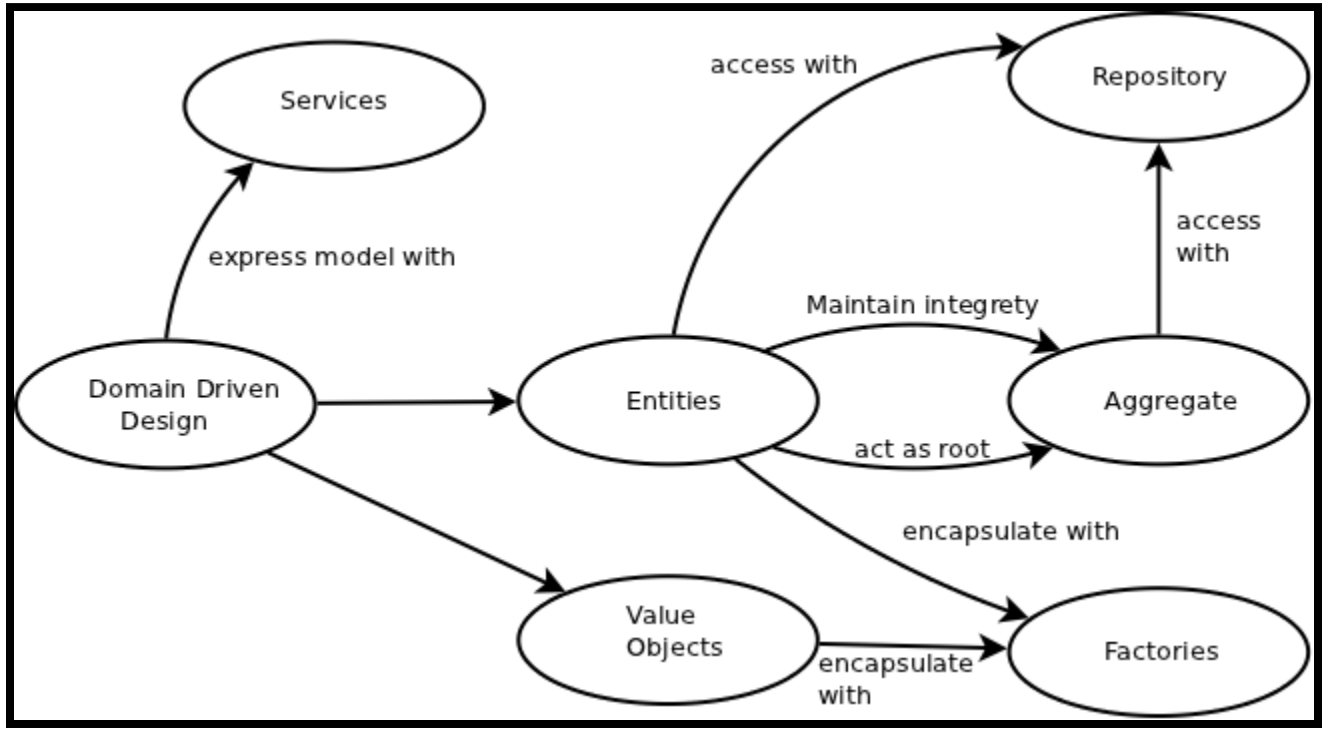
Eine Sprache für die Beschreibung der Fachlichkeit, der Elemente des Domänenmodells, der Klassen und Methoden etc. Sie wird definiert als:

“A language structured around the domain model and used by all team members to connect all the activities of the team with the software.”

Domain-driven Design

Bestandteile des Domänenmodells

Domain-driven Design unterscheidet die folgenden Bestandteile des Domänenmodells:



Domain-driven Design

Entitäten *(Entities, reference objects)*

Objekte des Modelles, welche nicht durch ihre Eigenschaften, sondern durch ihre Identität definiert werden.

Beispiel: ”Person”

Domain-driven Design

Wertobjekte *(value objects)*

Objekte des Modelles, welche keine konzeptionelle Identität haben oder benötigen und somit allein durch ihre Eigenschaften definiert werden.

Wertobjekte werden üblicherweise als unveränderliche Objekte (immutable objects) modelliert, damit sind sie wiederverwendbar und verteilbar.

Beispiel: ”Konfiguration”

Domain-driven Design

Aggregate *(aggregates)*

Aggregate sind Zusammenfassungen von Entitäten und Wertobjekten und deren Assoziationen untereinander zu einer gemeinsamen transaktionalen Einheit.

