Architektur von Informationssystemen

Hochschule für angewandte Wissenschaften

Sommersemester 2016

Nils Löwe / nils@loewe.io / @NilsLoewe

Praktikumsaufgabe

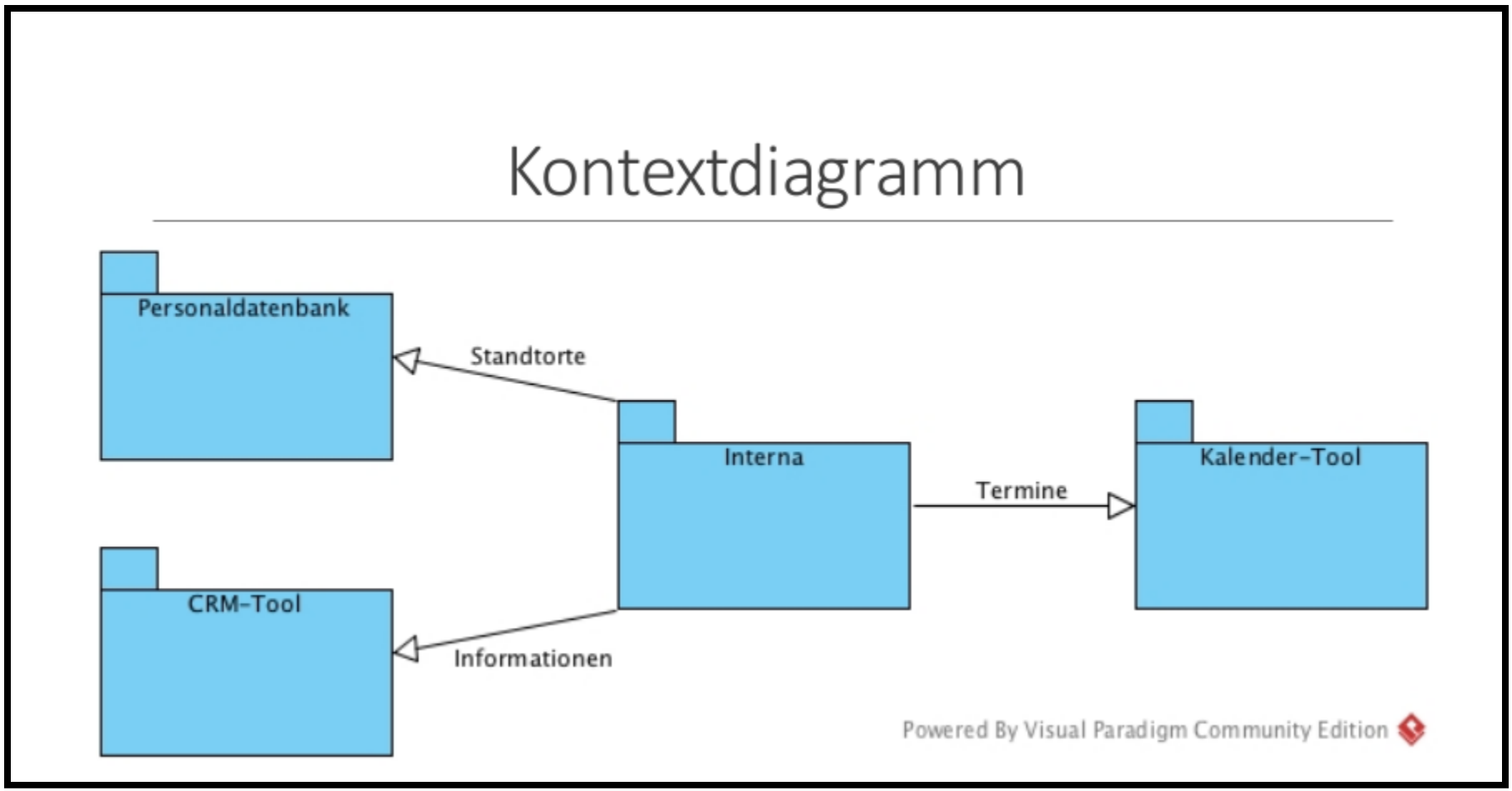
2. Praktikumsaufgabe

Fragen?

