Softwareengineering in der Praxis

Am Beispiel der SaaS-CRM-Lösung CAS PIA

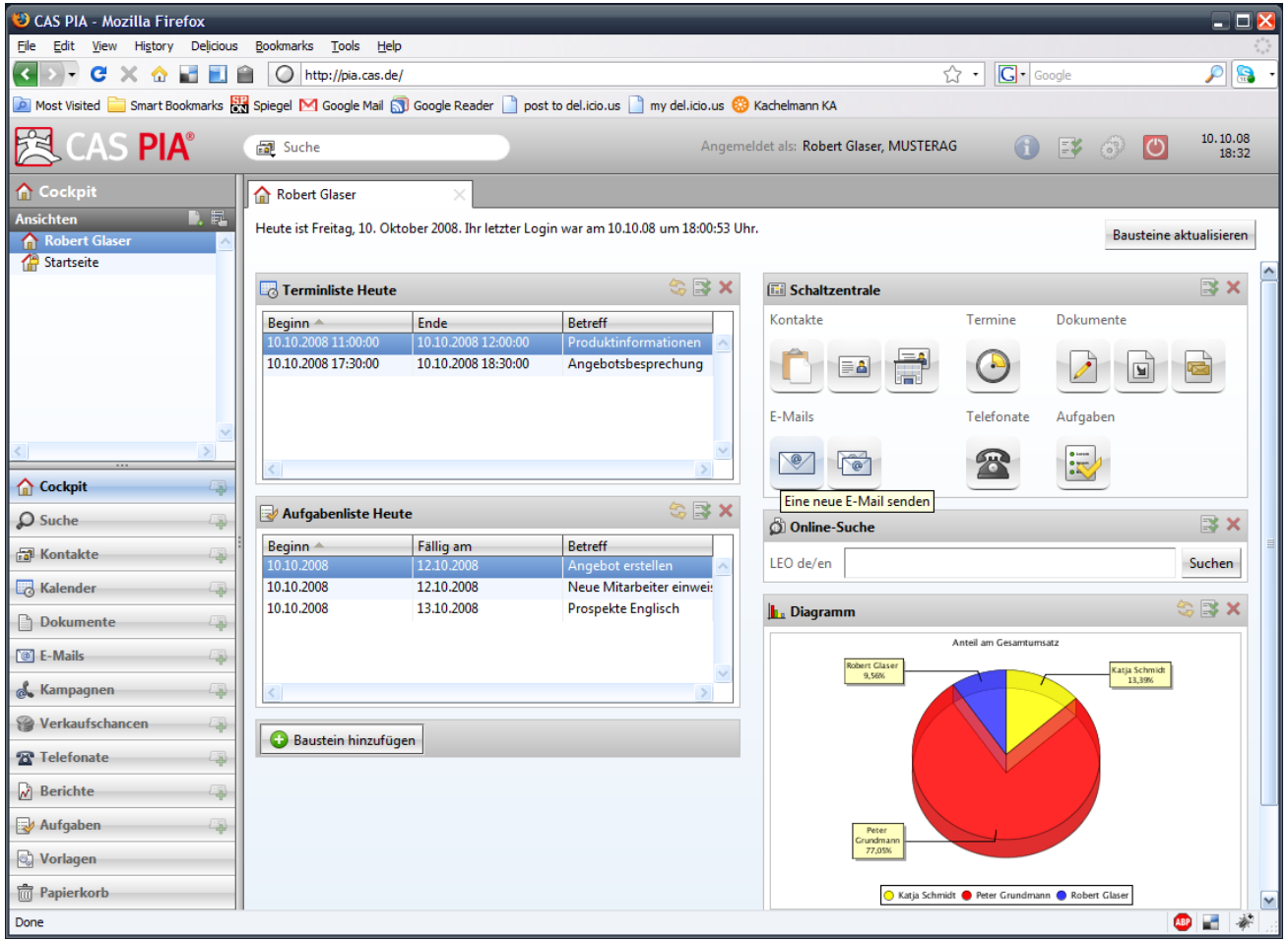
Dr. Markus Bauer
Leiter Entwicklung CAS PIA und CAS Open

31. Januar 2011

Agenda

- Einführung
- Architektur
  - Grundprinzipien
  - Muster
- Entwicklungsprozess
  - Scrum
  - Erkenntnisse aus der Praxis
- Softwarequalität

CAS PIA


CAS PIA - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://pia.cas.de/
Most Visited Smart Bookmarks Spiegel Google Mail Google Reader post to del.icio.us my del.icio.us Kachelmann KA
CAS PIA
Suche
Angemeldet als: Robert Glaser, MUSTERAG
10.10.08 18:32
Cockpit
Ansichten
Robert Glaser
Startseite
Robert Glaser
Heute ist Freitag, 10. Oktober 2008. Ihr letzter Login war am 10.10.08 um 18:00:53 Uhr.
Bausteine aktualisieren
Terminliste Heute
Beginn Ende Betreff
10.10.2008 11:00:00 10.10.2008 12:00:00 Produktinformationen
10.10.2008 17:30:00 10.10.2008 18:30:00 Angebotsbesprechung
Schaltzentrale
Kontakte Termine Dokumente
E-Mails Telefonate Aufgaben
Eine neue E-Mail senden
Aufgabenliste Heute
Beginn Fällig am Betreff
10.10.2008 12.10.2008 Angebot erstellen
10.10.2008 12.10.2008 Neue Mitarbeiter einwei
10.10.2008 13.10.2008 Prospekte Englisch
Online-Suche
LEO de/en
Suchen
Diagramm
Anteil am Gesamtumsatz
Robert Glaser 9,56%
Katja Schmidt 13,39%
Peter Grundmann 77,05%
Katja Schmidt Peter Grundmann Robert Glaser
Baustein hinzufügen
Cockpit
Suche
Kontakte
Kalender
Dokumente
E-Mails
Kampagnen
Verkaufschancen
Telefonate
Berichte
Aufgaben
Vorlagen
Papierkorb
Done

Kontext: Standardsoftware für CRM

Kontext: Software-as-a-Service (SaaS)

- Software as a Service:
  - Software wird nicht mehr beim Kunden installiert, sondern zentral beim Anbieter installiert und betrieben
  - Die Kunden verwenden die Software im Browser über's Internet
  - Statt Anschaffungs- und Betriebskosten: monatliche Miete als all-inclusive Angebot, jederzeit kündbar

- Vorteile für die Kunden:
  - Geringere TCO: keine Anschaffungskosten für Hardware / Software (Lizenzen) , keine Wartungsaufwände, keine Aufwände für Backup, Update,…
  - Keine Bedarfsplanung hinsichtlichh Skalierbarkeit/Nutzerzahlen nötig

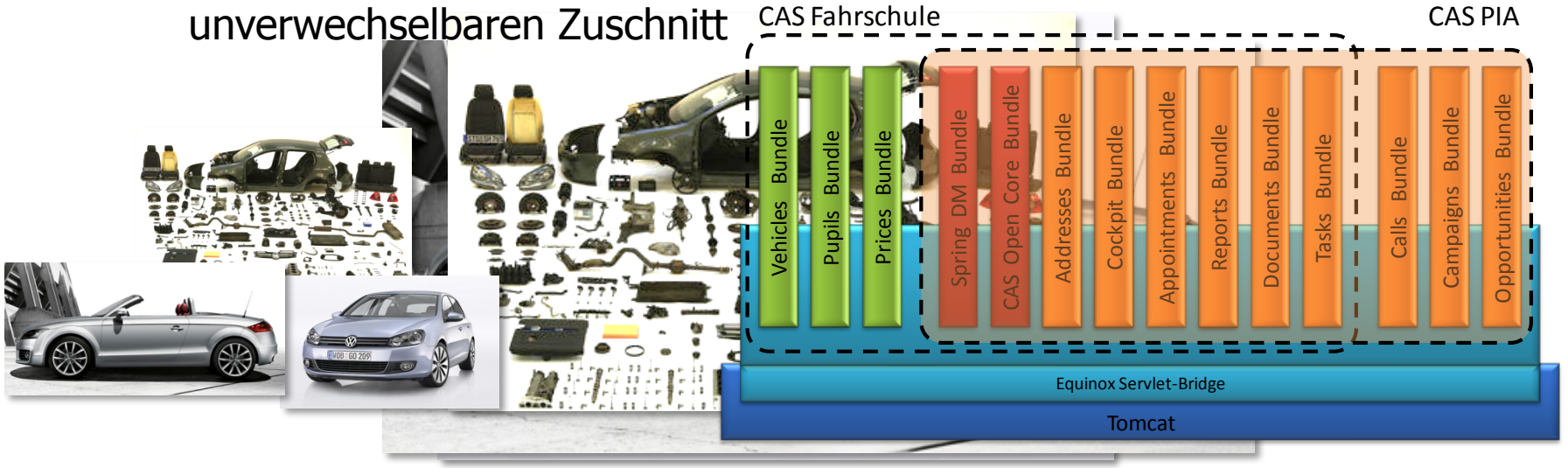
- Vorteile für den Anbieter/Hersteller:
  - Vergrößerung des Marktes (selling to the long tail)
  - Stete Einnahmen (insbesondere im Mietmodell)
  - Geringere Kosten für Support (alle Nutzer verwenden dieselbe Installation, Version, Unterstützung und Bugfixing alter Releases entfällt)

Kontext: SaaS -- Herausforderungen

- Zielkonflikt:
  - Economy of scale: eine gemeinsame Lösung (Codebasis, Installation) für möglichst viele Kunden
  - Anpassbarkeit: Perfekter Zuschnitt auf die individuellen Bedürfnisse jedes Kunden
- Perfekter, störungsfreier Betrieb 24/7:
  - Verfügbarkeit, Sicherheit, Skalierbarkeit, Performance, Wartbarkeit...

Kontext: Plattformgedanke

- Ziel: Kostengünstige Entwicklung von Informationssystemen
- Vorgehen wie beim Automobilbau:
  - Kundensicht: verschiedenste Produkte mit genau auf die Kundenwünsche zugeschnittenen Eigenschaften
  - Entwicklungssicht: einheitliche Basisplattform und gemeinsam genutzte Komponenten
  - Wenige individuelle Komponenten geben dem Produkt seinen unverwechselbaren Zuschnitt



