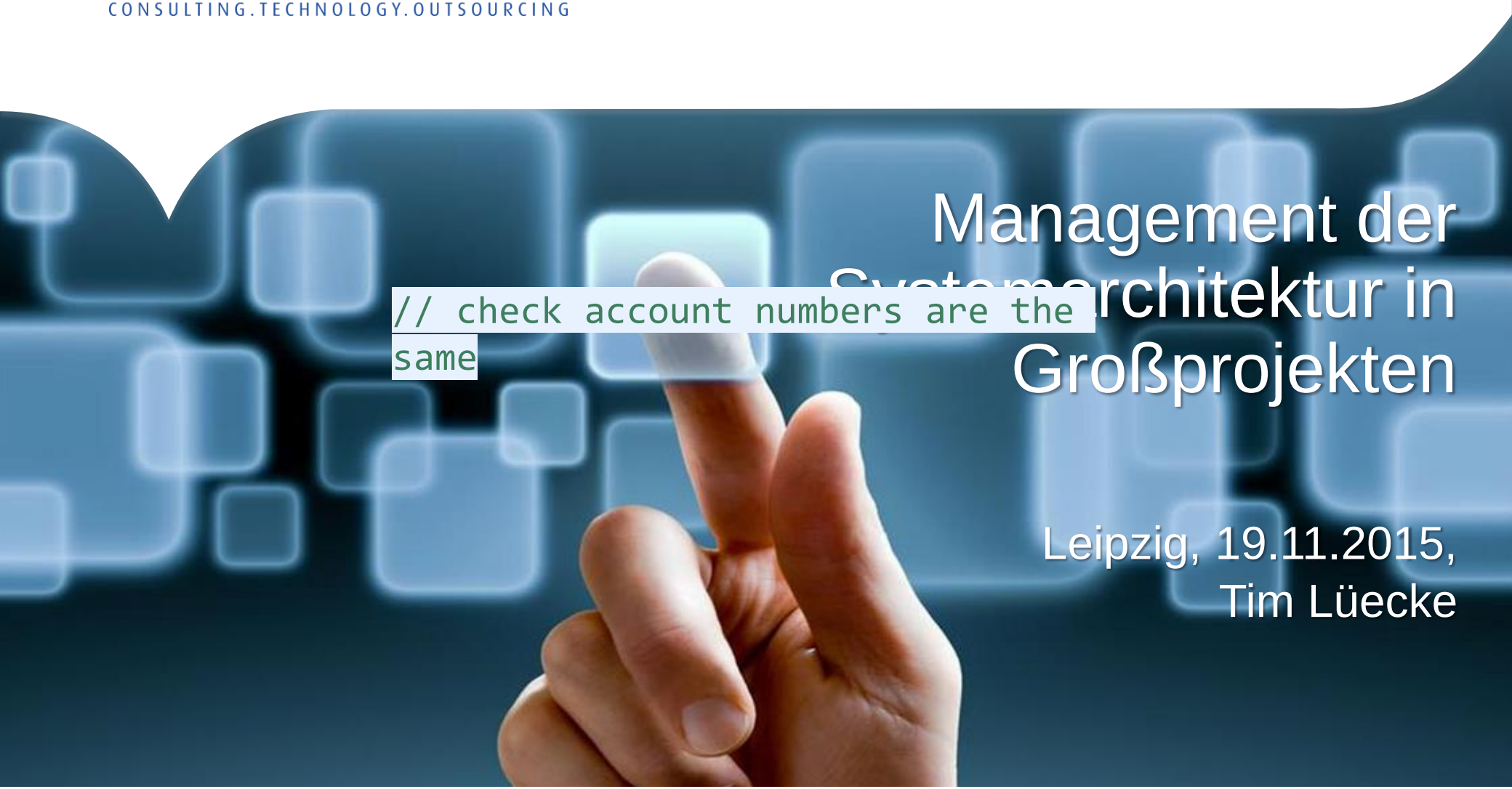
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Softwareforen Leipzig