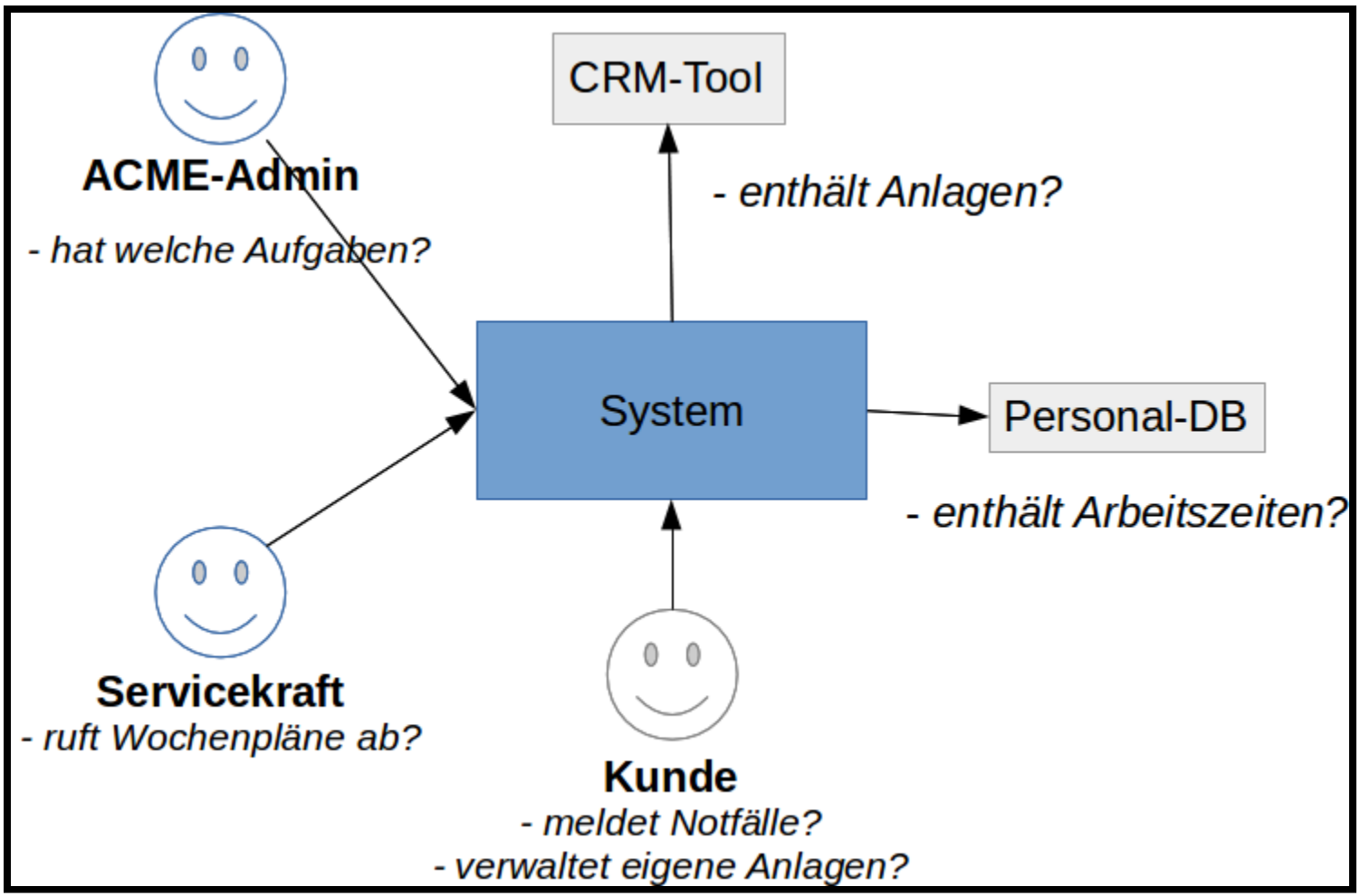
1. Praktikumsaufgabe - Rückblick

- 13 Kontextdiagramme
- 147 Fragen
- 18 Technologievorschläge

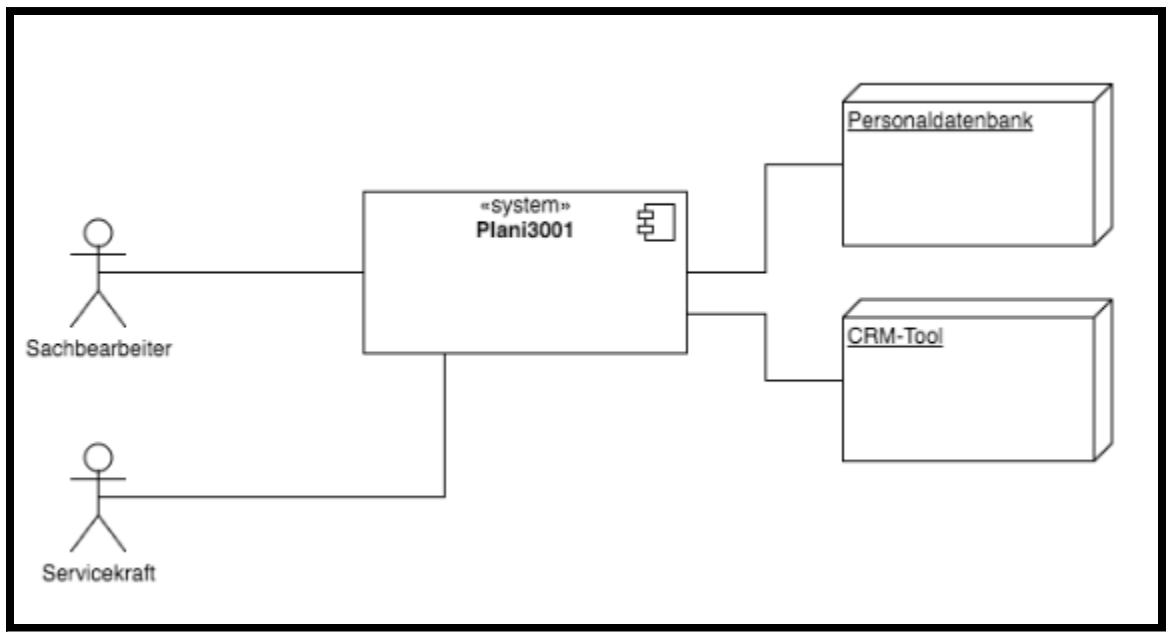
1. Praktikumsaufgabe - Kontextdiagramme



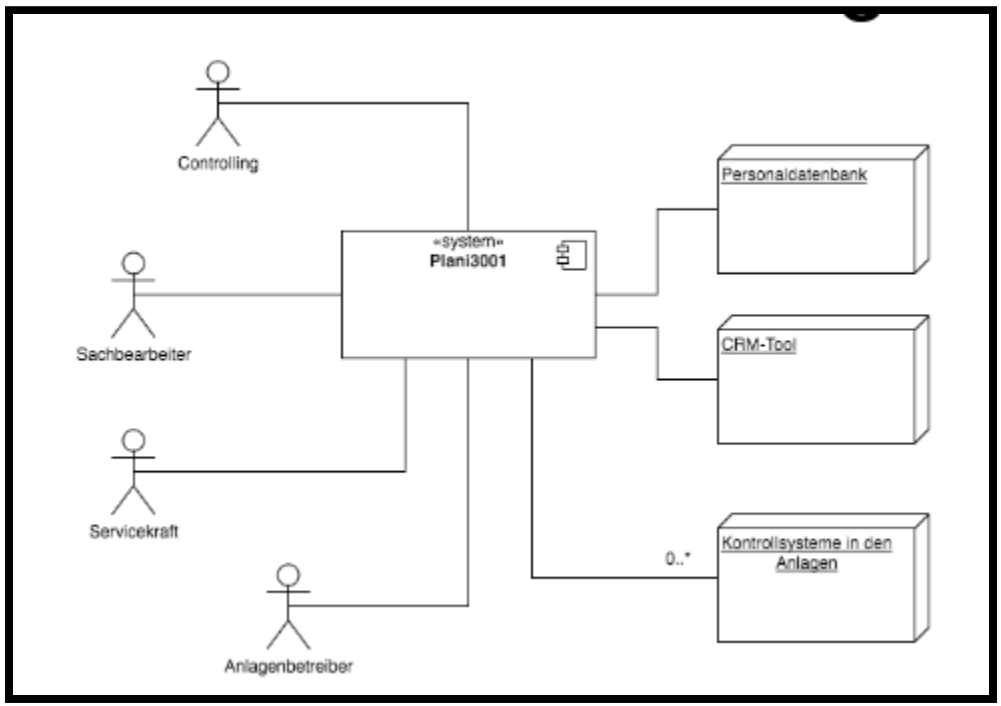
1. Praktikumsaufgabe - Kontextdiagramme



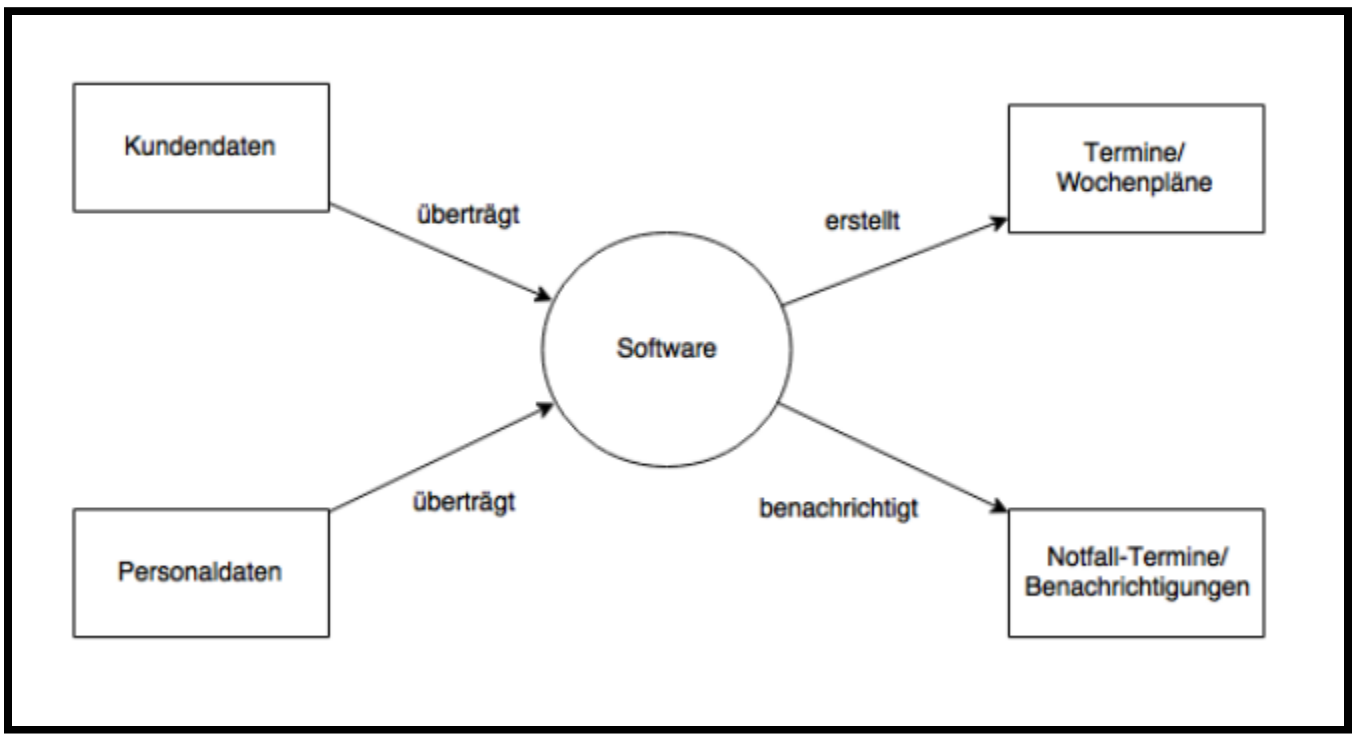
1. Praktikumsaufgabe - Kontextdiagramme



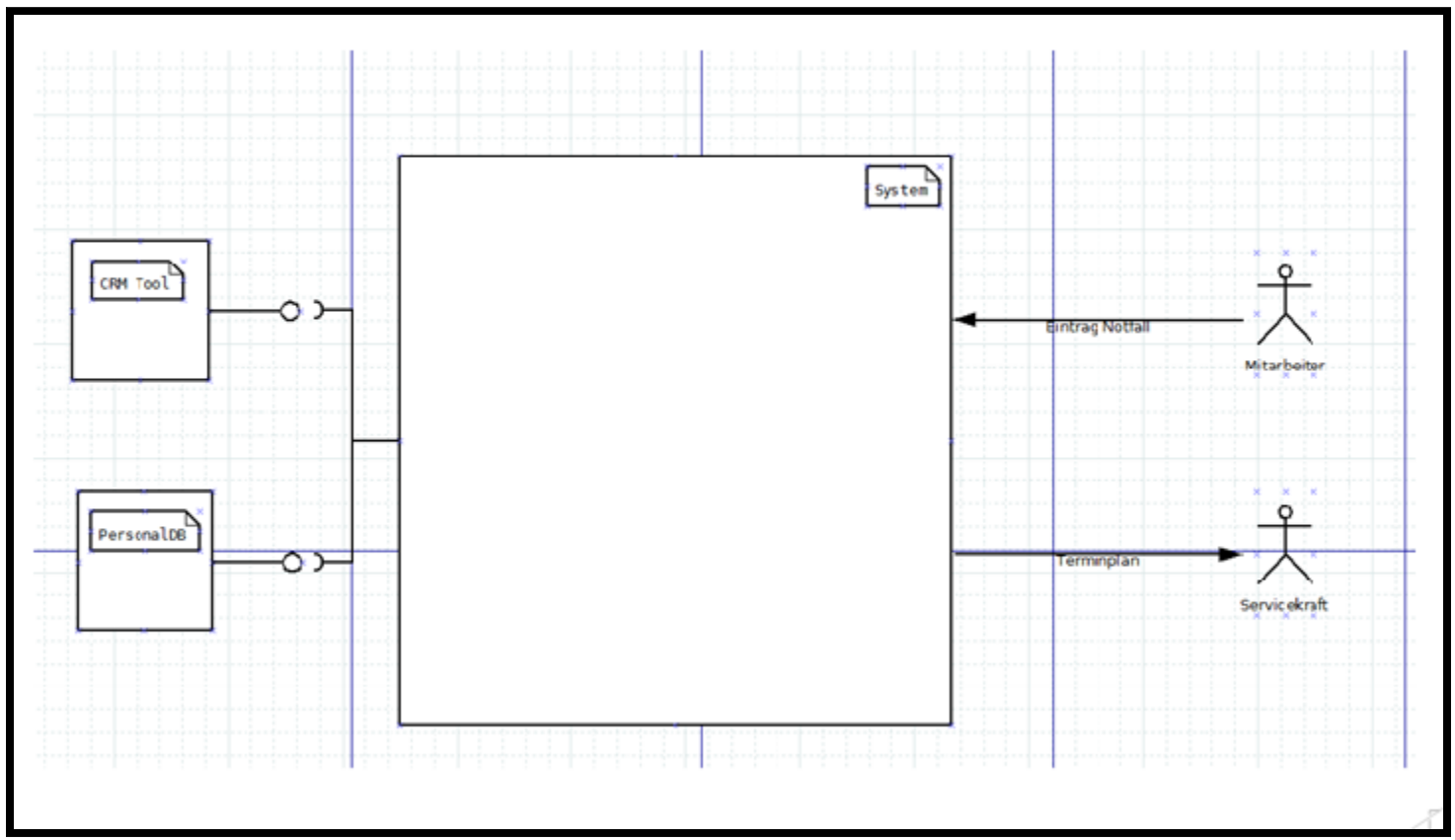
1. Praktikumsaufgabe - Kontextdiagramme



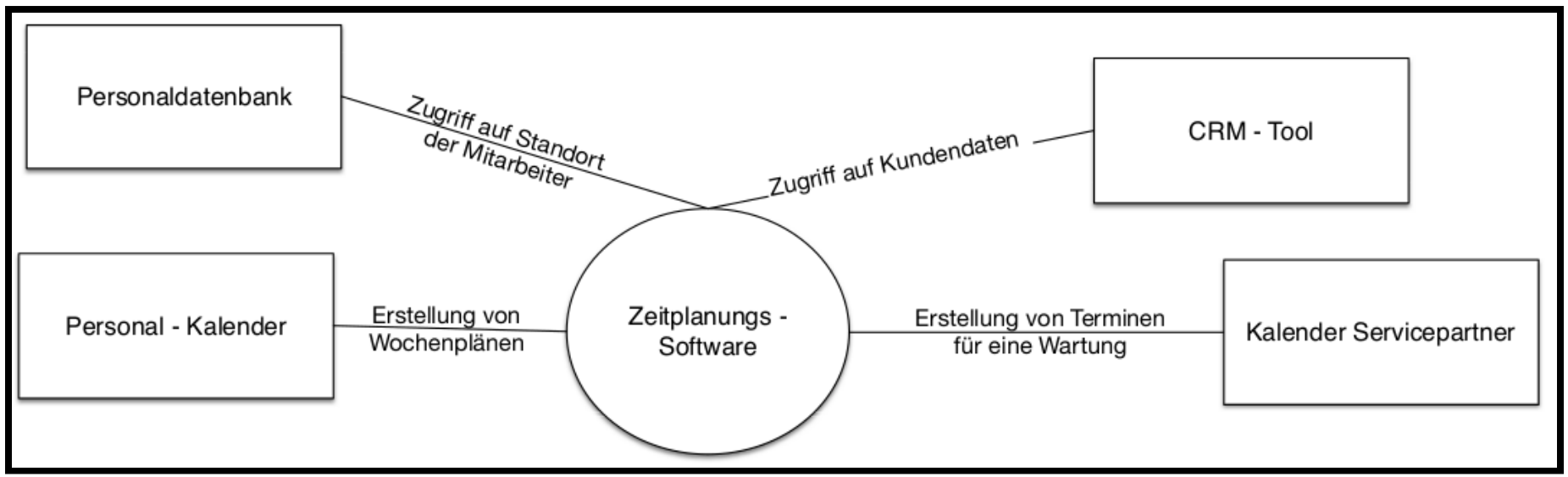
1. Praktikumsaufgabe - Kontextdiagramme



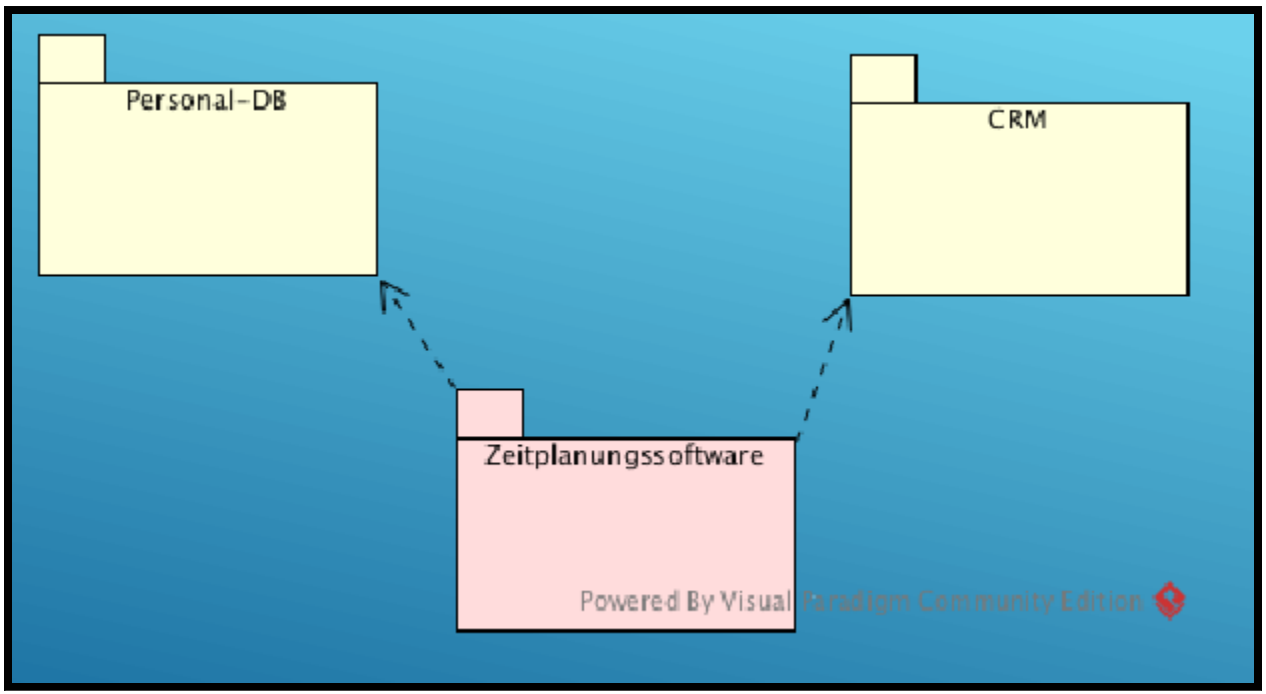
1. Praktikumsaufgabe - Kontextdiagramme



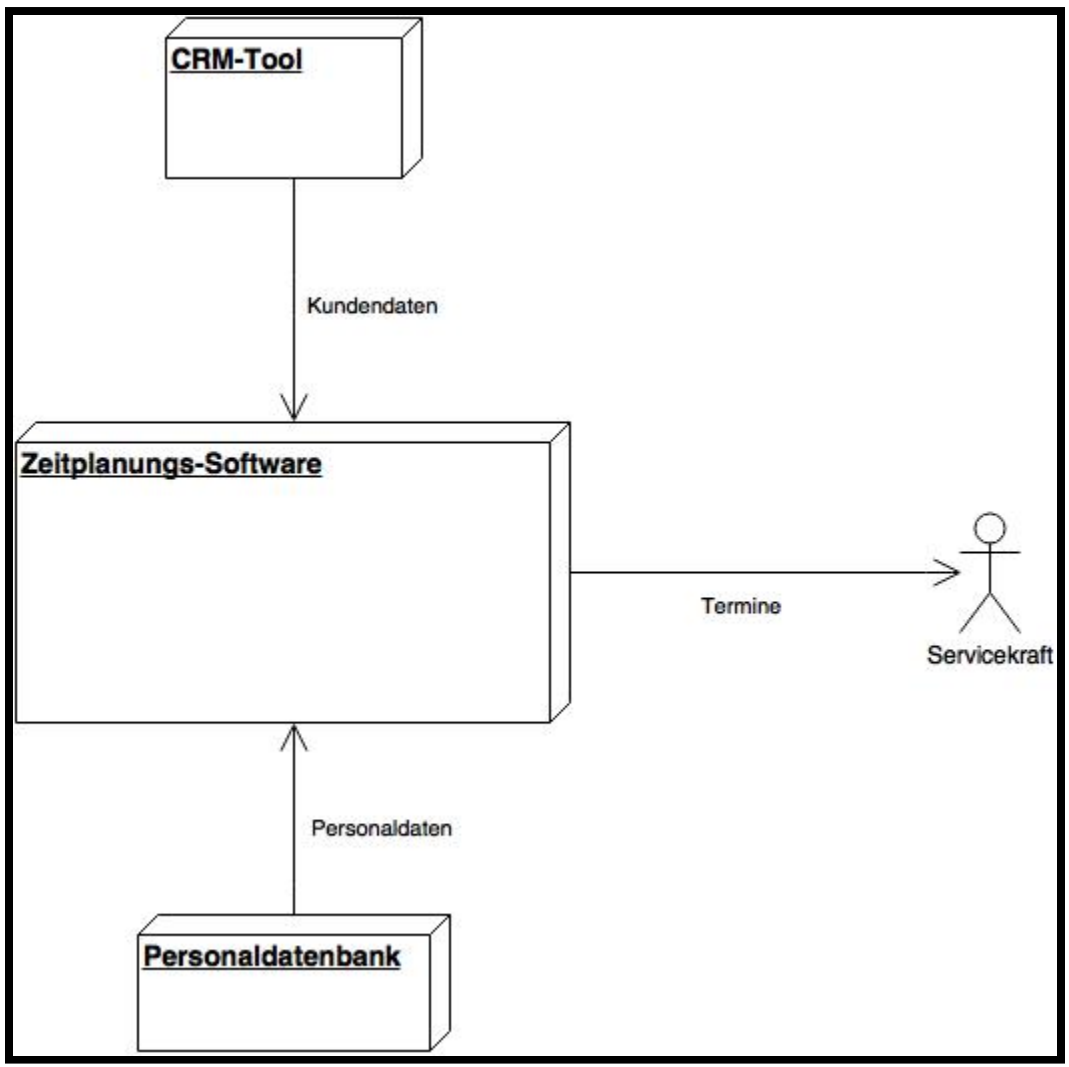
1. Praktikumsaufgabe - Kontextdiagramme



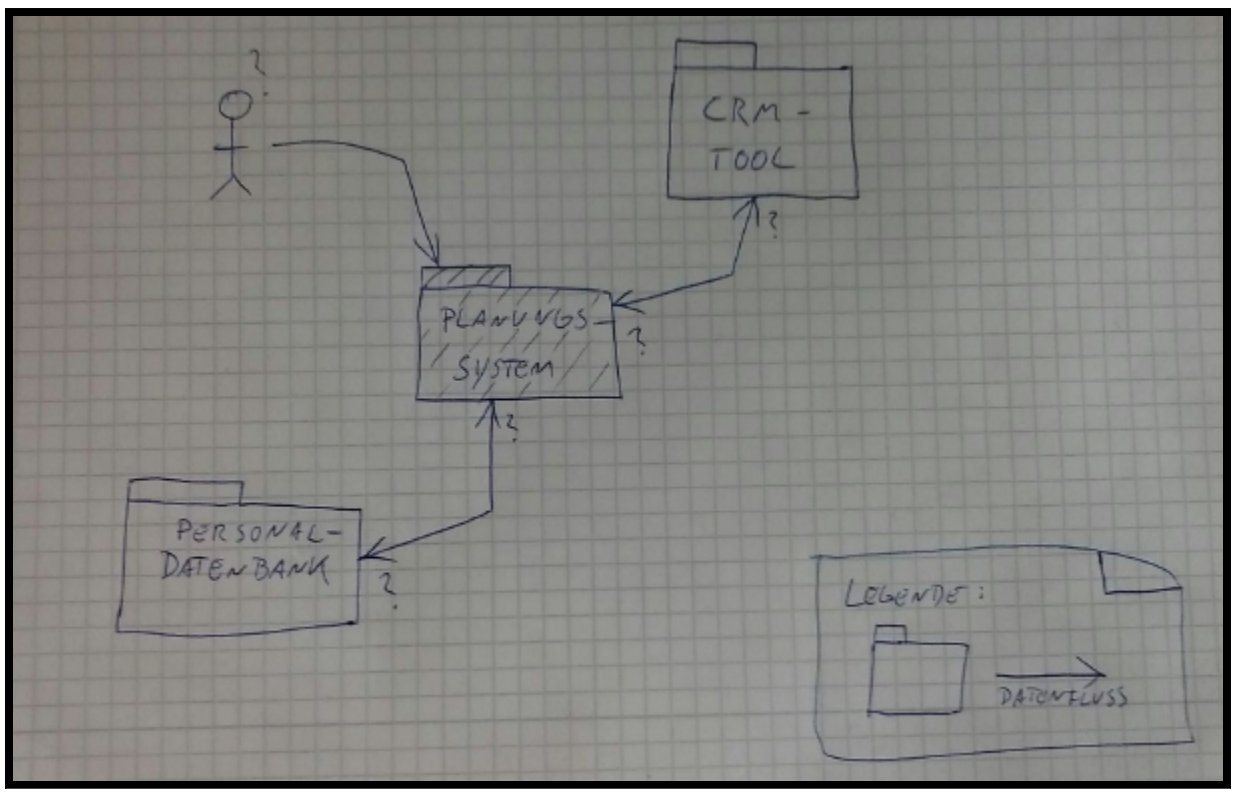
1. Praktikumsaufgabe - Kontextdiagramme



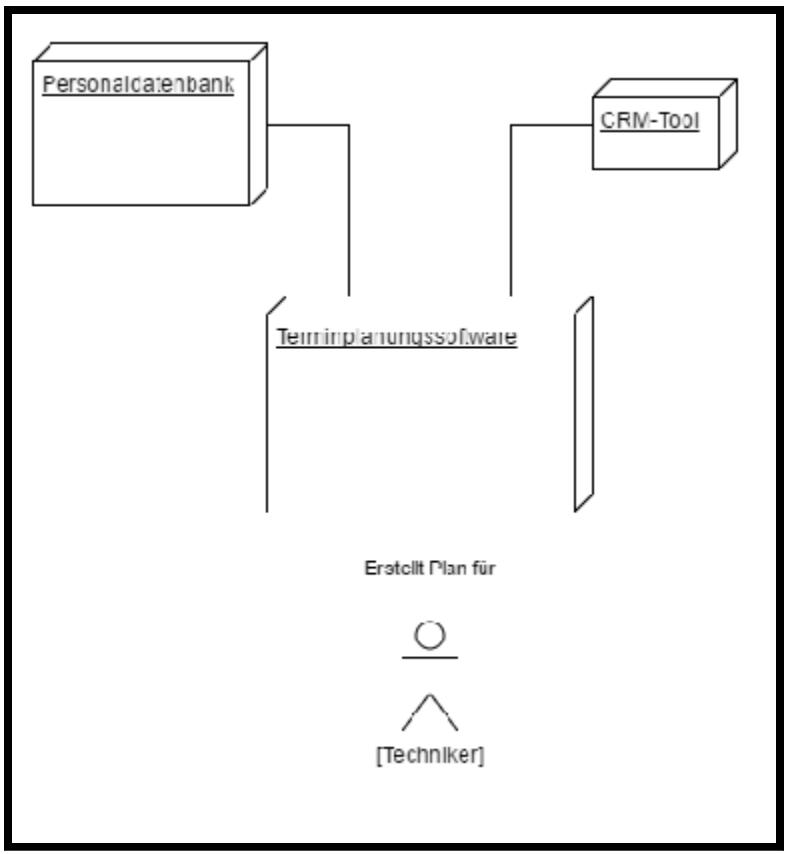