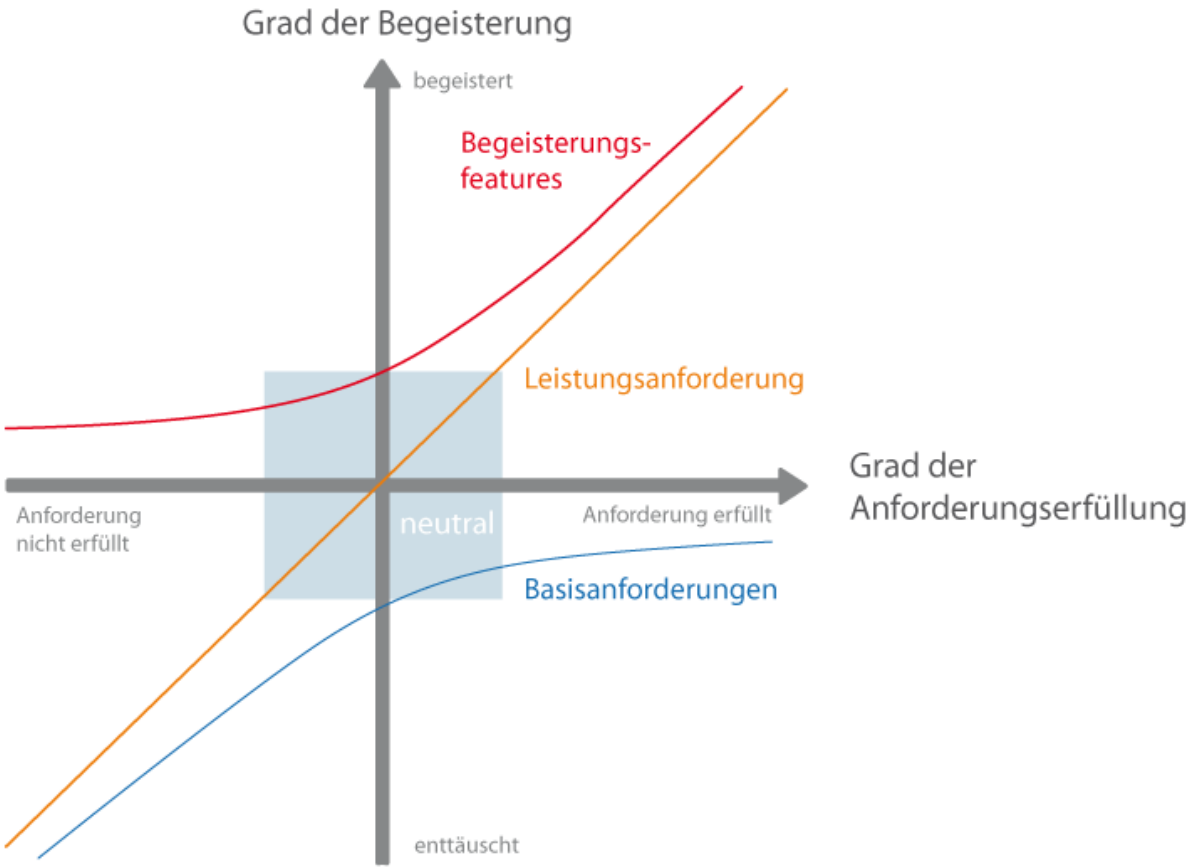
Anforderungen

- Leichte Bedienbarkeit, attraktive GUI
- CRM „hüpft aus dem Päckle“

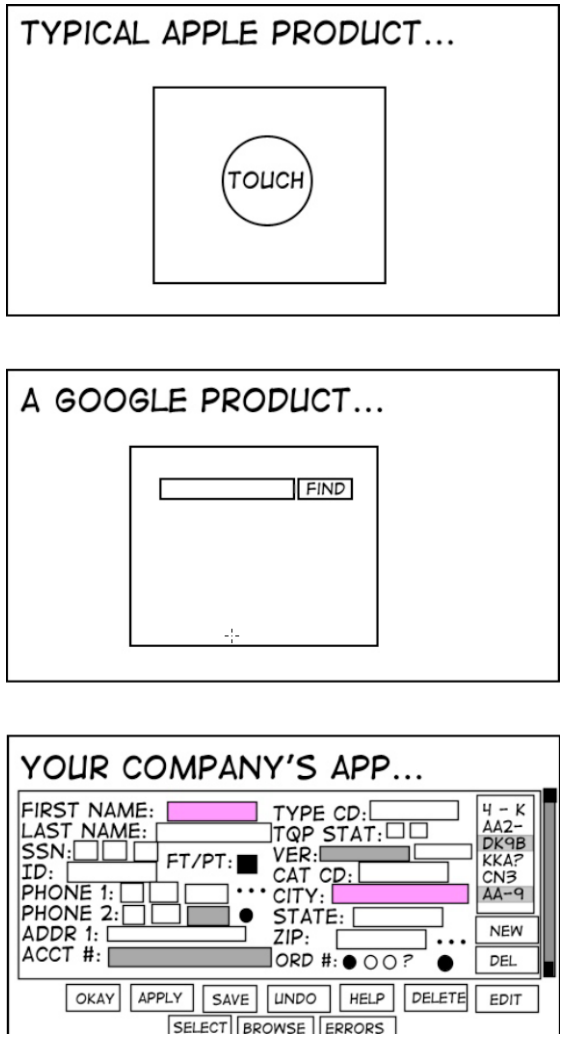
- Solide Groupwarefunktionen
- Starke CRM-Funktionen, zugeschnitten auf die Zielgruppe
- Gute Vorkonfigurationen
- Umfangreiche Anpassbarkeit (im Rahmen von Projekten)
- Integration von Mehrwertdiensten (aus der CAS-Gruppe, später auch über Partner)

- Hohe Performance, hohe Skalierbarkeit, hohe Sicherheit
- Hohe Qualität, Robustheit

Exkurs: das Kano-Modell



Exkurs: User Centric Design

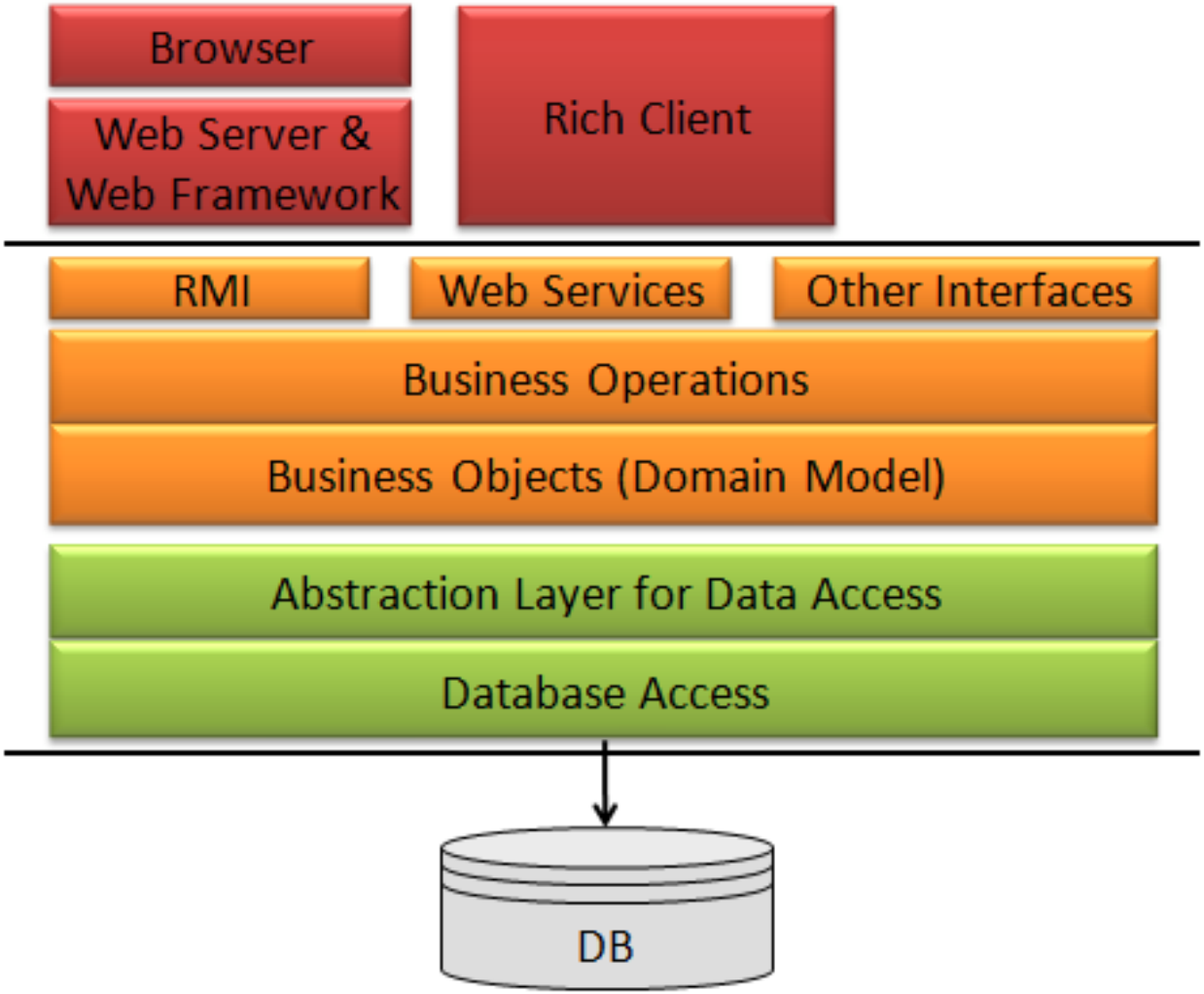


- Prozess der bewussten, planmäßigen Gestaltung
- Merkmale:
  - Rückt den Menschen in den Vordergrund
  - Ausrichtung auf Personas und dazu passende User Stories
  - Prototyping: Nutzen von User-Feedback

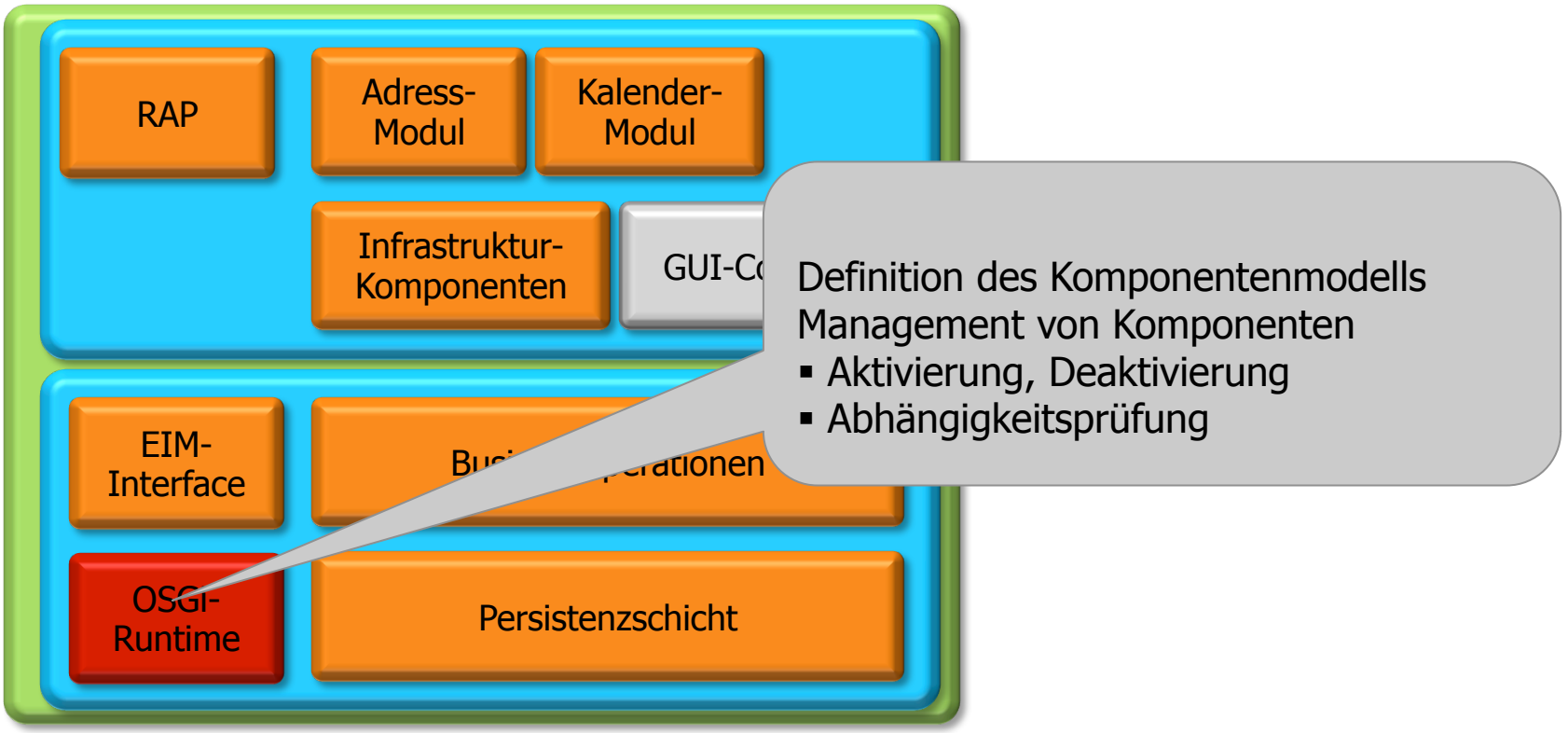
Architektur

- Ergebnis des Systementwurfs
- Definition der Bauteile eines Systems und ihrer Zusammenhänge
  - Statische Sicht: Teilsysteme, Bauteile,…
  - Dynamische Sicht: Zusammenspiel der Bauteile: Schnittstellen und Protokolle
  - Deployment- und Laufzeitsicht
  - …
- Ziele einer guten Architektur:
  - Übersichtlichkeit des Systementwurfs
  - Arbeitsteilung bei der Entwicklung
  - Kapseln von Entwurfsentscheidungen
  - Wiederverwendbarkeit des Systems bzw. seiner Bauteile
  - Verteilter Betrieb, Flexibilität bei Auslieferung und Installation