Aggregate definieren genau eine Entität als einzigen Zugriff auf das gesamte Aggregat. Alle anderen Entitäten und Wertobjekte dürfen von außerhalb nicht statisch referenziert werden. Damit wird garantiert, dass alle Invarianten des Aggregats und der einzelnen Bestandteile des Aggregats sichergestellt werden können.

Beispiel: ”Datenbanktransaktion”

Domain-driven Design

Assoziationen *(associations)*

Assoziationen sind, wie bei UML definiert, Beziehungen zwischen zwei oder mehr Objekten des Domänenmodells.

Hier werden nicht nur statische, durch Referenzen definierte Beziehungen betrachtet, sondern auch dynamische Beziehungen, die beispielsweise erst durch die Abarbeitung von SQL-Queries entstehen.

Domain-driven Design

Serviceobjekte *(services)*

Bei Domain-driven Design werden Funktionalitäten, welche ein wichtiges Konzept der Fachlichkeit darstellen und konzeptionell zu mehreren Objekten des Domänenmodells gehören, als eigenständige Serviceobjekte modelliert.

Serviceobjekte sind üblicherweise zustandslose (eng. stateless) und daher wiederverwendbare Klassen ohne Assoziationen, mit Methoden, die den angebotenen Funktionalitäten entsprechen. Diese Methoden bekommen die Wertobjekte und Entitäten übergeben, die zur Abarbeitung der Funktionalität notwendig sind.

Beispiel: ”API Wrapper”

Domain-driven Design

Fachliche Ereignisse *(domain events)*

Fachliche Ereignisse sind Objekte, welche komplexe, sich unter Umständen dynamisch ändernde, Aktionen des Domänenmodells beschreiben, die ein oder mehrere Aktionen oder Änderungen in den Fachobjekten bewirken.

Fachliche Ereignisse ermöglichen auch die Modellierung verteilter Systeme. Die einzelnen Subsysteme kommunizieren ausschließlich über fachliche Ereignisse, damit werden sie stark entkoppelt und das gesamte System somit wartbarer und skalierbarer.

Domain-driven Design

Module *(modules, packages)*

Module teilen das Domänenmodell in fachliche (nicht technische) Bestandteile. Sie sind gekennzeichnet durch starke innere Kohäsion und geringe Kopplung zwischen den Modulen.

Domain-driven Design

Fabriken *(factories)*

Fabriken dienen dazu, die Erzeugung von Fachobjekten in spezielle Fabrik-Objekte auszulagern.

Dies ist sinnvoll, wenn entweder die Erzeugung komplex ist (und beispielsweise Assoziationen benötigt, die das Fachobjekt selbst nicht mehr benötigt) oder die spezifische Erzeugung der Fachobjekte zur Laufzeit ausgetauscht werden können soll.

Fabriken werden üblicherweise durch erzeugende Entwurfsmuster wie abstrakte Fabrik, Fabrikmethode oder Erbauer umgesetzt.

Domain-driven Design

Repositories

Repositories abstrahieren die Persistierung und Suche von Fachobjekten. Mittels Repositories werden die technische Infrastruktur sowie alle Zugriffsmechanismen auf diese von der Geschäftslogikschicht getrennt.

Für alle Fachobjekte, welche über die Infrastruktur-Schicht geladen werden, wird eine Repository-Klasse bereitgestellt, welche die verwendeten Lade- und Suchtechnologien nach außen abkapselt. Die Repositories selbst sind Teil des Domänenmodells und somit Teil der Geschäftslogikschicht. Sie greifen als einzige auf die Objekte der Infrastruktur-Schicht zu.

Domain-driven Design

Architekturtechniken

Evolvierende Struktur (evolving order)

Systemmetapher (system metaphor)

Verantwortlichkeitsschichten (responsibility layers)

Wissenslevel (knowledge level)

Erweiterungsframeworks (pluggable component framework)

Domain-driven Design

Evolvierende Struktur *(evolving order)*

Große Strukturen im Domänenmodell sollten idealerweise erst mit der Zeit entstehen, beziehungsweise sich über die Zeit entwickeln.

Große Strukturen sollten möglichst einfach und mit möglichst wenigen Ausnahmen umgesetzt sein.

Domain-driven Design

Systemmetapher *(system metaphor)*

Die Systemmetapher ist ein Konzept aus Extreme Programming, welche die Kommunikation zwischen allen Beteiligten erleichtert, indem es das System mittels einer Metapher, einer inhaltlich ähnlichen, für alle Seiten verständlichen Alltagsgeschichte beschreibt. Diese sollte möglichst gut passen und zur Stärkung der ubiquitären Sprache verwendet werden.