1. Praktikumsaufgabe - Kontextdiagramme



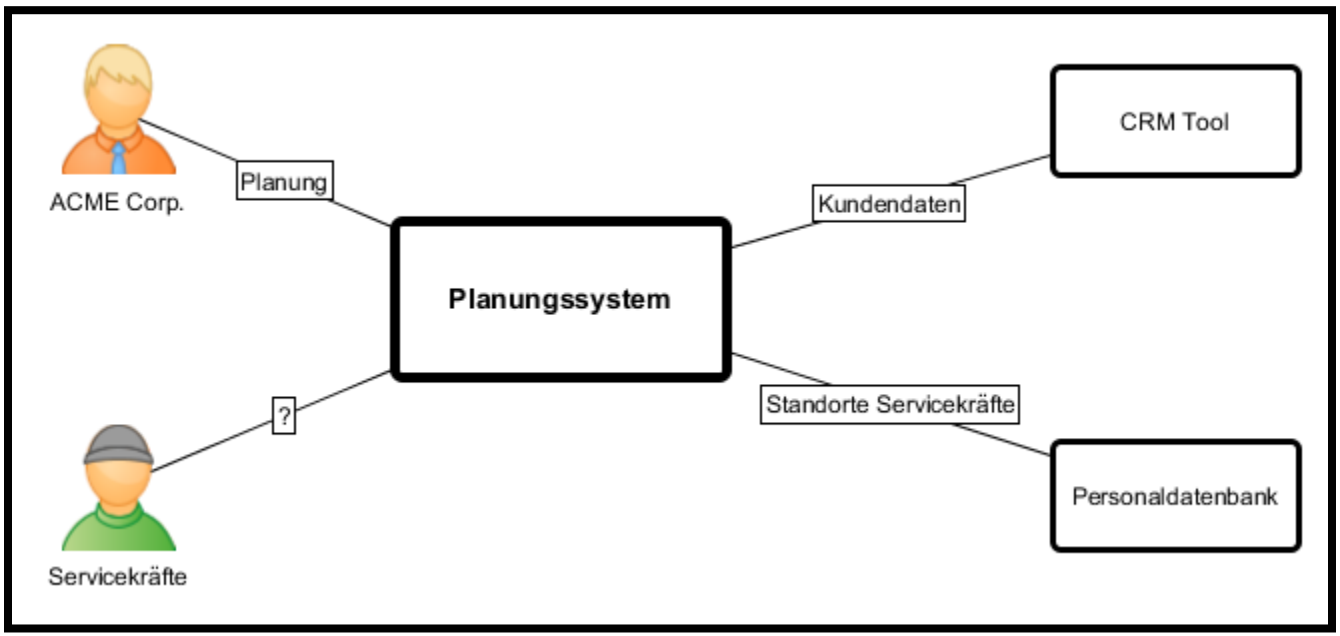
1. Praktikumsaufgabe - Kontextdiagramme



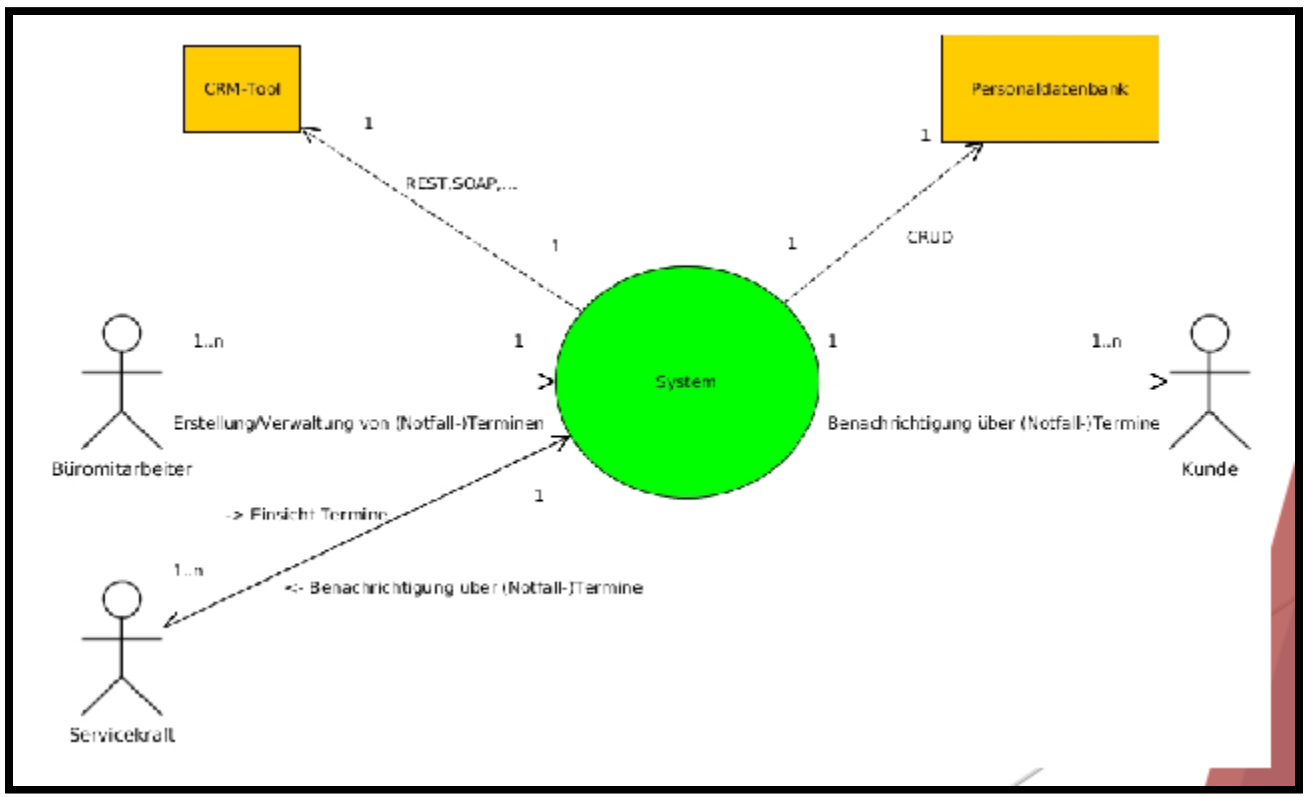
1. Praktikumsaufgabe - Kontextdiagramme



1. Praktikumsaufgabe - Kontextdiagramme



1. Praktikumsaufgabe - Kontextdiagramme



1. Praktikumsaufgabe - Fachliche Fragen

- Wie ist die aktuelle Vorgehensweise bei der Verteilung der Termine?
- Wie werden die Aufträge spezifiziert?
- Nach welche Kriterien werden die Servicekräfte eingeteilt?
- Wie lange dauert eine Wartung?
- Soll ein bestimmter (halb-) Tag freigehalten werden für Notfälle
- Wie groß ist der Intervall zwischen den Wartungsterminen?
- Was geschieht tatsächlich automatisch und wo ist das Zutun eines Mitarbeiters notwendig?
- Wer benutzt das System?
- Wer kann und darf was?
- Wie werden Notfälle gemeldet?
- Kann jede Servicekraft jede Anlage warten? (verschiedene Hersteller)

1. Praktikumsaufgabe - Fachliche Fragen

- In welchem Zeitrahmen soll das Projekt abgewickelt werden?
- Woher kommen die Termine? Welche Informationen beinhalten diese?
- Gibt es ein System, das einen Notfall meldet, oder erfolgt das per Anruf, Mail, etc.?
- Sollen die Wochenpläne automatisch oder manuell erstellt werden?
- Soll unsere Firma in Zukunft die Wartung der Software übernehmen?
- Haben Anlagenbetreiber Zugriff auf das System, oder wie sollen Fragen, Termine etc. geklärt werden?
- Gibt es eine Controlling-Abteilung?
- Brauchen die Mitarbeiter Zugriff auf das System?
- Bedarf es für unterschiedliche Anlagentypen (Nordex, Enercon, etc.) verschiedene Qualifikationen der Servicekräfte?
- Wie lange ist mindestens ein Sachbearbeiter verfügbar?