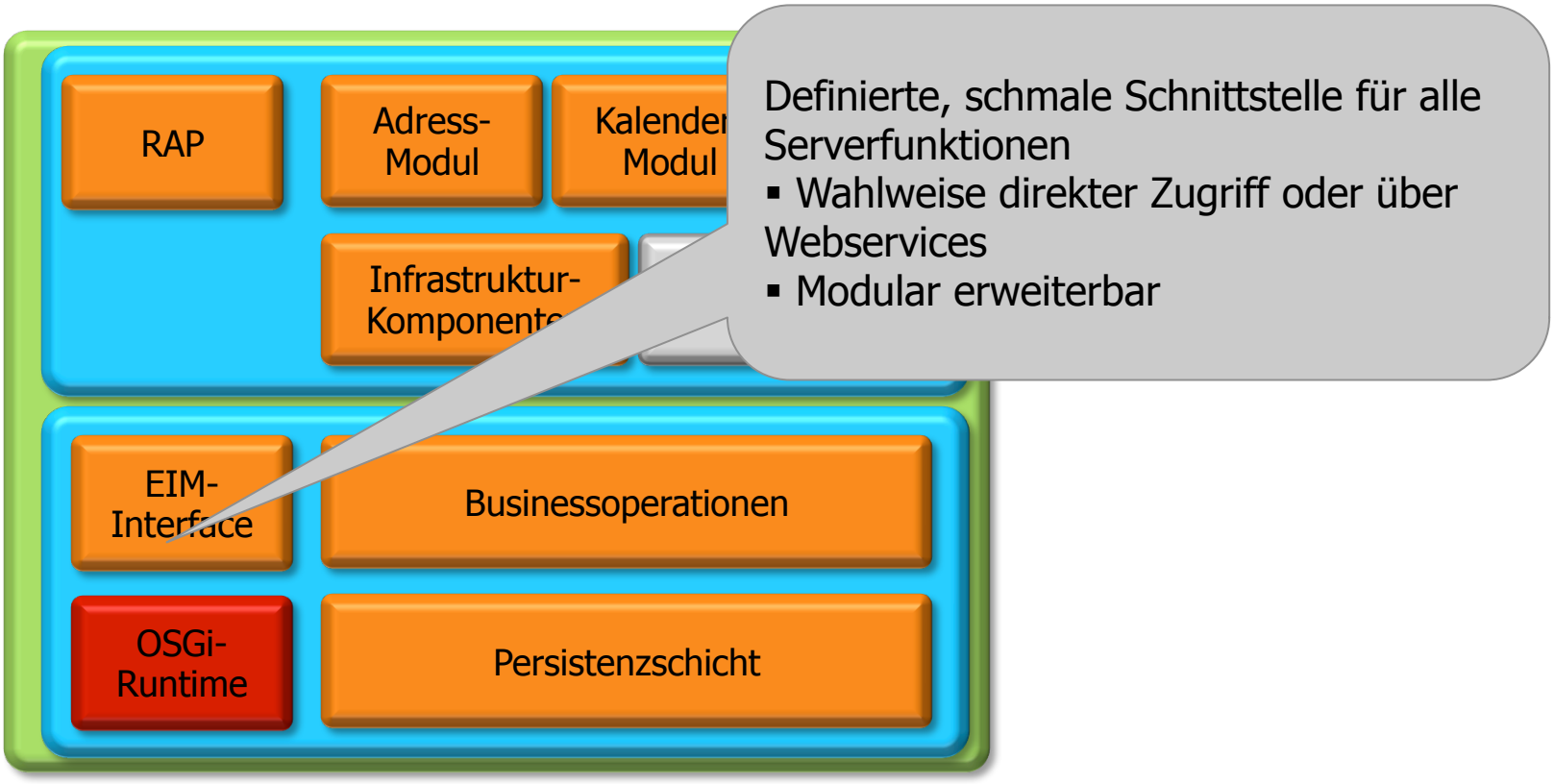
CAS PIA: Statische Architektur



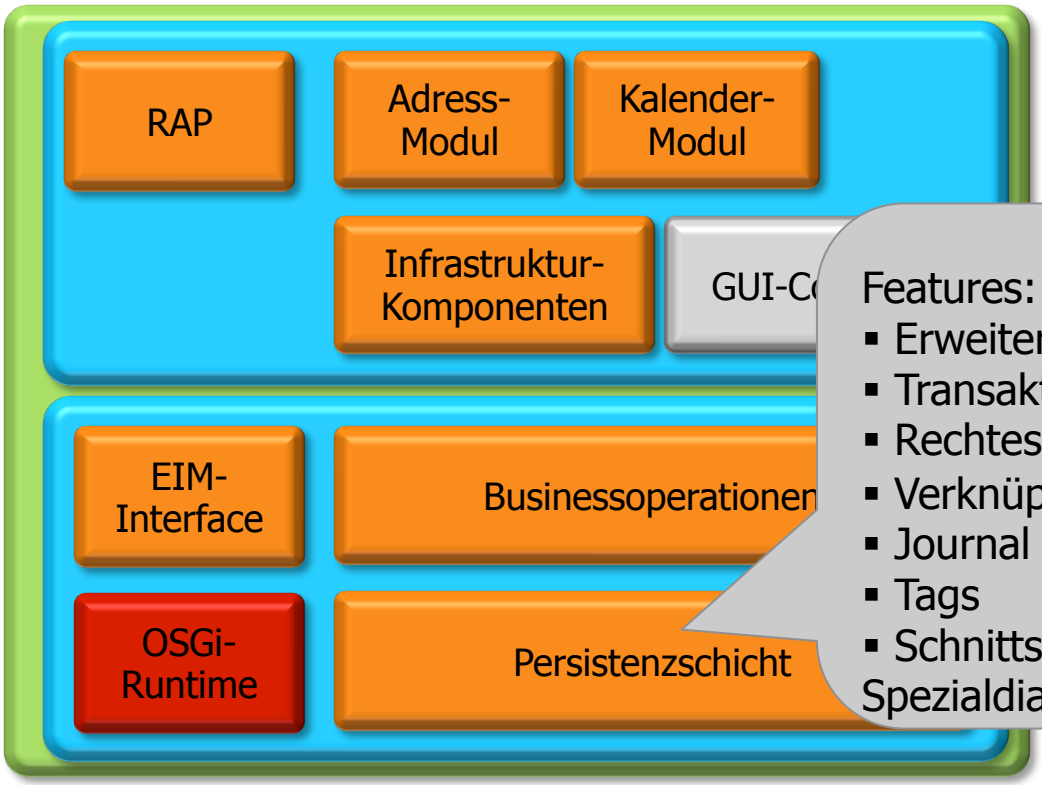
CAS PIA:
Statische Architektur mit Funktionszuordnung



CAS PIA:
Statische Architektur mit Funktionszuordnung

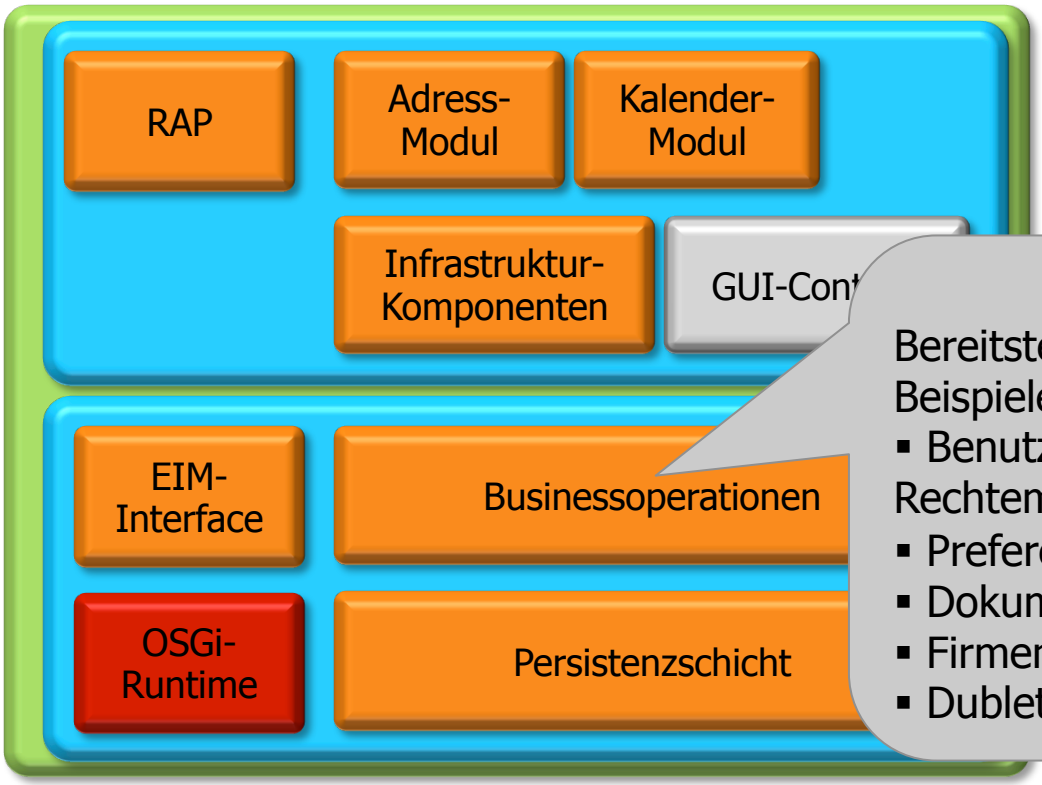


CAS PIA:
Statische Architektur mit Funktionszuordnung



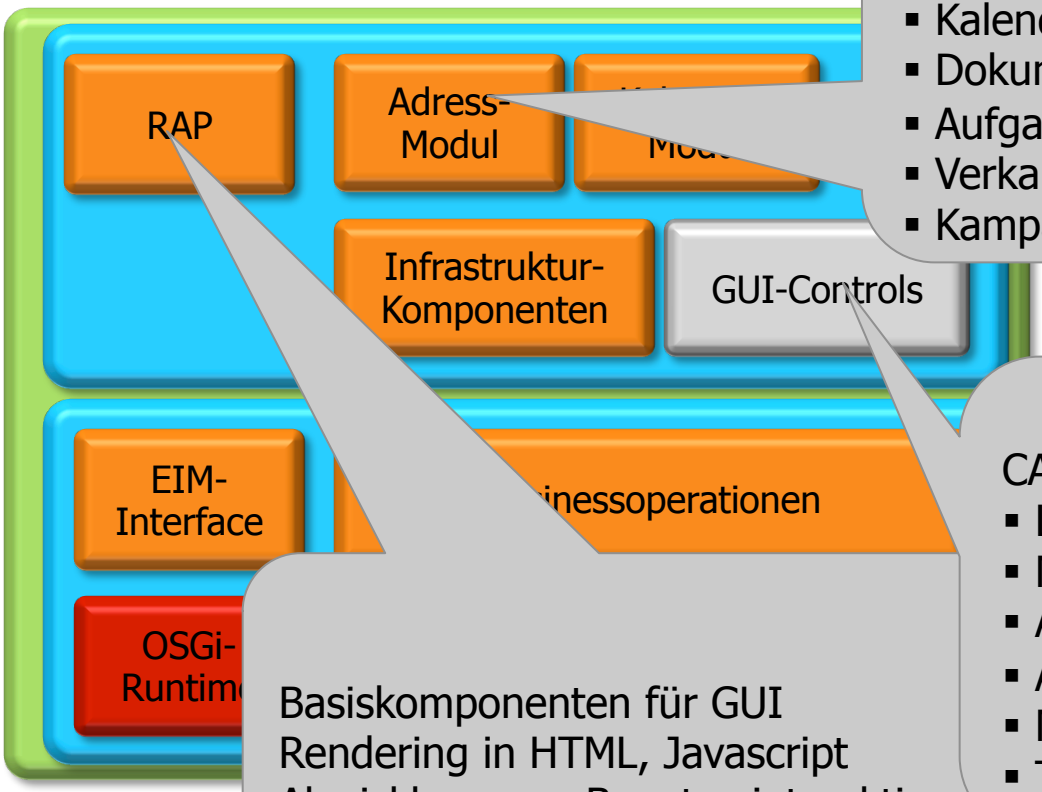
Features:
- Erweiterbares Objektmodell
- Transaktionen
- Rechtesystem
- Verknüpfungen
- Journal
- Tags
- Schnittstelle: definierter SQL-Spezialdialekt

CAS PIA:
Statische Architektur mit Funktionszuordnung



Bereitstellung der Anwendungslogik
Beispiele:
- Benutzer-, Gruppen- und Rechtemanipulation
- Preferences
- Dokumentenverwaltung
- Firmenbildung
- Dublettensuche