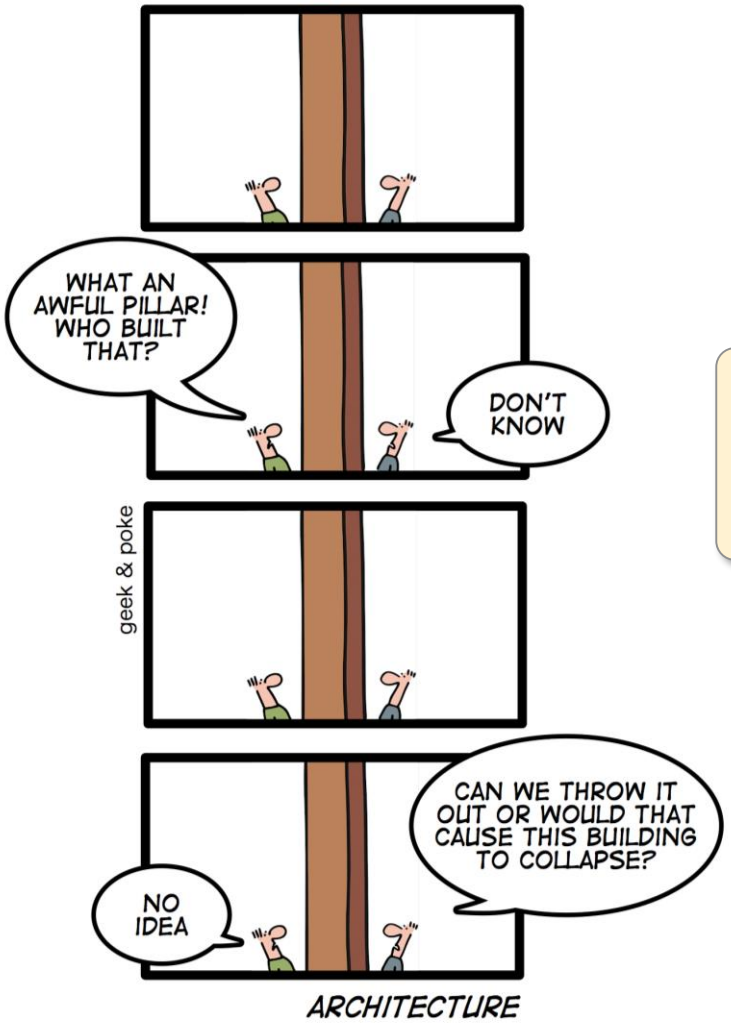


Management der Systemarchitektur in Großprojekten

Leipzig, 19.11.2015,
Tim Lüecke

Wozu Architektur-Management?