- Wie lange ist mindestens ein Sachbearbeiter verfügbar?
- Wie gelangen die Informationen an die Servicekräfte?

1. Praktikumsaufgabe - Fachliche Fragen

- Gibt es Bereitschaftsdienste für Servicekräfte und Sachbearbeiter?
- Wie ist der Prozess vom manuellen Planen derzeit?
- Wie lange dauern die Einsätze für gewöhnlich?
- Wie sollen Servicekräfte auf anfallende Termine aufmerksam gemacht werden?
- Wie präzise müssen die Termine sein?
- In welchen Zeiten finden normale Termine statt?
- Können normale Termine für Notfalltermine verschoben werden
- Was für Informationen gehören neben Datum und Uhrzeit in einen Termin/Wochenplan/Notfalltermin?
- Was unterscheidet einen normalen Termin von einem Notfalltermin?
- Sollen die Wochenpläne automatisch aus den Terminen und Notfallterminen erstellt werden?
- Wann sollen die Wochenpläne erstellt werden?

Wann sollen die Wochenpläne erstellt werden?

1. Praktikumsaufgabe - Fachliche Fragen

- Wie sollen die Servicekräfte den Terminplan bekommen?
- Sollen die Servicekräfte die Möglichkeit haben die zugeteilten Termine abzulehnen?
- Sollen abgelehnte Termine automatisch anderen zugewiesen werden?
- Wie werden die regelmäßigen Wartungsintervalle festgelegt?
- Übernimmt Planungssystem andere Aufgaben als reine Planung, zb. Informieren der Mitarbeiter?
- Wer ist für Installation zuständig?
- Haben Mitarbeiter verschiedene Zeitkontingente?
- Gibt es Qualifizierungs-Unterschiede zwischen den Mitarbeitern?
- Steht im Notfall-Fall jeder Mitarbeiter zur Verfügung oder gibt es Bereitschaftzeiten?
- Wie lange soll im Voraus geplant werden?
- Welche Reaktionszeit im Notfall?

1. Praktikumsaufgabe - Fachliche Fragen

- Gibt es Bereitschafts-/Springerdienste (für Notfälle/Krankheitsvertretung etc.)?
- Gibt es individuellen Mitarbeiterkalender (Urlaubstage etc.)?
- Gibt es Präferenzen, welcher Mitarbeiter bei welchem Kunden/Anlage eingesetzt werden soll? Bzw. andere Dinge als Verfügbarkeit bei der Planung zu beachten?
- Wie gut muss die automatische Planung optimiert werden?
- Kennen Sie bereits eine Planungssoftware die Ihnen gut gefällt?
- Sollen nur Arbeitseinsätze geplant werden? Oder auch generelle Arbeitszeit, Urlaub, Abrechnungen etc. ?
- Fahrwege der Mitarbeiter?
- Wie werden diese Hersteller bezogenen Daten gepflegt?
- Werden in Zukunft Hersteller hinzukommen oder wegfallen?
- Wie ist das Vorgehen bei Ausfall von Mitarbeitern?
- Gibt es immer ein Mitarbeiter pro Anlage oder kann das jeder machen?

1. Praktikumsaufgabe - Fachliche Fragen

- Sollen Termine manuell verändert werden können?
- Können wir uns persönlich treffen?
- Welche Möglichkeiten haben wir die Spezifikation zu erarbeiten?
- Wer ist Ansprechpartner für fachliche/technische/organisatorische Fragen?

1. Praktikumsaufgabe - Nicht-Funktionale Fragen

- Anzahl der Benutzer des zukünftigen Systems?
- Über wie viele Kundendatensätze verfügen Sie?
- Zahl der betreuten Kunden, Anlagen, …
- wie lang darf die Berechnung der Wochenpläne dauern?
- in welchem Maße wird das Unternehmen in Zukunft voraussichtlich wachsen?
- Ausfall des Systems gefährdet potentiell das Kapital der betreuten Kunden?
- Benutzbarkeit - Besondere Anforderungen?
- Erfüllung von DIN-Normen gewünscht/erforderlich?
- wie wahrscheinlich ist eine Änderung der Anforderungen in der Zukunft?
- rechtliche Rahmenbedingungen?
- besondere Anforderungen an die Benutzerauthentifizierung?

1. Praktikumsaufgabe - Nicht-Funktionale Fragen

- Wie viele Nutzer greifen zeitgleich zu?
- Welche Nutzergruppen gibt es?
- Welche Performanceanforderungen gibt es? Zum Beispiel Zeit zum Laden einer Seite
- Gibt es Benutzer mit Behinderungen? Wenn ja, welche?
- Welche Sicherheitsstandards müssen erfüllt werden?
- Müssen Lizenzen gekauft oder eingebunden werden?
- Welche SLAs mit den Kunden gibt es?
- Wie technikaffin sind die Benutzer?
- Gibt es Gestaltungsvorlagen / -anforderungen für die GUI?
- Welchen Informationsgehalt hat ein Kalendereintrag?
- Welche Antwortzeiten sind erwartet?

1. Praktikumsaufgabe - Nicht-Funktionale Fragen

- Service-Level-Agreement?
- Mobile Benutzbarkeit?
- wie schnell sollen Notfalltermine berechnet und verteilt werden?
- gibt es Anforderungen an die Programmiersprache/ Systemarchitektur?
- gibt es rechtliche Einschränkungen? (Datenschutz, unternehmensspezifische Rechte)
- welches Budget steht zur Verfügung?
- gibt es eine Deadline?
- soll es einen anschließenden Supportvertrag geben?
- SaaS oder Selbsthosting?
- Wie viele Pläne muss die Software verwalten können?
- Was für eine Datenmenge muss verarbeitet werden? (Anzahl Kunden/Mitarbeiter)

1. Praktikumsaufgabe - Nicht-Funktionale Fragen

- Was für Antwortzeiten werden erwartet? (Für Wochenplan und Notfalltermin)
- Muss das System skalierbar sein?
- Welche Ansprüche bestehen für das Look and Feel?
- Welche Ressourcen stehen zur Verfügung? (Hardware)
- Muss das System erweiterbar sein?
- Gibt es konkrete Sicherheitsanforderungen?
- Gibt es GUI? Falls ja, welche Anforderungen gibt es daran?
- Wieviele Zugriffe pro Tag/Stunde/Minute?
- Zeitanforderungen (Echtzeit, Batchorientiert?)
- Wieviele Daten werden im Durchschnitt übertragen?

1. Praktikumsaufgabe - Technische Fragen

- Welche Fremdsysteme nutzen Sie neben dem CRM-Tool und der DB?
- Wird zusätzlich ein Rechnungs- / Buchhaltungssystem benötigt?
- Wie ist die Rechteverteilung bei der Auftragserstellung geregelt?
- Welche Betriebssprachen haben Sie?
- Arbeiten sie International z.B. auch in Englisch
- Wo soll die Software gehostet werden?
- Sind Server vorhanden?

1. Praktikumsaufgabe - Nicht-Funktionale Fragen

- Welches CRM? Welche Personaldatenbank? Was ist alles erfasst?
- Sollen CRM und Personaldatenbank aus dem System heraus veränderbar sein oder gibt es nur Lesezugriff?
- Soll das System permanent erreichbar sein?
- Auf wie vielen Systemen soll die Software laufen?
- Welche Betriebssysteme auf welcher Hardware müssen unterstützt werden?
- welche Endgeräte sind bei Ihnen im Einsatz? (Plattform)
- Wie ist die Infrastruktur ?Wie ist die Software-Umgebung für das Zielsystem?

1. Praktikumsaufgabe - Rückblick

Fragen?

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualiät und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Wiederholung
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.

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

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.

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.

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.

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

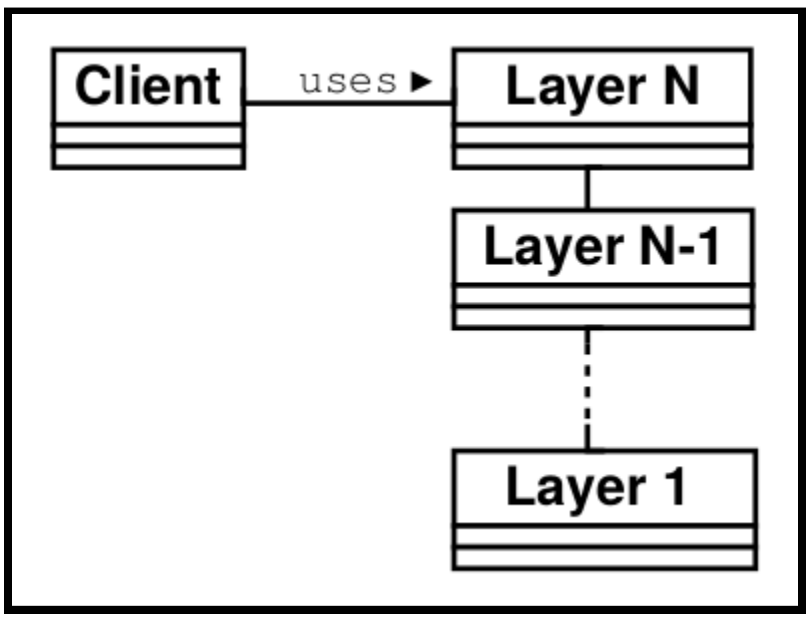
Layers

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

Layers

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

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

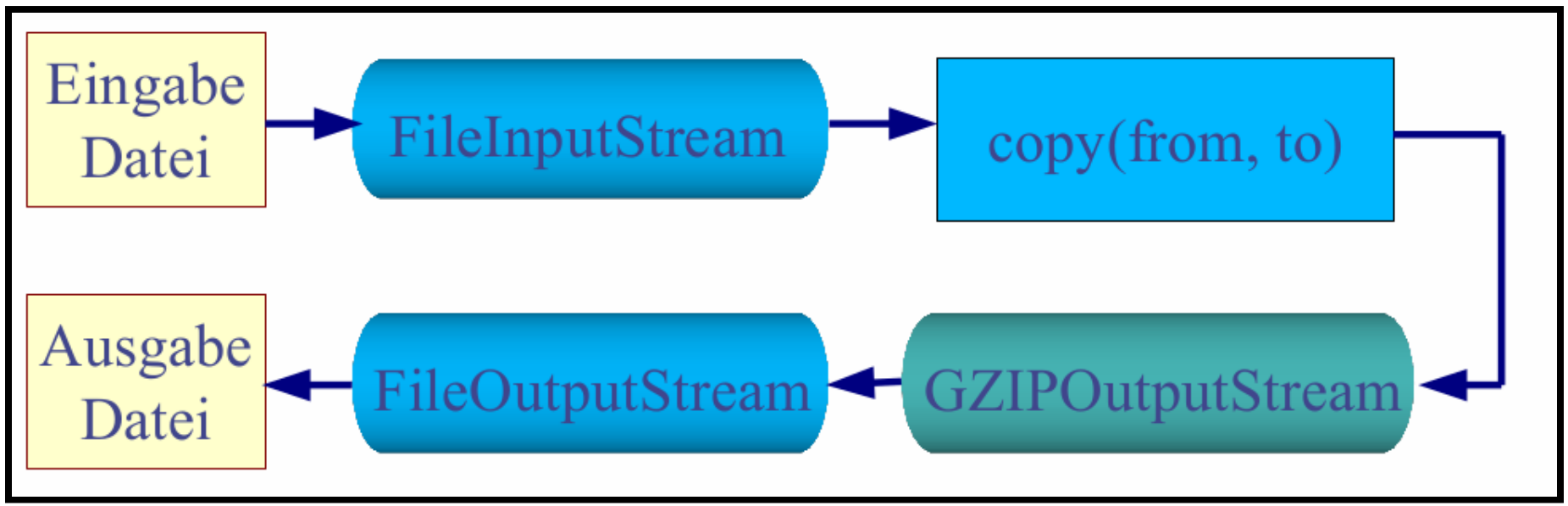
Blackboard

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 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