CAS PIA:
Statische Architektur mit Funktionszuordnung

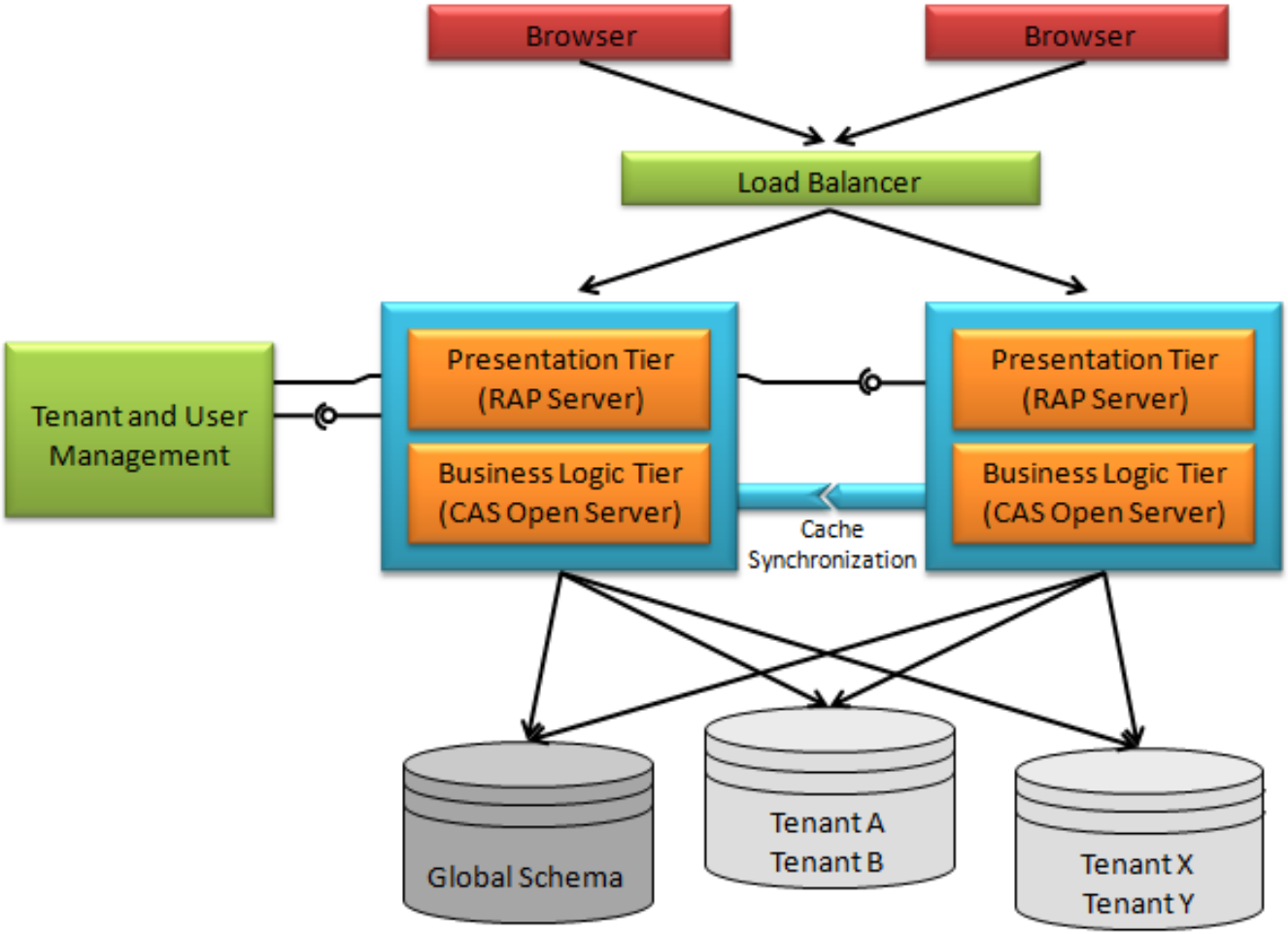


Kapselung von Funktionsbereichen
- Adressen
- Kalender
- Dokumente
- Aufgaben
- Verkaufschancen
- Kampagnen

CAS PIA-spezifische Komponenten
- Listen, gemischte Listen
- Masken
- Akte
- Anwendungsrahmen
- Navigator
- Toolbar

Basiskomponenten für GUI
Rendering in HTML, Javascript
Abwicklung von Benutzerinteraktionen

CAS PIA: Deploymentarchitektur



PIA - Browserclient
Firefox, Safari, IE8

CAS Open - Server
Linux +
Apache Tomcat
Java + Spring + RAP

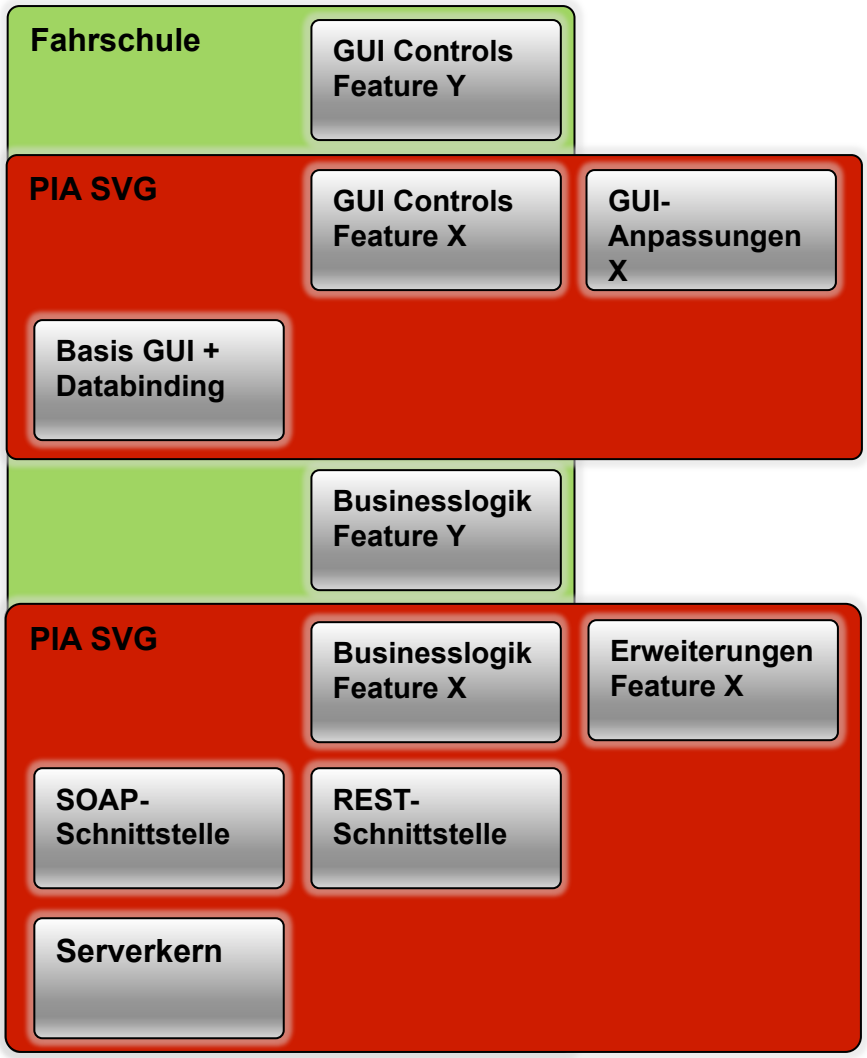
DB-Server
Linux + MySQL

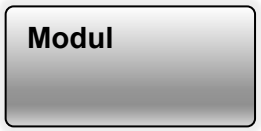
Technische Architektur

- Festlegung der technischen Infrastruktur und Standards
  - Ablaufumgebungen für die einzelnen Bauteile der Architektur
  - Schnittstellen und Protokolle zwischen den Bauteilen

- Beispiel: CAS PIA bzw. CAS Open
  - Ablaufumgebungen:
    - Standard PC-Server mit **Linux**
    - **MySQL** als Datenbank
    - **Java VM:** virtuelle „Hardware" als universelle Ablaufumgebung
    - **Apache Tomcat:** Container für den Anwendungsserver
    - **OSGi:** leichtgewichtiges, weitverbreitetes Framework zur Modularisierung
    - **Spring-DM: leichtgewichtiges Framework für IoC bzw. Dependency Injection**
    - **Spring-Security:** Framework für Authentifizierung und Security
    - **Eclipse RAP:** Präsentationsschicht für RIA-Anwendungen
    - **JSF bzw. Spring/MVC:** Präsentationsschicht für seitenorientierte Anwendungen (Portale, mobiler Zugriff)
  - Protokolle (Auszug):
    - **Asynchrone XML-HTTP-Requests über SSL** zwischen Browser und Anwendungsservern
    - Wahlweise **SOAP-Services, RMI** oder direkte Java-Aufrufe zwischen Präsentationsschicht und Serverkern
    - **UDP** zur Kommunikation zwischen Anwendungsservern
    - **REST-Services** zur leichtgewichtigen Datenversorgung

Modulkonzept und technische Ablaufumgebung





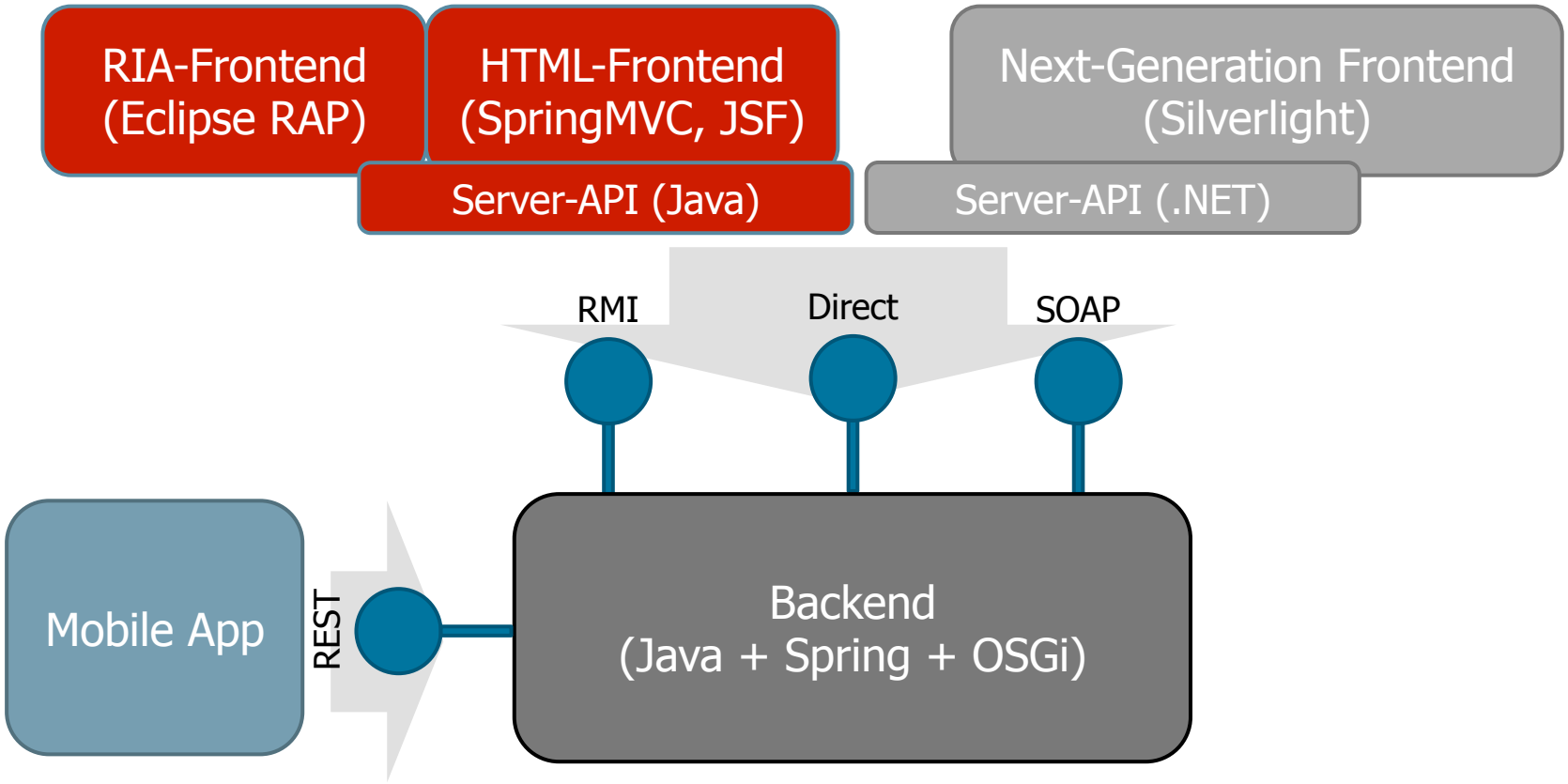
Eigenschaften:

- Definierte, versionierte Schnittstelle (inkl. Abhängigkeiten zu anderen Plugins)
- Flexibles Zusammenbauen von Modulen zu vollständigen Lösungen

Technische Lösung:

- OSGi: Etabliertes, bewährtes Konzept für Erweiterungen einer Basisplattform – Eclipse-Umgebung ist Referenzstudie

Technische Architektur -- Schnittstellen

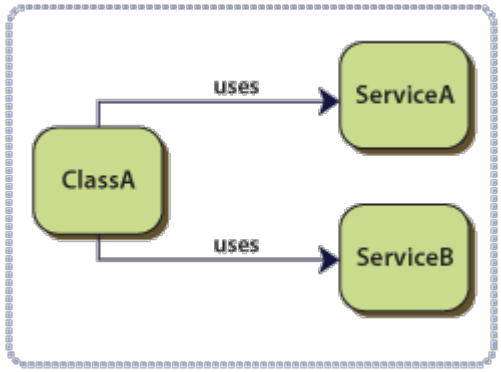


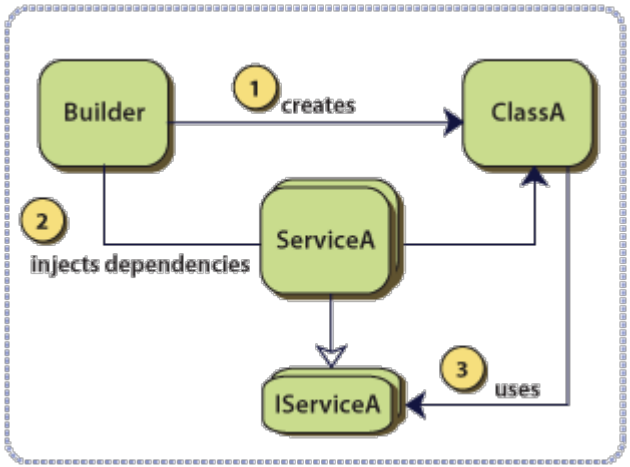
Verwenden von Mustern

- Ziel:
  - Ingenieurmäßiges Vorgehen: Verwenden bewährter, bekannter Lösungen für Standardprobleme
  - Klare Strukturierung: Entwickler finden sich sofort in Architektur zurecht
  - Vorteile und Nachteile der Lösungen sind bekannt
- Muster existieren auf allen Ebenen
  - Architekturmuster/-stile: Schichtenarchitektur, Dependency Injection
  - Klassische Entwurfsmuster, technologiespezifische Muster: Façade, DAO-Muster, Command-Muster
  - Implementierungsmuster