Domain-driven Design

Verantwortlichkeitsschichten *(responsibility layers)*

Aufteilung des Domänenmodells in Schichten gemäß Verantwortlichkeiten. Domain-driven Design schlägt folgende Schichten vor:

- Entscheidungsschicht
- Regelschicht
- Zusagen
- Arbeitsabläufe
- Potential

Domain-driven Design

Wissenslevel *(knowledge level)*

Wissenslevel beschreibt das explizite Wissen über das Domänenmodell. Es ist in Situationen notwendig, wo die Abhängigkeiten und Rollen zwischen den Entitäten situationsbedingt variieren.

Das Wissenslevel sollte diese Abhängigkeiten und Rollen von außen anpassbar enthalten, damit das Domänenmodell weiterhin konkret und ohne unnötige Abhängigkeiten bleiben kann.

Domain-driven Design

Erweiterungsframeworks *(pluggable component framework)*

ist die Überlegung verschiedene Systeme über ein Komponentenframework miteinander zu verbinden.

Domain-driven Design

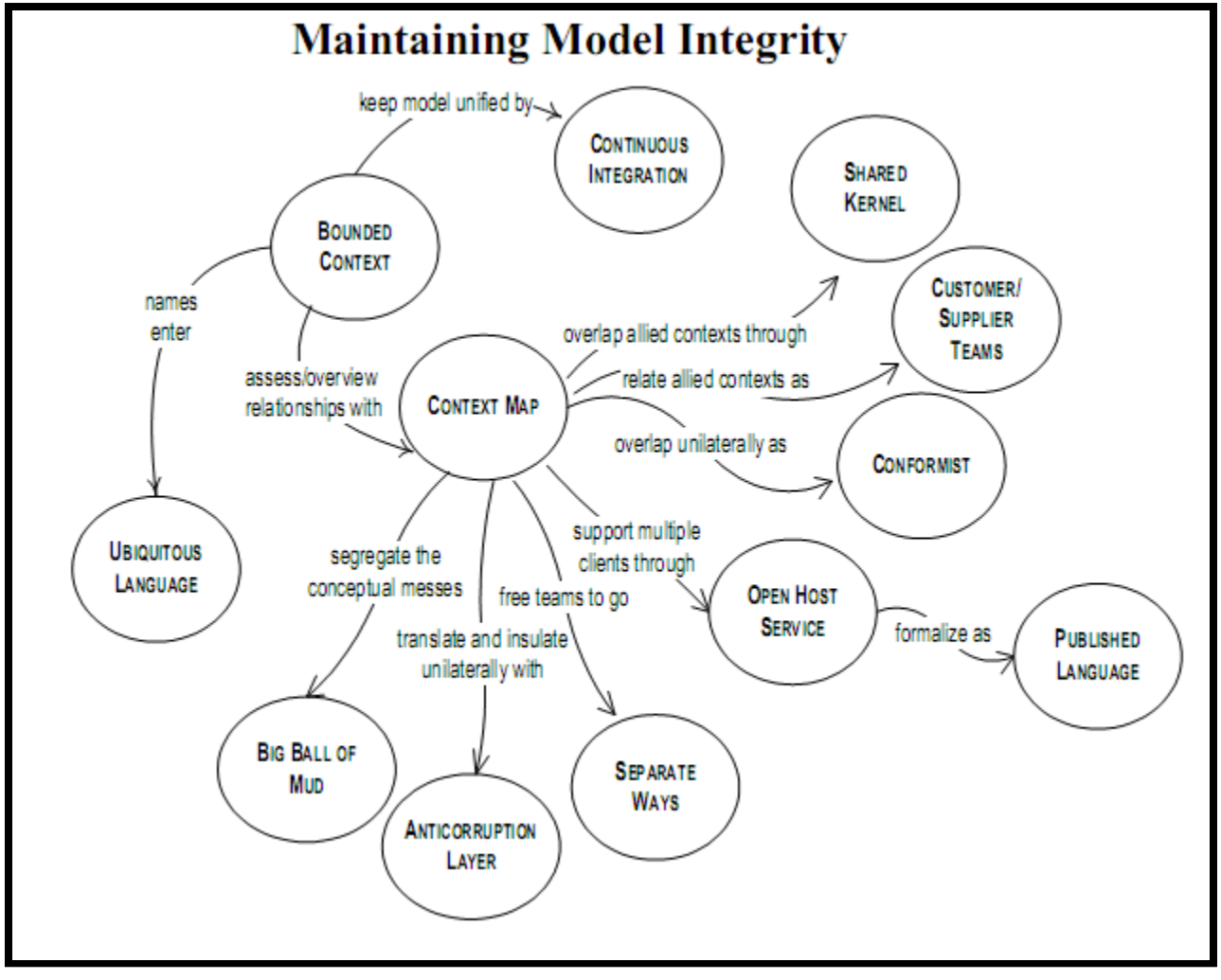
Vorgehensweisen

Domain-driven Design definiert eine Reihe von Vorgehensweisen, welche dazu dienen die Integrität der Modelle zu gewährleisten.