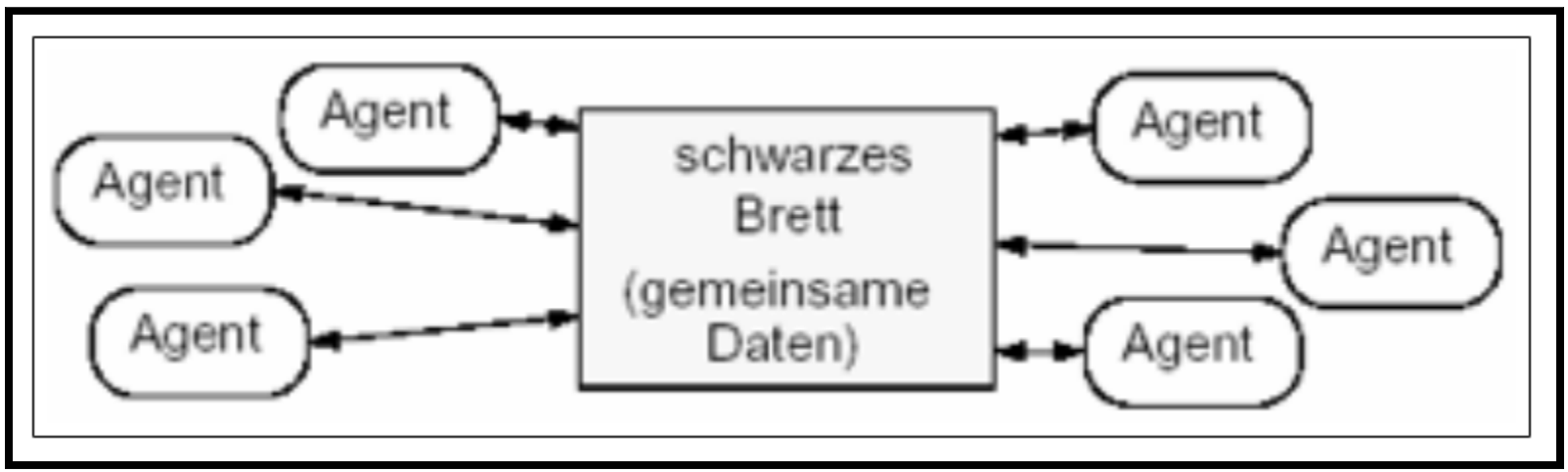
Blackboard

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

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

Konzepte

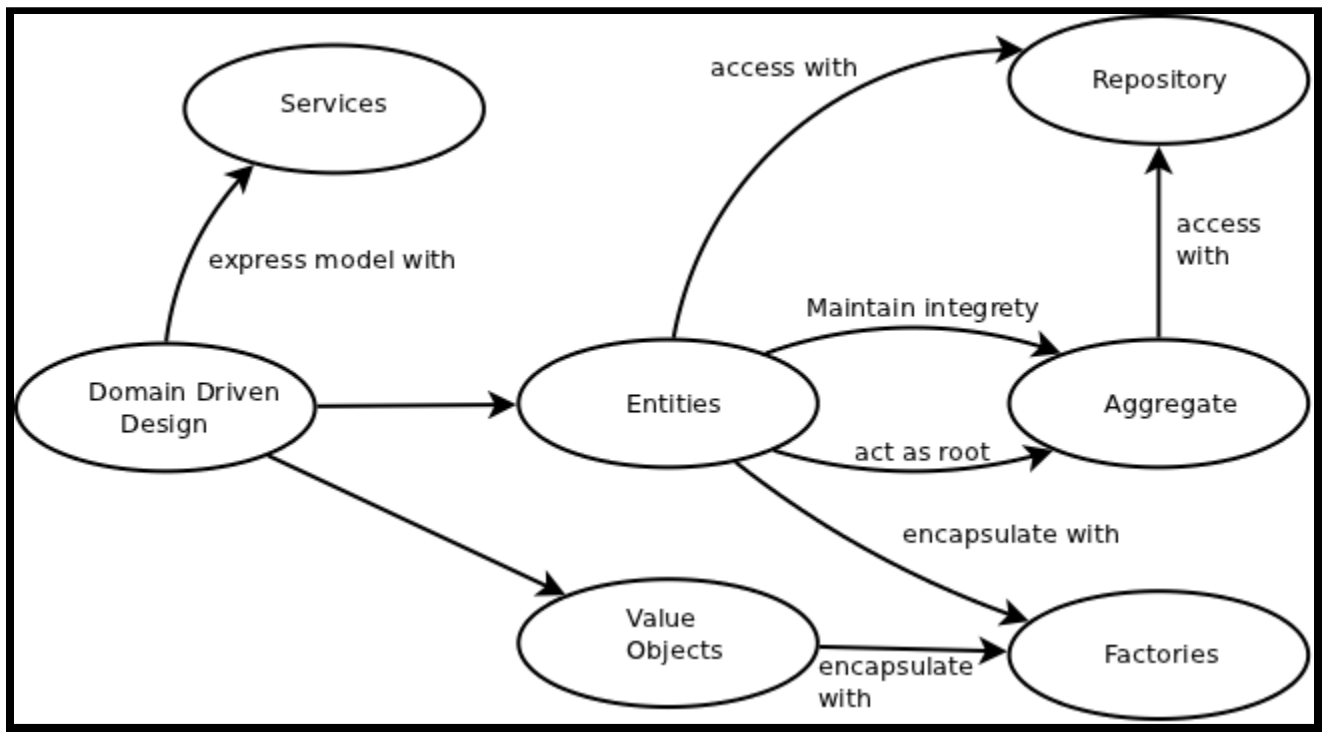
Domain-driven Design basiert auf einer Reihe von Konzepten, welche bei der Modellierung, aber auch anderen Tätigkeiten der Softwareentwicklung, berücksichtigt werden sollten.

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

Domain-driven Design

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.

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

Fragen?

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

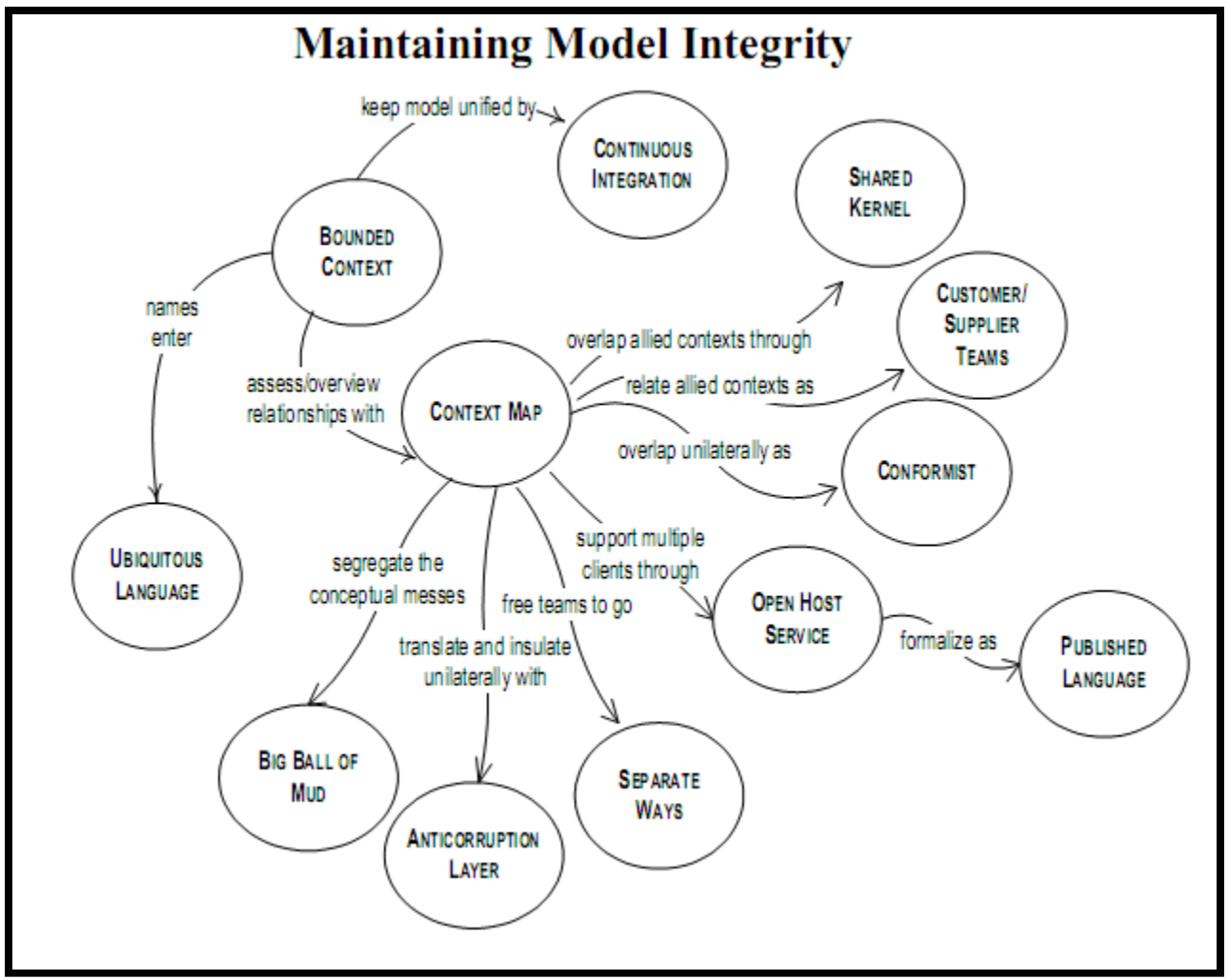
Vorgehensweisen

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?