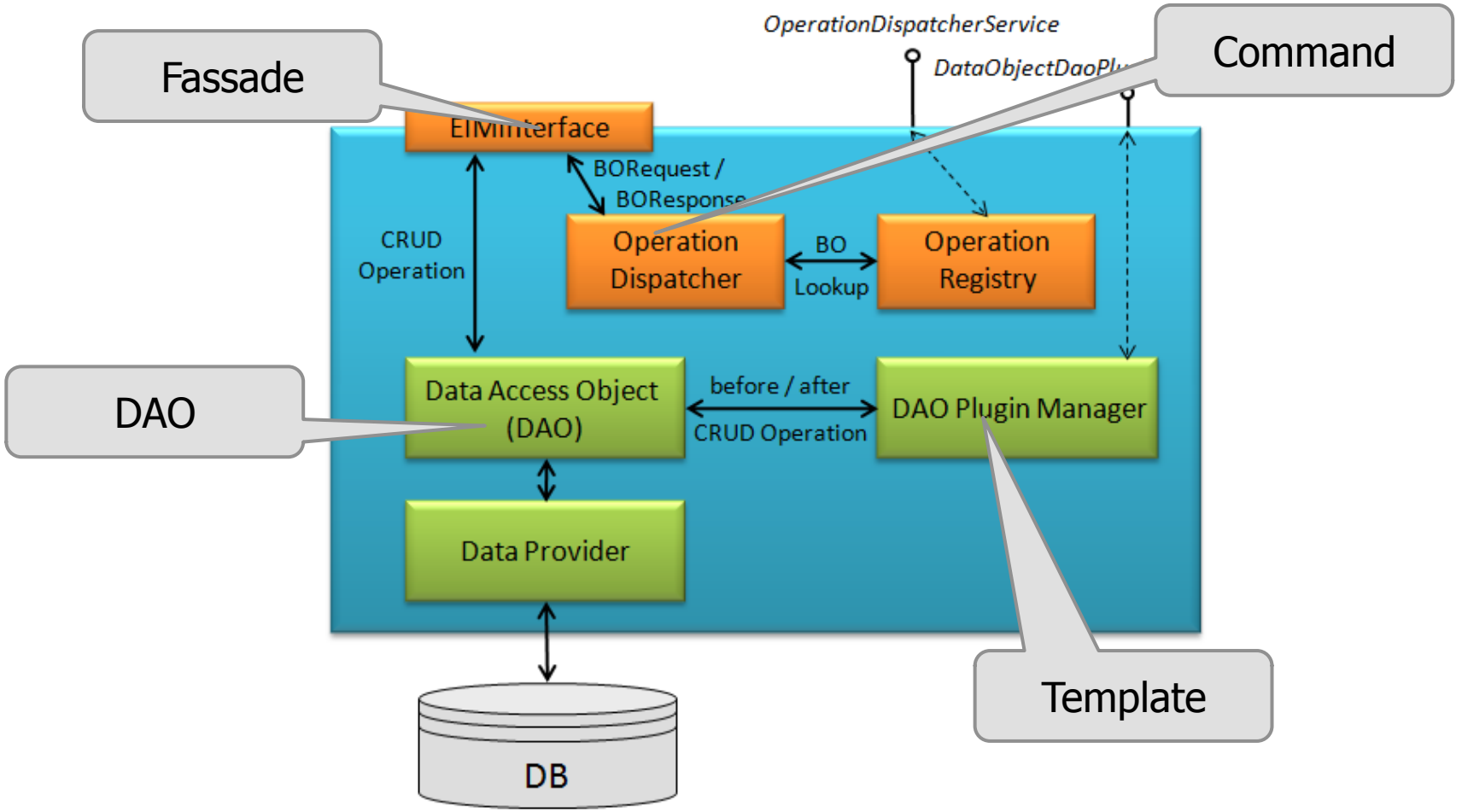
Architekturmechanismen
Inversion of Control, Dependency Injection

- Ziel:
  - Konfiguration von Varianten
  - Auflösen von Abhängigkeiten zur Laufzeit
  - Delegation der Auswahl einer bestimmten Implementierung an eine externe Einheit
- Beispiel:
  - Je nach Konfiguration soll in Klasse A entweder Service A und B oder Service A1 und B1 verwendet werden
  - Builder kontrolliert Objekterzeugung und Objektauswahl, indem er passende Instanzen in die Klasse A injiziert, in der Regel über Konfigurationsmechanismus
  - Klasse A nützt nur noch abstrakte Services A und B



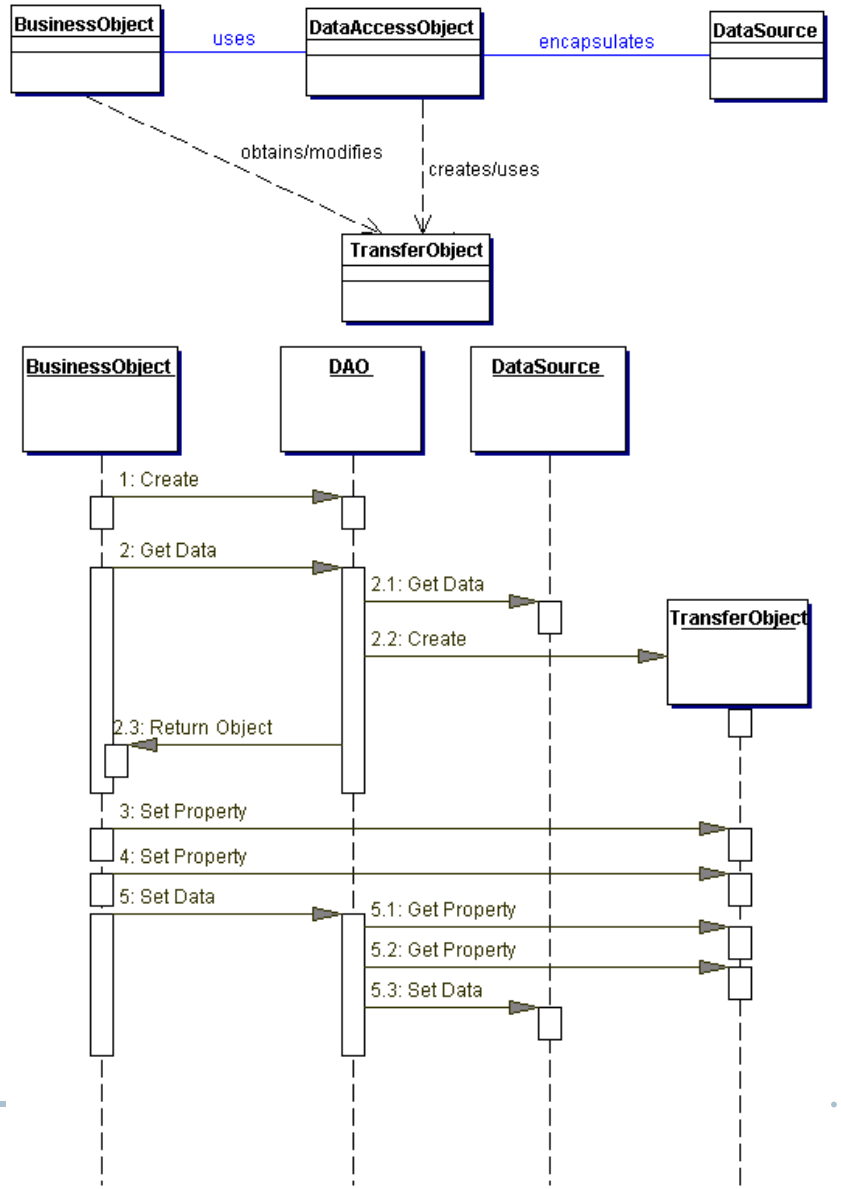


Serverkern: innerer Aufbau



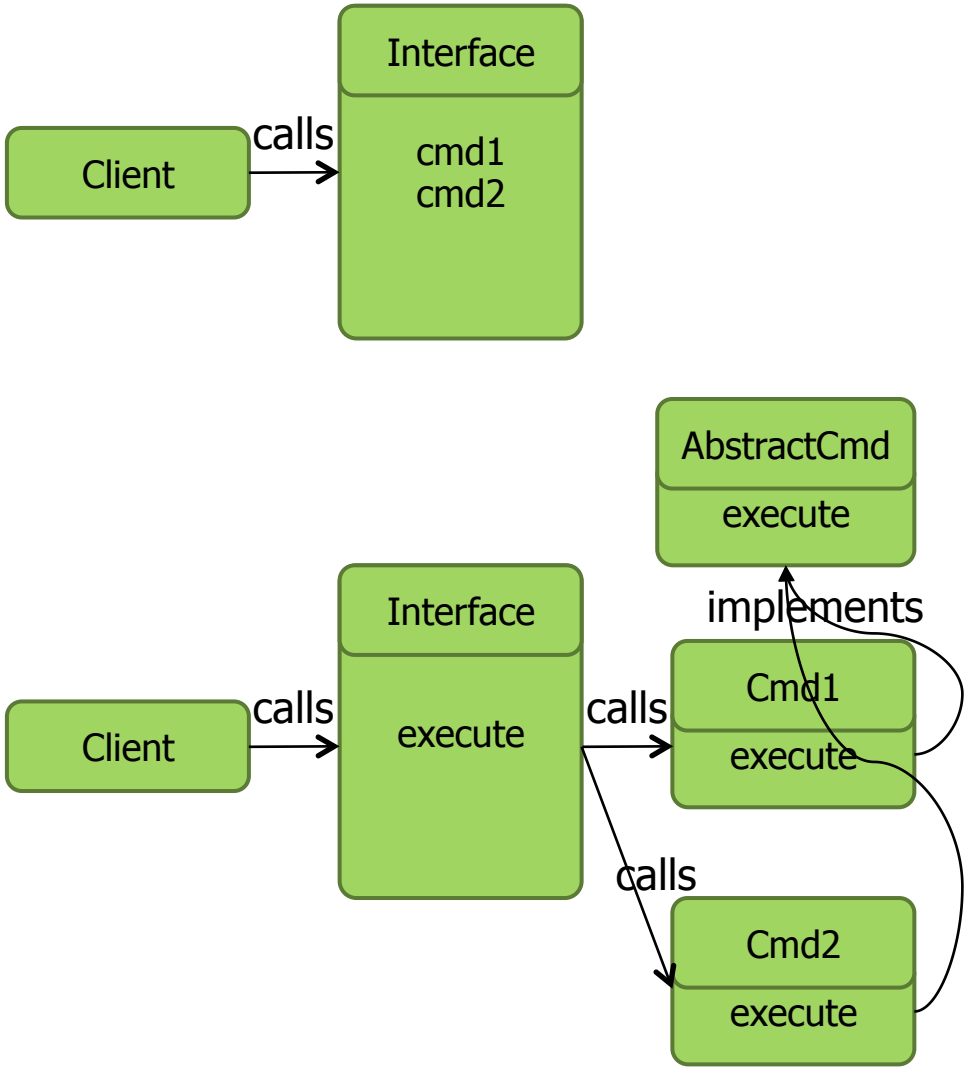
Architekturdetail: DAO-Muster

- Ziele:
  - einheitliche API für CRUD-Operationen
  - Entkoppeln von Datenobjekten (DTOs) von Serverlogik
- Vorteile:
  - Klare Zuständigkeiten
  - DataSource z.B. für Testbetrieb austauschbar
  - Konkrete Implementierung kann in spezielle DAO-Varianten für unterschiedliche DataSources ausgelagert werden



Architekturdetail: Flexible Service-Schnittstelle

- Ziel:
  - Erweiterung der Schnittstelle um neue Kommandos, u.a. zur Deployzeit
  - Delegation der Kommandos in Kommandoobjekte
  - In der Praxis: Zusammenspiel mit Dependency Injection und Registry für verfügbare Kommandos