Dies ist insbesondere dann notwendig, wenn mehrere Teams unter unterschiedlichem Management und Koordination an verschiedenen Fachlichkeiten, aber in einem großen Projekt zusammenarbeiten sollen.

Domain-driven Design

Vorgehensweisen



Domain-driven Design

Vision der Fachlichkeit *(domain vision statement)*

ist eine kurze Beschreibung der hinter der Kernfachlichkeit stehenden Vision und der damit verbundenen Ziele.

Sie gibt die Entwicklungsrichtung des Domänenmodells vor und dient als Bindeglied zwischen Projektvision/Systemmetapher und den Details der Kernfachlichkeit und des Codes.

Domain-driven Design

Kontextübersicht *(context map)*

dient einer gesamthaften Übersicht über alle Modelle, deren Grenzen und Schnittstellen.

Dadurch wachsen die Kontexte nicht in Bereiche anderer Kontexte und die Kommunikation zwischen den Kontexten läuft über wohldefinierte Schnittstellen.

Domain-driven Design

Kontextgrenzen *(bounded context)*

beschreiben die Grenzen jedes Kontexts in vielfältiger Hinsicht wie beispielsweise Teamzuordnung, Verwendungszweck, dahinter liegende Datenbankschemata.

Damit wird klar, wo ein Kontext seine Gültigkeit verliert und potentiell ein anderer Kontext seinen Platz einnimmt.

Domain-driven Design

Kernfachlichkeit *(core domain)*

ist der wertvollste Teil des Domänenmodells, der Teil, welcher am meisten Anwendernutzen stiftet.

Die anderen Teile des Domänenmodells dienen vor allem dazu die Kernfachlichkeit zu unterstützen und mit weniger wichtigen Funktionen anzureichern.

Bei der Modellierung sollte besonderes Augenmerk auf die Kernfachlichkeit gelegt werden und sie sollte durch die besten Entwickler umgesetzt werden.

Domain-driven Design

Geteilter Kern *(shared kernel)*

ist ein Teil der Fachlichkeit der zwischen unterschiedlichen Projektteilen geteilt wird.

Dies ist sinnvoll, wenn die verschiedenen Projektteile nur lose miteinander verbunden sind und das Projekt zu groß ist um in einem Team umgesetzt zu werden. Der geteilte Kern wird hierbei von allen Projektteams, die ihn nützen, gemeinsam entwickelt.

Dies benötigt sowohl viel Abstimmungs- als auch Integrationsaufwand.

Domain-driven Design

Kunde-Lieferant *(customer-supplier)*

ist die Metapher für die Beziehung zwischen Projektteams, bei denen ein Team eine Fachlichkeit umsetzt, auf die das andere Team aufbaut.

Damit wird sichergestellt, dass das abhängige Team vom umsetzenden Team gut unterstützt wird, da ihre Anforderungen mit derselben Priorität umgesetzt werden, wie die eigentlichen Anforderungen an das Lieferantenteam.

Domain-driven Design

Separierter Kern *(segregated core)*

bezeichnet die Überlegung die Kernfachlichkeit, auch wenn sie eng mit unterstützenden Modellelementen gekoppelt ist, in ein eigenes Modul zu verlagern und die Kopplung mit anderen Modulen zu reduzieren.

Damit wird die Kernfachlichkeit vor hoher Komplexität bewahrt und die Wartbarkeit erhöht.

Domain-driven Design

Generische Sub-Fachlichkeiten *(generic subdomains)*

bezeichnet die Idee, diejenigen Teile des Domänenmodells, welche nicht zur Kernfachlichkeit gehören, in Form von möglichst generischen Modellen in eigenen Modulen abzulegen.

Diese könnten, da sie nicht die Kernfachlichkeit repräsentieren und generisch sind, outgesourced entwickelt oder durch Standardsoftware ersetzt werden.

Domain-driven Design

Kontinuierliche Integration *(continuous integration)*

dient beim Domain-driven Design dazu, alle Veränderungen eines Domänenmodells laufend miteinander zu integrieren und gegen bestehende Fachlichkeit automatisiert testen zu können.

Domain-driven Design

Literatur

”Domain-Driven Design. Tackling Complexity in the Heart of Software”

(Eric Evans)

Fragen?

Unterlagen: ai2016.nils-loewe.de