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)

Beispiel für einen Geschäftsprozess: **„Vergib einen Kredit"**

- Auf einer hohen Ebene angesiedelt
- Zusammengesetzt aus
- *„Eröffnen der Geschäftsbeziehung"*
- *„Eröffnen eines oder mehrerer Konten"*
- *„Kreditvertrag"*

Serviceorientierte Architektur (SOA)

Beispiel für einen Geschäftsprozess: **„Trage den Kunden ins Kundenverzeichnis ein"**

- Auf einer niedrigeren Ebene angesiedelt.
- Durch Zusammensetzen (Orchestrierung) von Services niedriger Abstraktionsebenen können flexibel und wiederverwendbar Services höherer Abstraktionsebenen geschaffen werden.

Serviceorientierte Architektur (SOA)

Maßgeblich sind nicht technische Einzelaufgaben wie Datenbankabfragen, Berechnungen und Datenaufbereitungen, sondern die Zusammenführung dieser IT-Leistungen zu „höheren Zwecken“, die eine Organisationsabteilung anbietet.

Serviceorientierte Architektur (SOA)

Bei SOA handelt es sich um eine Struktur, welche die Unternehmensanwendungsintegration ermöglicht, indem die Komplexität der einzelnen Anwendungen („Applications“) hinter den standardisierten Schnittstellen verborgen wird.

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)

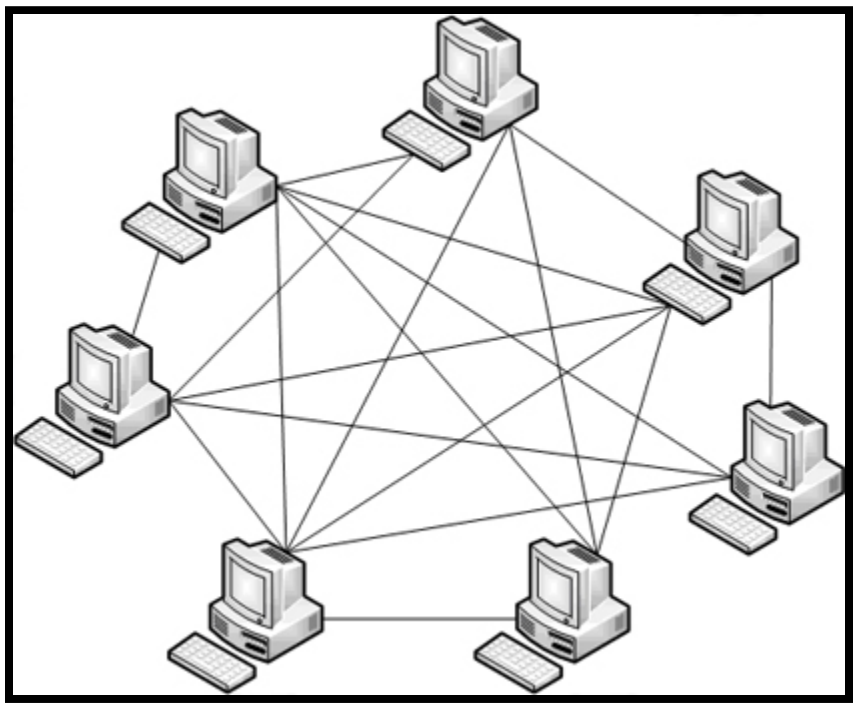
Peer-to-Peer

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

In modernen P2P-Netzwerken werden die Netzwerkteilnehmer abhängig von ihrer Qualifikation in verschiedene Gruppen eingeteilt, die spezifische Aufgaben übernehmen. Kernkomponente aller modernen Peer-to-Peer-Architekturen, ist daher ein zweites internes Overlay-Netz, welches normalerweise aus den besten Computern des Netzwerks besteht und die Organisation der anderen Computer sowie die Bereitstellung der Such-Funktion übernimmt.

Peer-to-Peer

- Mit der Suchfunktion ("lookup") können Peers im Netzwerk diejenigen Peers identifizieren, die für ein bestimmtes Objekt zuständig sind.
- Strukturierte Peer-to-Peer Netze: Die Verantwortlichkeit für jedes einzelne Objekt ist mindestens einem Peer fest zugeteilt
- Unstrukturierte Peer-to-Peer Netze: Es gibt für die Objekte im P2P-System keine Zuordnungsstruktur

Peer-to-Peer

- Sobald die Peers, die die gesuchten Objekte halten, identifiziert wurden, wird die Datei (in Dateitauschbörsen) direkt von Peer zu Peer übertragen.
- Es existieren unterschiedliche Verteilungsstrategien, welche Teile der Datei von welchem Peer heruntergeladen werden soll, z. B. BitTorrent.

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

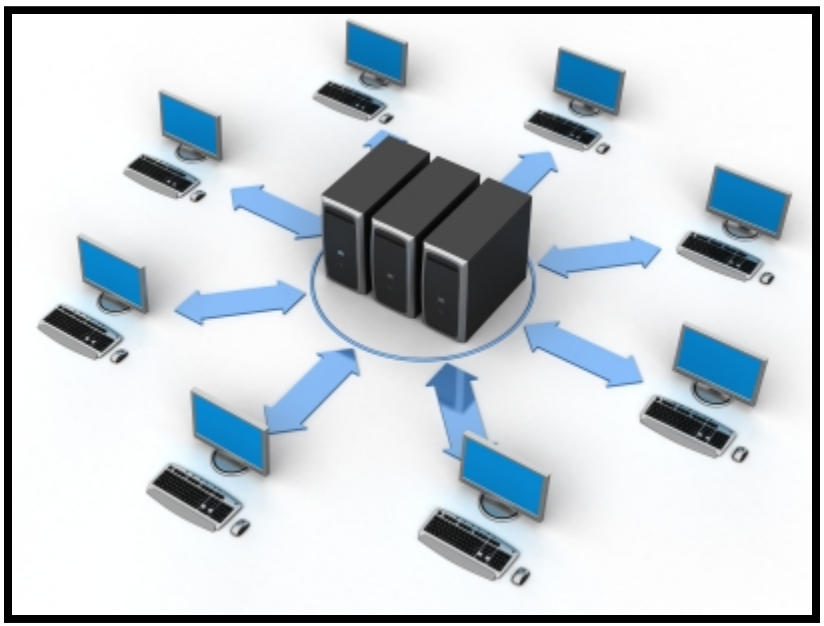
Client-Server

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

Fragen?

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)

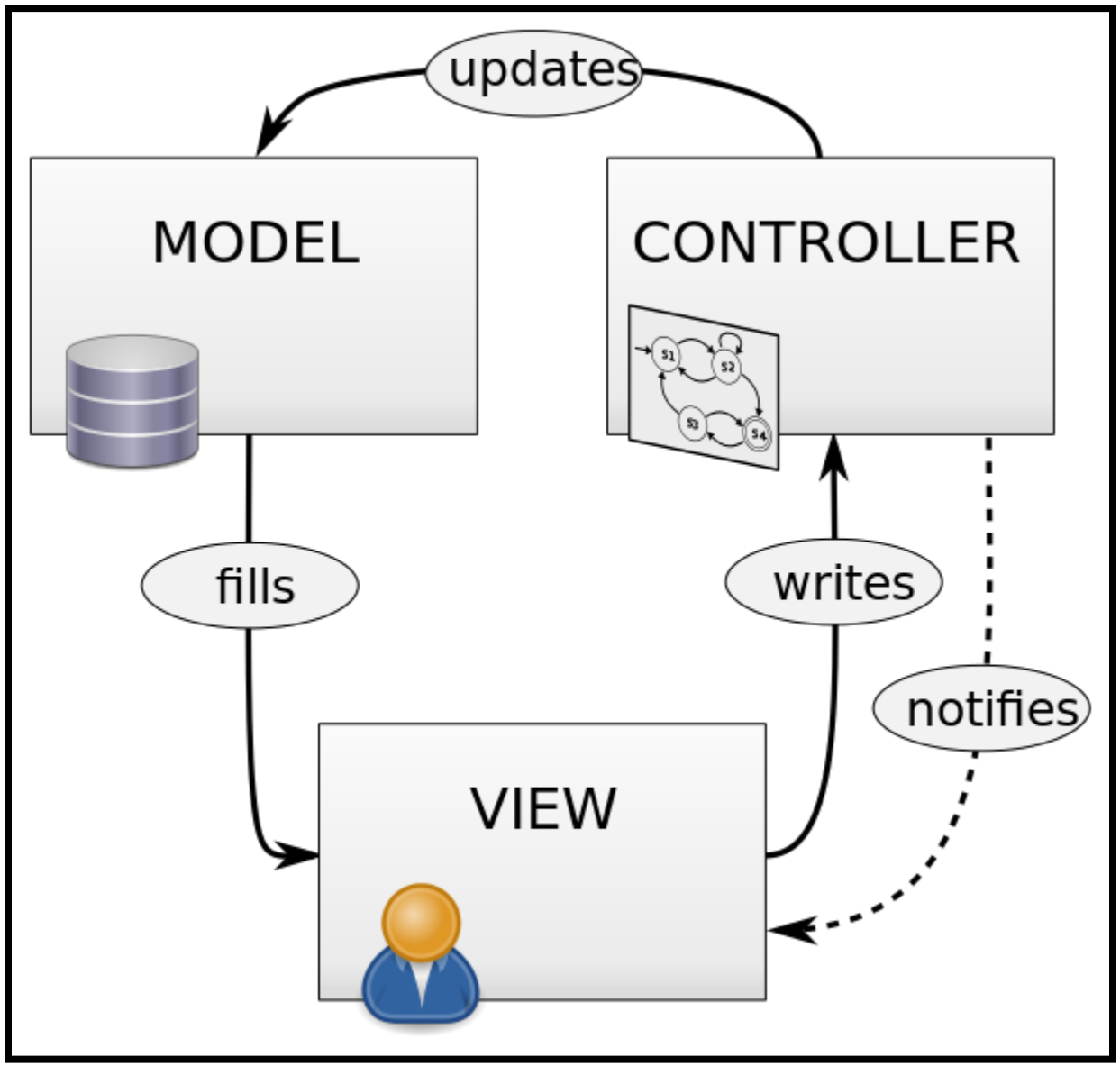
Model View Controller (MVC)

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

Teilnehmer: Model

- Das Modell kapselt Kerndaten und Funktionalität.
- Das Modell ist unabhängig von einer bestimmten Darstellung der Ausgabe oder einem bestimmten Verhalten der Eingabe.
- Das Modell bildet die Kernfunktionalität der Anwendung ab.
- (Das Modell benachrichtigt registrierte bei Datenänderungen.)