Erfahrungen

- SaaS-Lösung CAS PIA ist seit 1,5 Jahren 24/7-Betrieb in zwei Rechenzentren, (fast) keine Ausfälle, mehrere 1000 Kunden
- Grundvoraussetzungen:
  - hohe Qualität:
    - Agile Entwicklung (Scrum) mit regelmäßigen Releases
    - Ausrichtung des Entwicklungsprozesses auf Qualität
  - nahtlose Integration in „typischen Rechenzentrumsbetrieb"
    - Redundanz, Skalierfähigkeit (inkl. Lastverteilung) preisgünstig über Standardkomponenten
    - Installation über Standardmechanismen (Paketverwaltung bzw. Images)
    - Überwachung und Diagnose über Nagios
    - Passendes Know-How ist in fast jedem Rechenzentrum vorhanden
- Flexible Architekturen zahlen sich aus:
  die Basis von CAS PIA, CAS Open, ist heute Grundlage für verschiedenste Systeme
    - SaaS-Lösung PIA für viele kleine Unternehmen (viele kleine Kunden)
    - Branchenlösung für Versicherungen (wenige, sehr große, verteilt arbeitende Kunden)
    - Portallösung für Technologieprojekte (viele Nutzer)

Agile Entwicklung mit Scrum

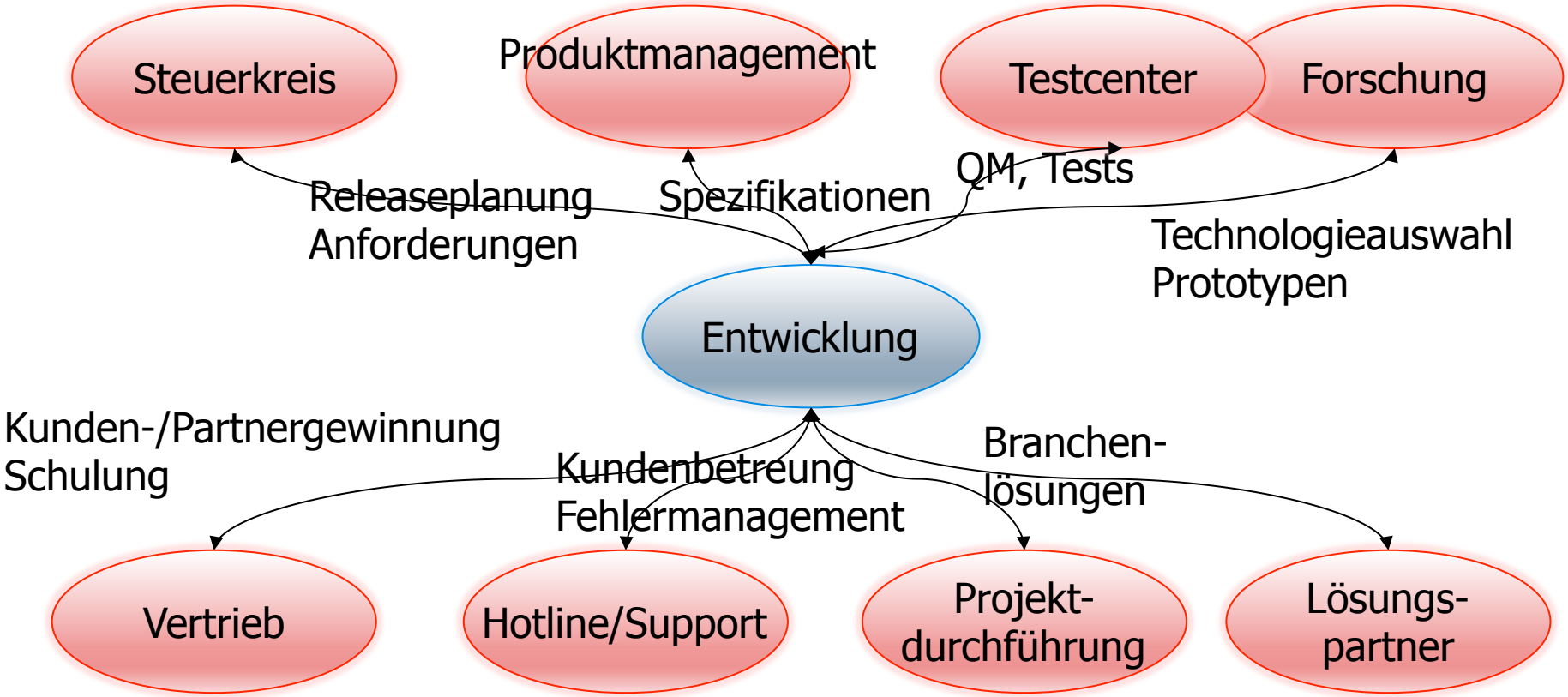
- Grundideen:
  - Time-Boxing
  - Fokus auf Personen und Kommunikation
  - Selbstorganisierende Teams
  - Radikale Vereinfachung der Entwicklungsziele und des Prozesses
  - Subversive Psychologie ☺
  - Planung ist wichtig, aber noch wichtiger ist Flexibilität!

Wiederholung: Schlüsselkonzepte von Scrum

- Schlüsselkonzepte
  - Product Owner
  - Team
  - Scrum Master
  - Daily Scrum
  - Userstory
  - Product Backlog
  - Sprint
  - Sprint Backlog
  - Impediment Backlog

[Quelle: http://softhouseeducation.com]

Exkurs: Vernetzung der Produktentwicklung



Scrum bei CAS --Modifikationen

- Wer ist Product Owner?
  - Steuerkreis, Produktmanagement, oder Entwicklungsleitung?
  - Steuerkreis ist Kunde(!) – bestimmt Priorisierung und Budget
  - Entwicklungsleitung ist Product Owner – vermittelt zwischen Kunde und Entwicklungsteam
  - Alternative: Produktmanagement ist Product Owner
- Einbindung von Produktmanagement und Testcenter
  - Produktmanagement ist Teil des Scrum-Teams
  - Testcenter ist Teil des Scrum-Teams
- Projektleiter
  - Erfahrene Mitarbeiter werden Projektleiter
  - Projektleiter: Stakeholder für Kundenprojekte oder Teilprodukte und arbeiten dem Product Owner zu
  - Product Owner ist nach wie vor einzige “Schnittstelle” zum Team

Scrum: Erfahrungen der CAS

- Vorgehensweise "jeder Sprint liefert eine (theoretisch) release-fähige Version" passt optimal zu SaaS
  - Häufige und qualitativ hochwertige Releases sind notwendig, um Kunden für SaaS-Prinzip zu begeistern!
- Agile Methoden sind hilfreich, garantieren aber keine erfolgreichen Projekte! Gute Leute sind der Schlüssel zum Erfolg!
  - Selbstorganisierende Teams erfordern Teammitglieder mit hohem technischem und sozialem Know-how .
- Vertrauen und Unterstützung von seitens des Managements sind unverzichtbar
  - Das Team muss Sprintziele selbst bestimmen dürfen; Festlegung der Ziele von Außen endet meist im „Deathmarch": Team liefert immer, aber immer mehr Murks
  - Das Team muss sich immer auf die Sprintziele konzentrieren können, neue Anforderungen müssen bis zum nächsten Sprint warten – Ausnahme: Fehler!
- Scrum fördert den Spass an der Arbeit!
  - Scrum betont die (Eigen-)Verantwortung des Teams
  - Übernahme von Verantwortung wirkt extrem motivierend
- Solide Softwareentwicklung und Know-How sind immer noch notwendig! ☺
- Scrum ist besonders erfolgreich, wenn es mit einer guten Qualitätssicherung verknüpft wird!

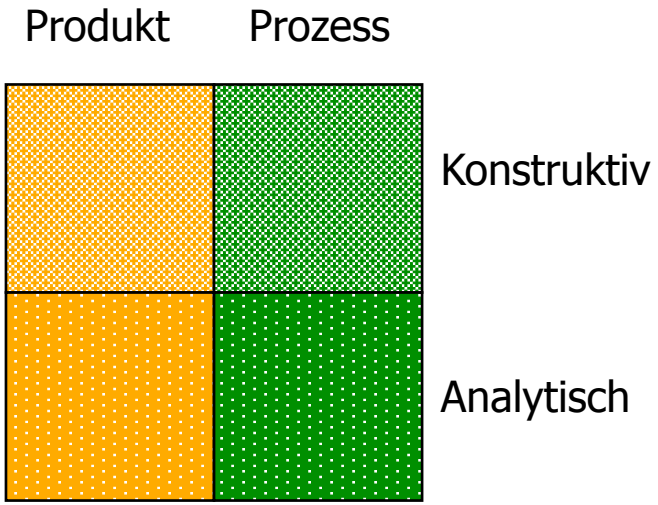
Softwarequalität – was ist das?

„Qualität ist, wenn der Kunde wieder kommt,
und nicht die Ware."

- Externe Qualität = Kundenperspektive:
  Einfache Verwendbarkeit, Performance, Robustheit, ...
- Interne (Software-)Qualität = Entwicklerperspektive:
  Flexibilität, Verständlichkeit, Wartbarkeit, ...
- Hypothese: Gute interne Qualität ist eine Voraussetzung für gute externe Qualität!

Softwarequalität

- Produktbezogene Sicht
  - Konstruktive Maßnahmen:
    Architekturen, Methoden, Sprachen, Standards, Werkzeuge, Muster und Richtlinien zur effizienten Konstruktion hochwertiger Systeme
    Testverfahren (Unit-Tests), Lasttests
  - Analytische Maßnahmen:
    Methoden und Werkzeuge zur Bewertung der Produktqualität
- Prozessbezogene Sicht
  - Konstruktiv: Verwendung eines Softwareprozesses, der wiederholbar Produkte mit hoher Qualität hervorbringt, SPI-Maßnahmen
  - Analytische Maßnahmen zur Kontrolle und Bewertung der Prozessqualität
- Innere Qualität ist Teil der produktbezogenen Sicht

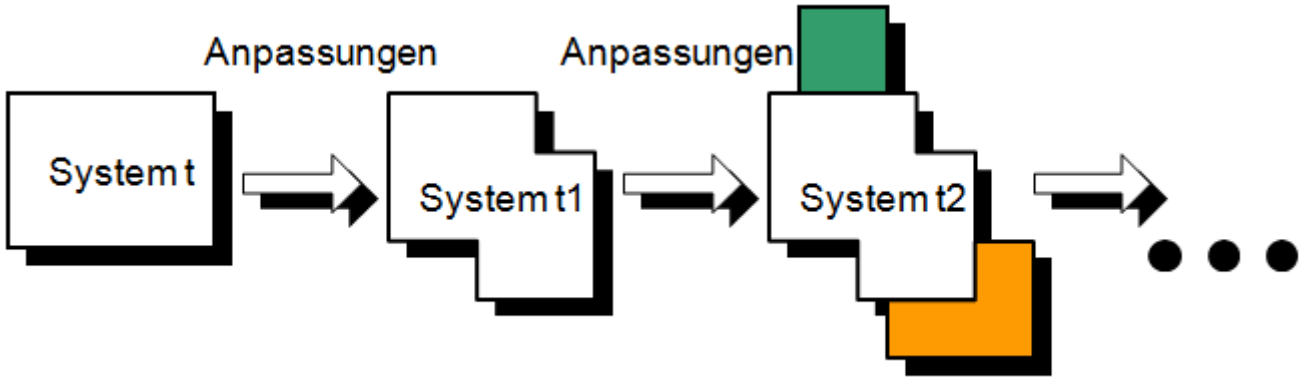


Ursachen für Qualitätsprobleme

- Zeitdruck während der Entwicklung
- Know-How-Defizite der Softwareentwickler

- Anforderungen an Softwaresysteme sind schwer zu erfassen und ändern sich ständig
  - Kluft zwischen Fachexperten und Softwareexperten
  - Lebensdauer moderner Systeme
  - Neue Einsatzkontexte

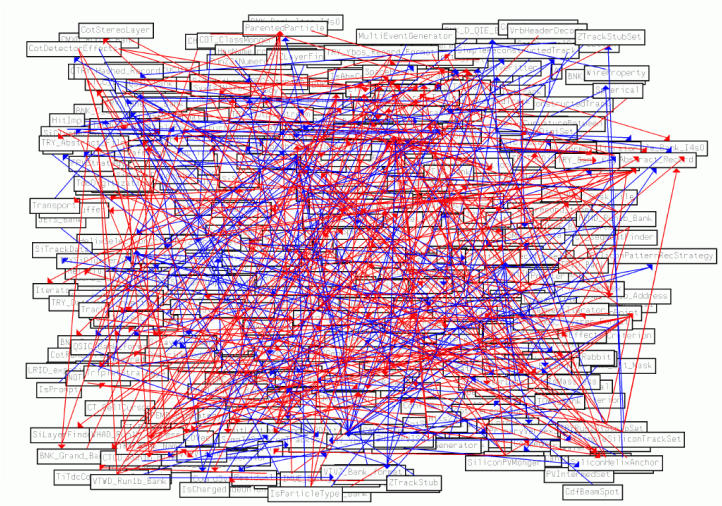
⇒ Softwarestrukturen degenerieren durch wiederholte Anpassungen

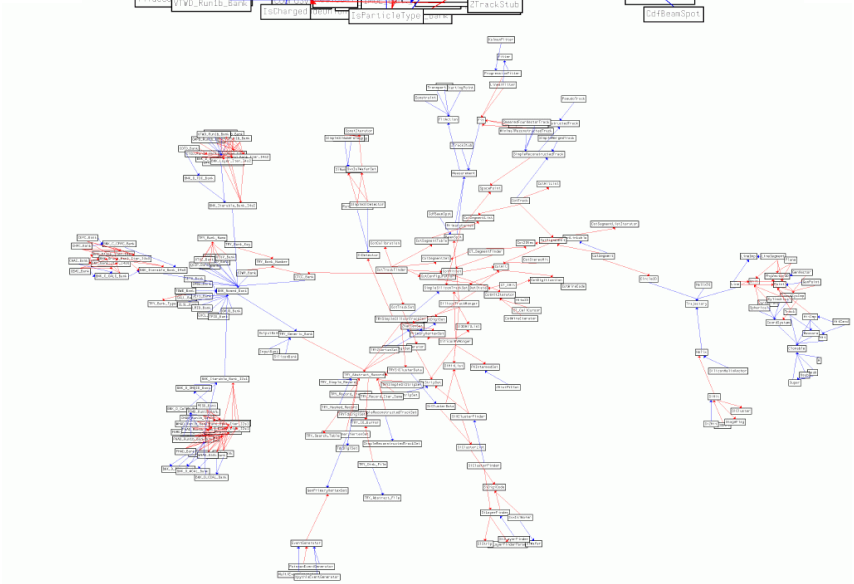


Grundprinzipien guter innerer Qualität

- **Abstraktion:**
  Schaffen einer vereinfachten Sicht auf Konzepte der Anwendungsdomäne
  (→Klassenbildung)
- **Kapselung:**
  Trennen von Schnittstelle und Implementierung
- **Modularisierung:**
  Zerlegen der Komplexität in handhabbare Einheiten
  (→Subsystembildung)

- Vernünftige Komplexität
- Geringe Kopplung, hohe Kohäsion





Richtlinien auf unterschiedlichen Ebenen

Architektur

- Vermeidung zirkulärer Abhängigkeiten
- Vermeidung breiter Subsystemschnittstellen
- Keine fragilen Einheiten in Schnittstellen
- Hohe Kohäsion im Subsystem, geringe Kopplung zum Rest des Systems

Design

- Klassen sollten nicht von Unterklassen abhängen
- Vermeidung von Implementierungs-vererbung
- Keine Flaschenhalsklassen
- Keine Gott-Klassen

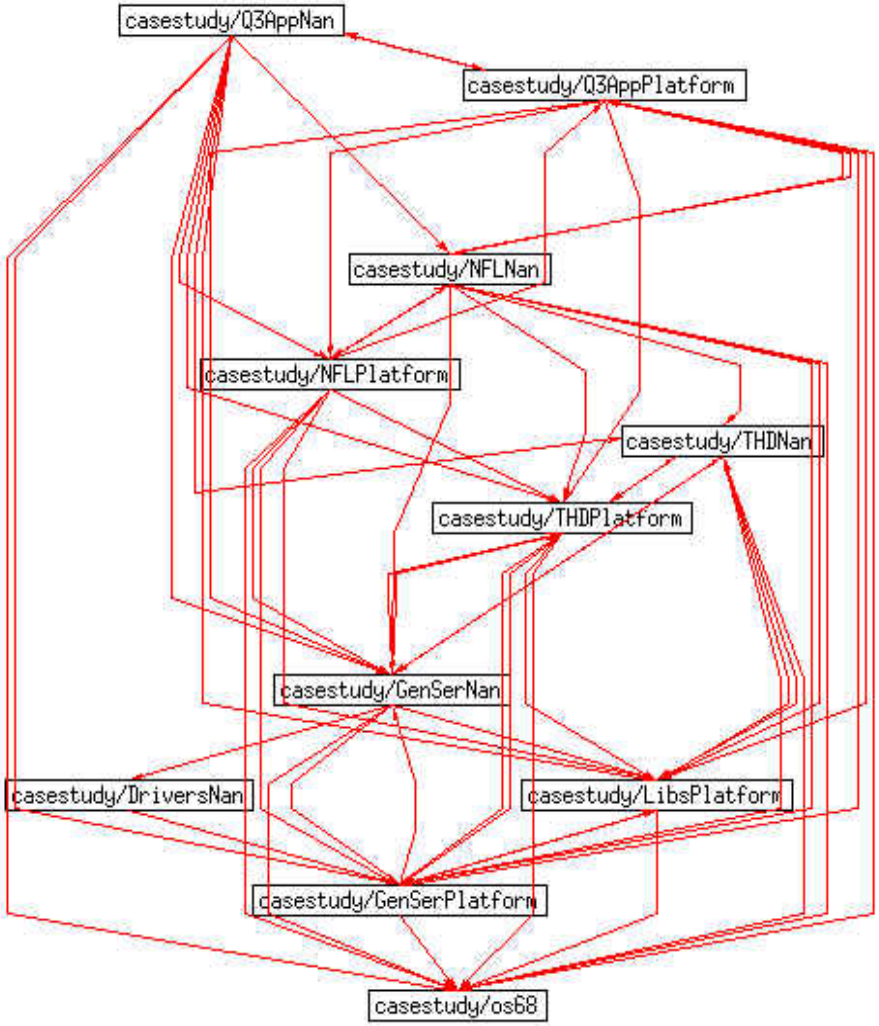
Implementierung

- Lokale Objekte sollten lokal freigegeben werden
- Variablen müssen vor Zugriff initialisiert werden
- Vermeidung schwer verständlicher Konstruktionen
- Kein toter Code
- Keine Code-Duplikation

Analyse der inneren Qualität

- Struktur- und graphbasierte Analysen
  - Visuelle Begutachtung der Architektur
  - Prüfen von Architekturregeln
  - Abhängigkeitsanalysen
- Bewertung von Softwaremaßen
  - Kopplung, Kapselung und Komplexität von Subsystemen und Klassen
  - Komplexität und Aufrufabhängigkeiten zwischen Methoden
- Musterbasierte Schwachstellensuche
  - Auffinden von Bad Smells, z.B.: Oberklassen mit Kenntnis ihrer Unterklassen
  - Überwachung (struktureller) Programmierrichtlinien
  - Auffinden typischer Fehlersituationen (Erfahrung nutzen!)
- Analyse von Codeduplikation

Beispiel: Visuelle Begutachtung der Architektur



Befunde:

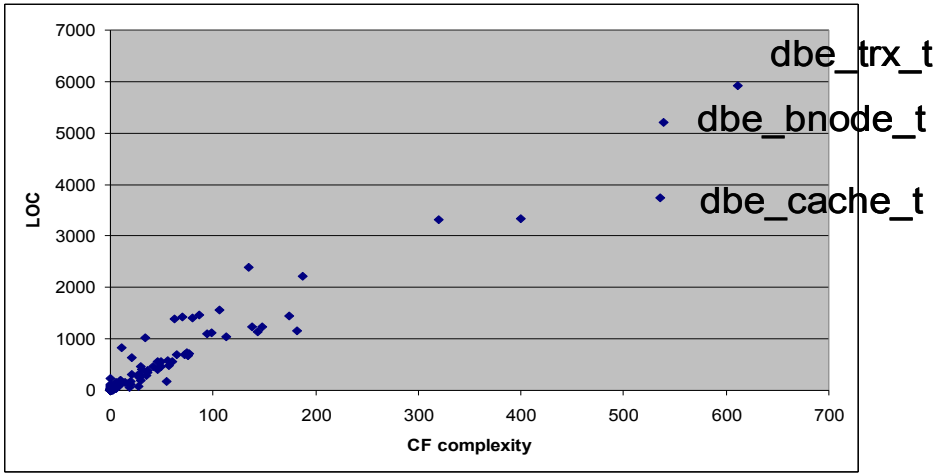
- Schichtung z.T. verletzt: LibsPlatform -> Q3AppPlatform
- Abbhängigkeiten von Plattformcode zum Produktcode: Q3Platform -> Q3AppNan

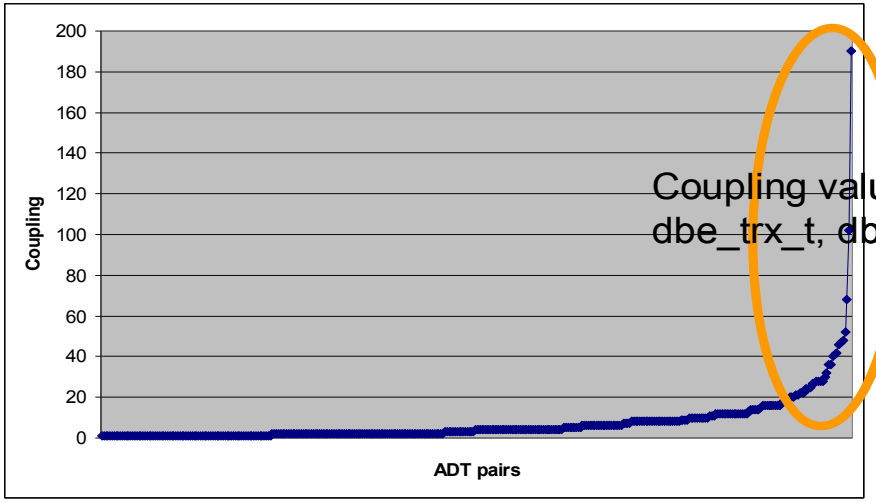
Korrektur von Abhängigkeiten:

- Plattform wiederverwendbar
- System leichter verständlich
- Buildzeit von 8h auf 2h gedrückt

Beispiel: Interpretation von Softwaremaßen

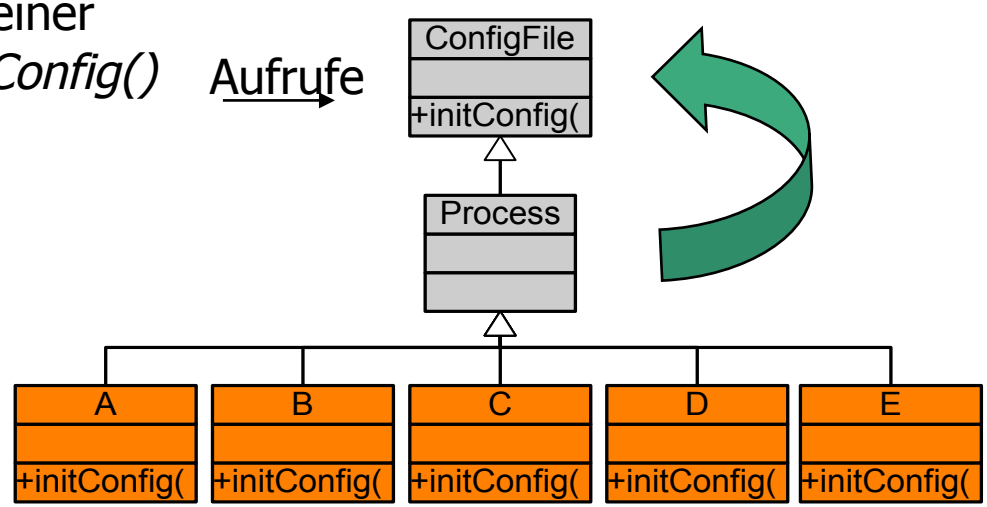
- Komplexitätsanalysen
  Nur wenige Datentypen sind sehr komplex
- Kopplung
  Nur wenige Paare von Datentypen mit hohen Kopplungswerten
- Probleme:
  - Einige Datentypen komplex und hoch mit anderen gekoppelt
  - Diese repräsentieren zentrale Konzepte; wahrscheinlich kann dies nicht verhindert werden





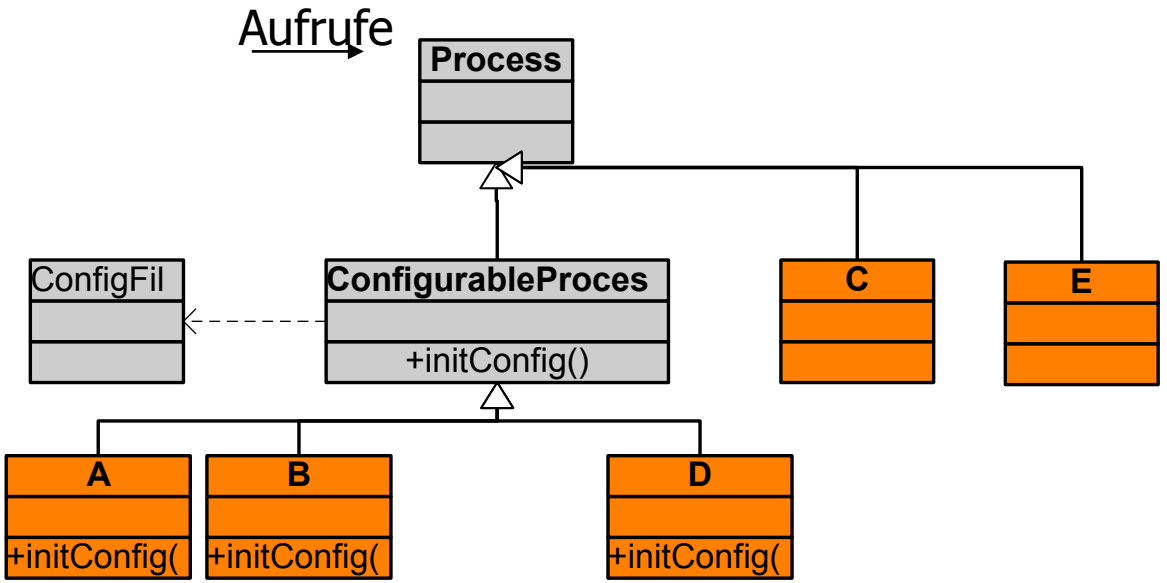
Beispiel: Musterbasierte Schwachstellensuche