Model View Controller (MVC)

Teilnehmer: View

- Die Sicht (view) zeigt dem Benutzer Informationen an.
- Es kann mehrere Sichten pro Modell geben.
- Ggf. zugeordnete Eingabeelemente anzeigen

Model View Controller (MVC)

Teilnehmer: Controller

- Der Controller verarbeitet Eingaben und ruft passende Dienste der zugeordeten Sicht oder des Modells auf.
- Jede Controller ist einer Sicht zugeordnet
- Es kann mehrere Controller pro Modell geben.

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

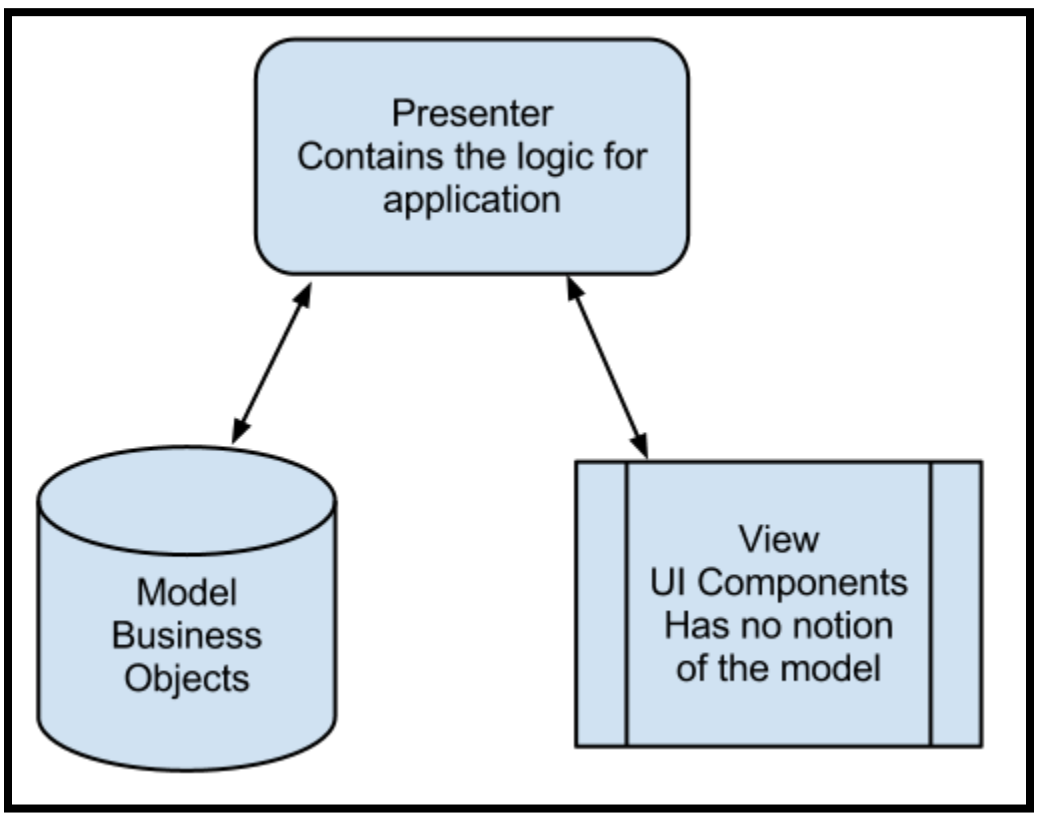
Hervorgegangen aus dem Model-View-Controller (MVC) Architekturmuster.

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)

Teilnehmer: Model

- Stellt die Logik der Ansicht dar (dies kann auch die Geschäftslogik sein)
- Über das Modell muss alle Funktionalität erreichbar sein, um die Views betreiben zu können.
- Die Steuerung des Modells erfolgt allein vom Presenter
- Das Modell selbst kennt weder die View, noch den Presenter

Model View Presenter (MVP)

Teilnehmer: View

- Enthält keinerlei steuernde Logik
- Allein für die Darstellung und die Ein- und Ausgaben zuständig
- Erhält weder Zugriff auf die Funktionalität des Presenters, noch auf das Model
- Sämtliche Steuerung der View erfolgt vom Presenter

Model View Presenter (MVP)

Teilnehmer: Presenter

- Bindeglied zwischen Modell und Ansicht
- Steuert die logischen Abläufe zwischen den beiden anderen Schichten
- Sorgt dafür, dass die Ansicht ihre Funktionalität erfüllen kann.

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 (MVC)

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 (MVC)

Nachteile

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

Interaktive Systeme

Model View Controller (MVC)

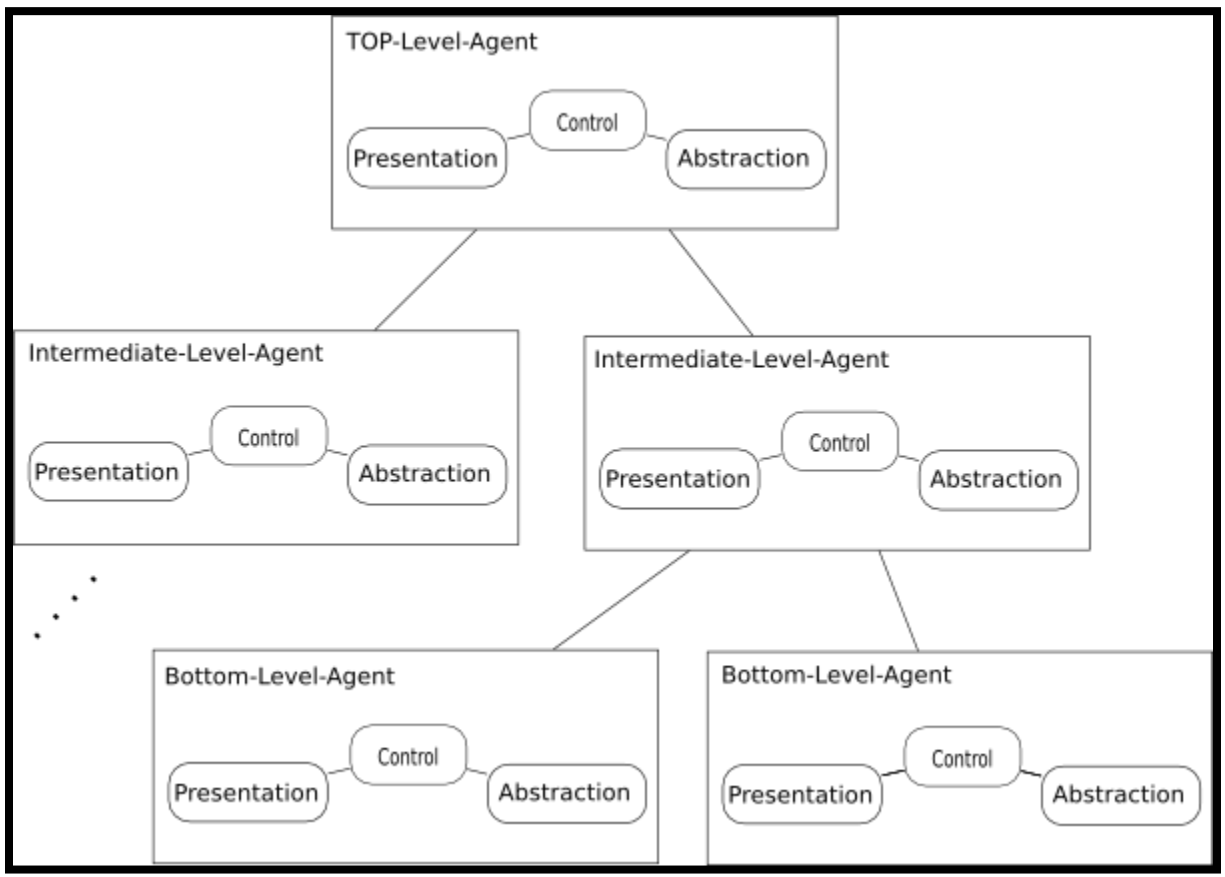
Model View Presenter

Presentation-Abstraction-Control (PAC)

Presentation-Abstraction-Control (PAC)

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

Presentation-Abstraction-Control (PAC)



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)

Top-Level-Agent

- Kommt nur einmal in einem System vor
- Übernimmt alle globalen Aufgaben, wie z. B. den Datenbankzugriff

Presentation-Abstraction-Control (PAC)

Intermediate-Level-Agent

- Bieten die eigentlichen Funktionen des interaktiven Systems an
- Eigene, möglichst abgeschlossene, Funktion
- Keine Abhängigkeiten zu anderen Bottom-Level-Agenten

Presentation-Abstraction-Control (PAC)

Bottom-Level-Agent

- Bilden die Schnittstelle zwischen der untersten (Bottom-Level) und der obersten (Top-Level) Schicht
- Fassen mehrere Bottom-Level-Agenten zu einem Teilsystem zusammen
- Teilsysteme auch weiter hierarchisch aufgeteilt werden
- Ein Teilsystem kann aus einem oder mehreren anderen Teilsystemen bestehen

Presentation-Abstraction-Control (PAC)

Architekturentwurf

- Aufteilung der geforderten Funktionalität auf mehrere Bottom-Level-Agenten
- Daraus: Top-Level-Agent definieren
- Intermediate-Level-Agenten definieren, die dann die Hierarchie bilden

Presentation-Abstraction-Control (PAC)

Aufteilung der Agenten

- Jeder einzelne Agent wird in die drei Komponenten strukturiert.
- --> Presentation: Graphische Benutzeroberfläche, komplette Ein- und Ausgabe
- --> Abstraction: Realisiert das Datenmodell des Agenten realisiert
- -->Control: Stellt die Verbindung zwischen den beiden anderen Komponenten her und ermöglicht die Kommunikation mit anderen Agenten. Die 'Controls' sind damit die zentrale Schnittstelle, die die Zusammenarbeit der einzelnen Teile eines PAC-Systems ermöglichen.

Presentation-Abstraction-Control (PAC)

Aufteilung der Agenten

- Nicht jeder Agent muss alle drei Komponenten implementieren
- Jeder Agent bringt die Benutzerschnittstelle und das Datenmodell für seine Aufgabe mit
- Jeder Agent muss die Control implementieren, um über sie die Kommunikation mit anderen Agenten und zwischen den Komponenten zu ermöglichen

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?

Unterlagen: ai2016.nils-loewe.de