- *ConfigFile* definiert Operationen zum Einlesen von Prozessparametern aus einer Konfigurationsdatei, z.B. *initConfig()*



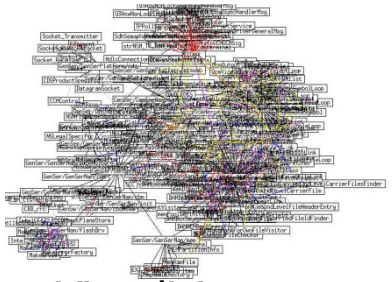
- Spezielle Prozesse *A* bis *E* überschreiben *initConfig():*
  - *C* und *E* : Implementierung von *initConfig()* ist leer
  - *A, B* und *E*: Implementierungen von *initConfig()* orientieren sich an der von *ConfigFile* (Codeduplikation!); zudem sind sie recht komplex.
- Viele Aufrufstellen verwenden die Schnittstelle von *Process*;
- *ConfigFile* wird nie verwendet!
- (*initConfig()* enthält case-Statements mit Typabfragen nach *A, B* und *E*)

Beispiel (Forts.)



- Es gibt jetzt explizit konfigurierbare und nicht konfigurierbare Prozesse
- Vererbung jetzt semantisch OK: Spezialisierung
- Gemeinsame Funktionalität von *initConfig()* in *A, B* und *D*:
  - Rumpfimplementierung in *ConfigurableProcess*
  - Hook-Methoden zur spezifischen Anpassung (Template Method Pattern) in *A, B, D*
- Einlesen der Konfigurationsdatei kann an *ConfigFile* delegiert werden

Werkzeugunterstützung



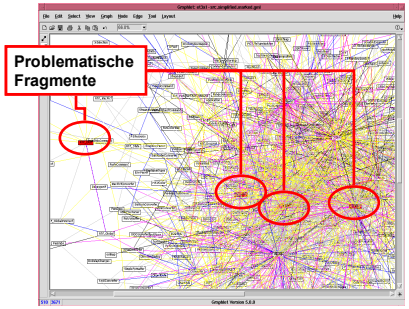
Visualisierung

Abstraktion

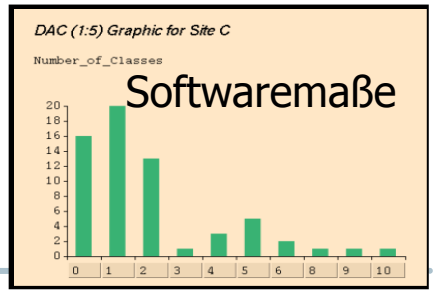
Quellcode

Struktur-DB

Faktenextraktion



Musterbasierte Suche;
Prüfung von Richtlinien



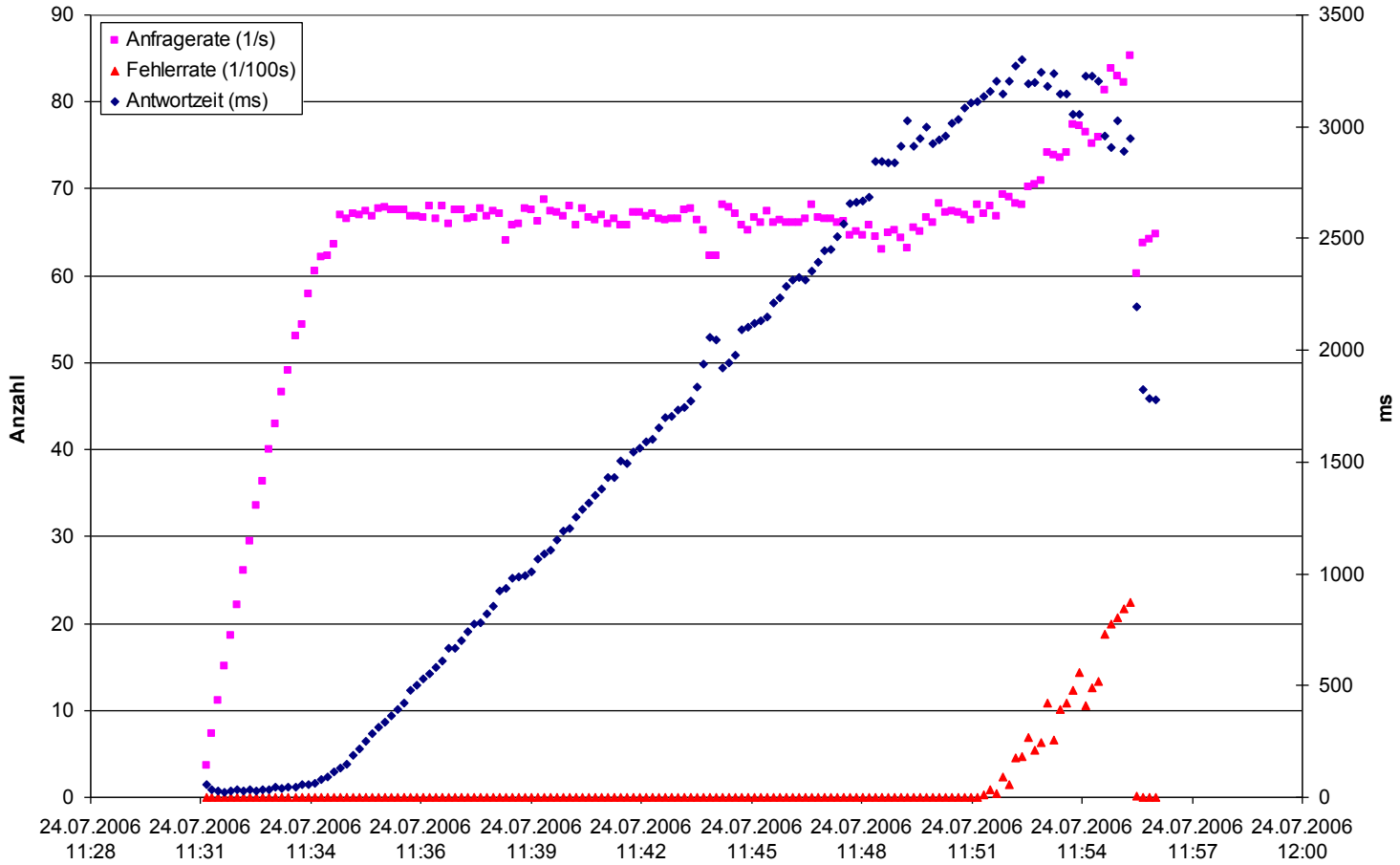
Beispiele für Werkzeuge

- Architekturanalyse, Strukturanalysen
  - STAN, http://stan4j.com (Java)
  - SonarJ, http://www.hello2morrow.com (Java)
  - SotoTools, http://www.hello2morrow.com (Java, C#, C/C++, APAP)
  - Codecrawler (Smalltalk) und Xray (Java), http://xray.inf.usi.ch/xray.php
- Implementierungsprüfung, musterbasierte Schwachstellenanalyse
  - Findbugs, http://findbugs.sourceforge.net/ (Java)
  - Checkstyle, http://checkstyle.sourceforge.net/ (Java)
  - Lint, http://splint.org/ (C/C++)

Tipps: Aus Erfahrung wird man Klug!

- Eine gute Struktur ist der Schlüssel zum Erfolg eines Systems!
- KISS: Keep it simple, stupid! (A. Tanenbaum)
  Wenn etwas zu kompliziert erscheint ➔ Vereinfachen!
- Zerlege Probleme in Teilprobleme!
  Miller‘s Law: Eine gute Struktur sollte es erlauben, dass man nie mehr als 7 (+/-2) Dinge im Kopf haben muss.
- Benenne Konzepte vernünftig!
  Wenn man etwas nicht benennen kann, hat man es noch nicht verstanden. ➔ Überdenken!
- DRY: Don’t repeat yourself!

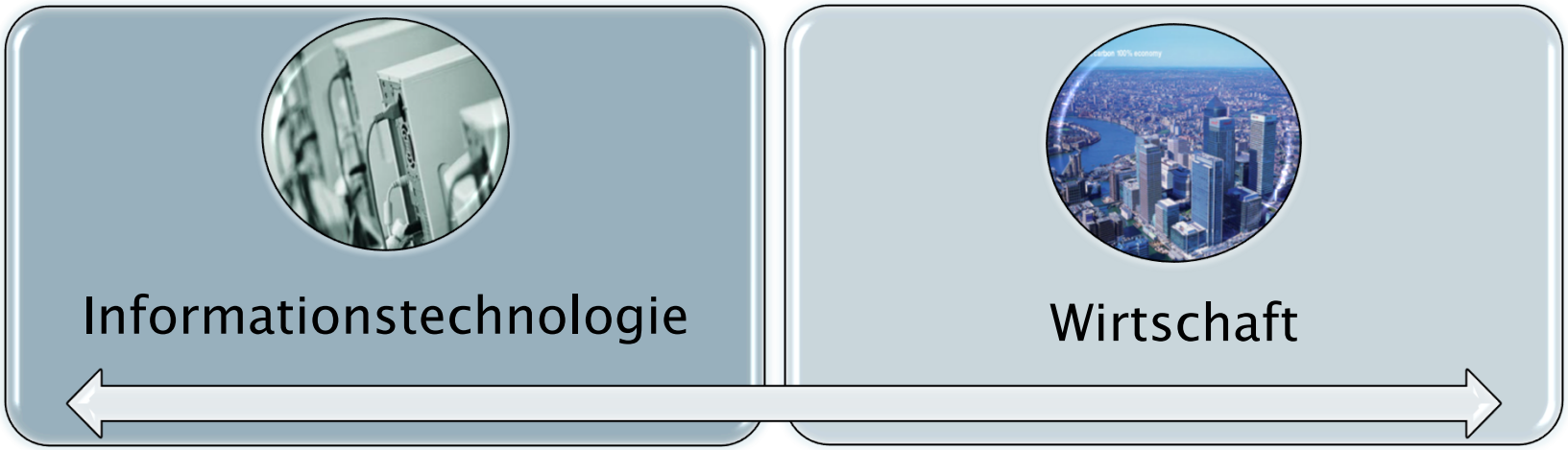
Beispiel: Auswertung von Performance-Tests



Fazit

- Softwareentwicklung ist eine Ingenieursdisziplin
- Gute Produkte erfordern ingeniermäßige Methoden
  - Nutzung erprobter Lösungen (→ Muster)
  - Entwicklung passender Architekturen
  - Solide Prozesse
  - Objektivierbarer Qualitätsbegriff, ständige Qualitätsprüfungen
- Es kommt auf die Menschen an!
  - Nutzer, die die Produkte nutzen (→ User Centric Design)
  - Ingenieure,… die die Produkte erschaffen (→ Know-How, Scrum)

Beruf(ung) des Wirtschaftsingenieurs (bei CAS)



Problemlöser, Nutzenstifter

Berater, Entscheidungshelfer

Prozessmanager (Kommunikation-, Geschäfts-, Projekt-…)

Product-manager	Consultant	Project-manager	Klassische Berufe der BWL

Fachliche Anforderungen

Grundsätzlich

- Hohe IT-Affinität
- Von Vorteil CRM-Knowhow
- Kommunikationsstärke
- Kundenorientierte & positve Einstellung

Einstiegsmöglichkeiten

- Produktmanagement
- Projektmanagement
- Vertrieb und Marketing
    - International & CRM-Software
    - CRM-Software für den Bildungsbereich
- Professional Services

Lernen Sie uns Kennen:

- Durch Praxiserfahrung – Abschlussarbeit/Praktikum
- Durch ein persönliches Bewerbungsgespräch
- Ansprechpartner für Ihre berufliche Entwicklung:

CAS Software AG
Eva Erdl und Daniel Kleinhans
Wilhelm-Schickard-Str. 8-12
76131 Karlsruhe
jobs@cas.de
0721-9638-779, oder -616