Vermeidung von strukturellen Monolithen



Auflösung von Architekturfehlern und Fehlentwicklungen

Quelle: http://geekandpoke.typepad.com/geekandpoke/2010/11/architecture.html

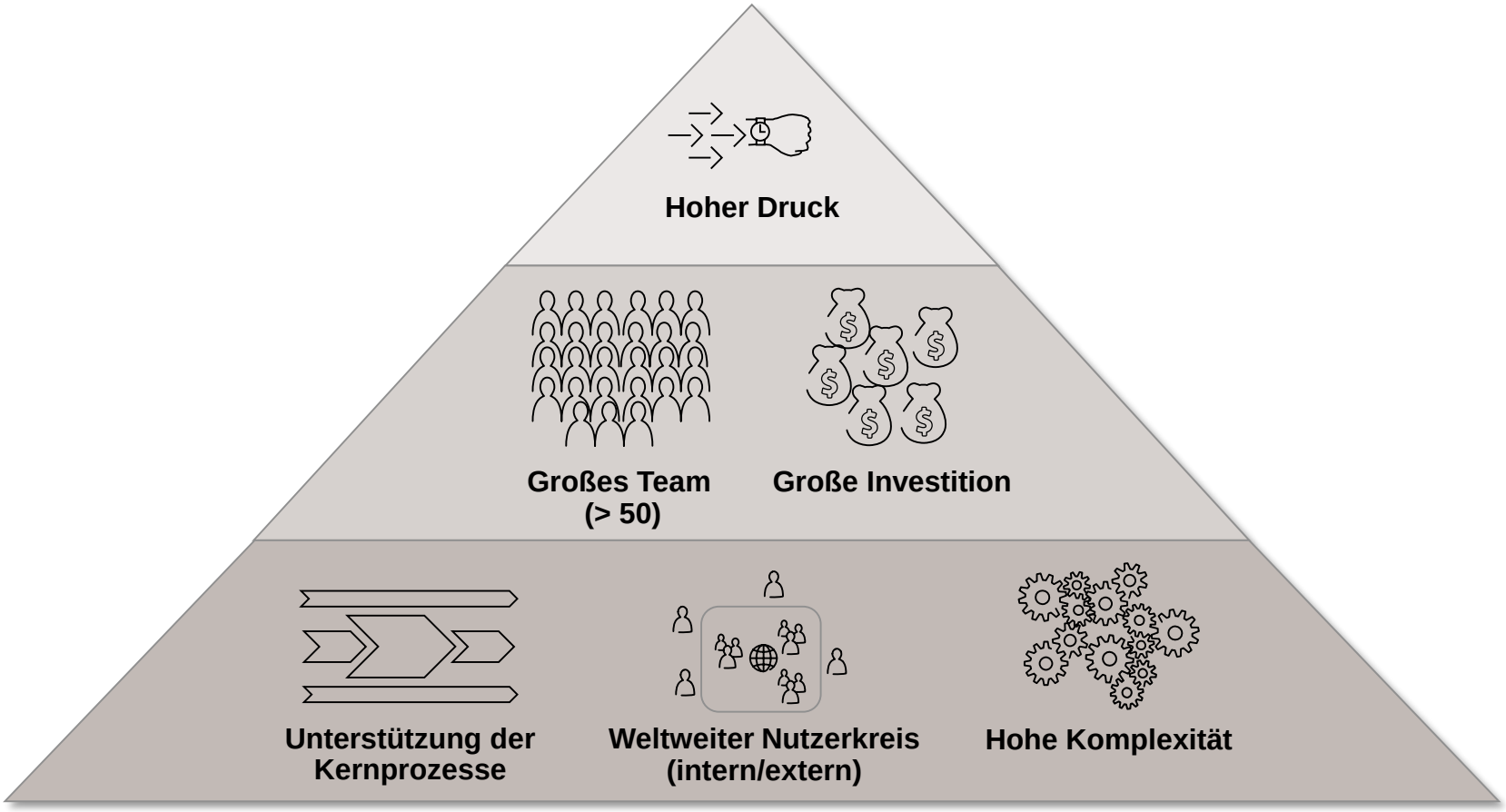
Agenda

- Großprojekte und ihre Herausforderungen
- Architekturmanagement in Großprojekten
- Refactoring in Großprojekten
- Zusammenfassung

Agenda

- **Großprojekte und ihre Herausforderungen**
- Architekturmanagement in Großprojekten
- Refactoring in Großprojekten
- Zusammenfassung

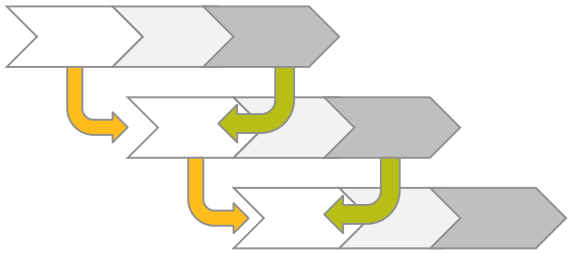
Großprojekte zeichnen sich durch eine hohe Komplexität gepaart mit einer hohen Management-Attention aus



Kontext dieses Vortrages sind Individual-Software-Großprojekte in einem iterativen Wasserfallprozess

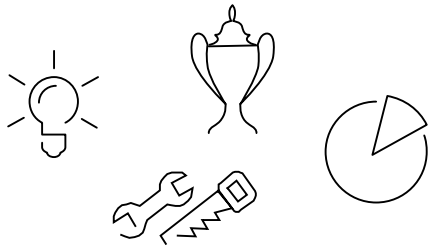
Klassisches Vorgehensmodell

- Kernprozesse meist schon gegeben (Vorgängersystem evtl. vorhanden)
- Ziel der Reise ist bekannt
- Grobplanung mit möglichst stabilen Terminplan zur Steuerung der Einführung erforderlich
- Diverse Stakeholder ohne klar identifizierbaren Product Owner

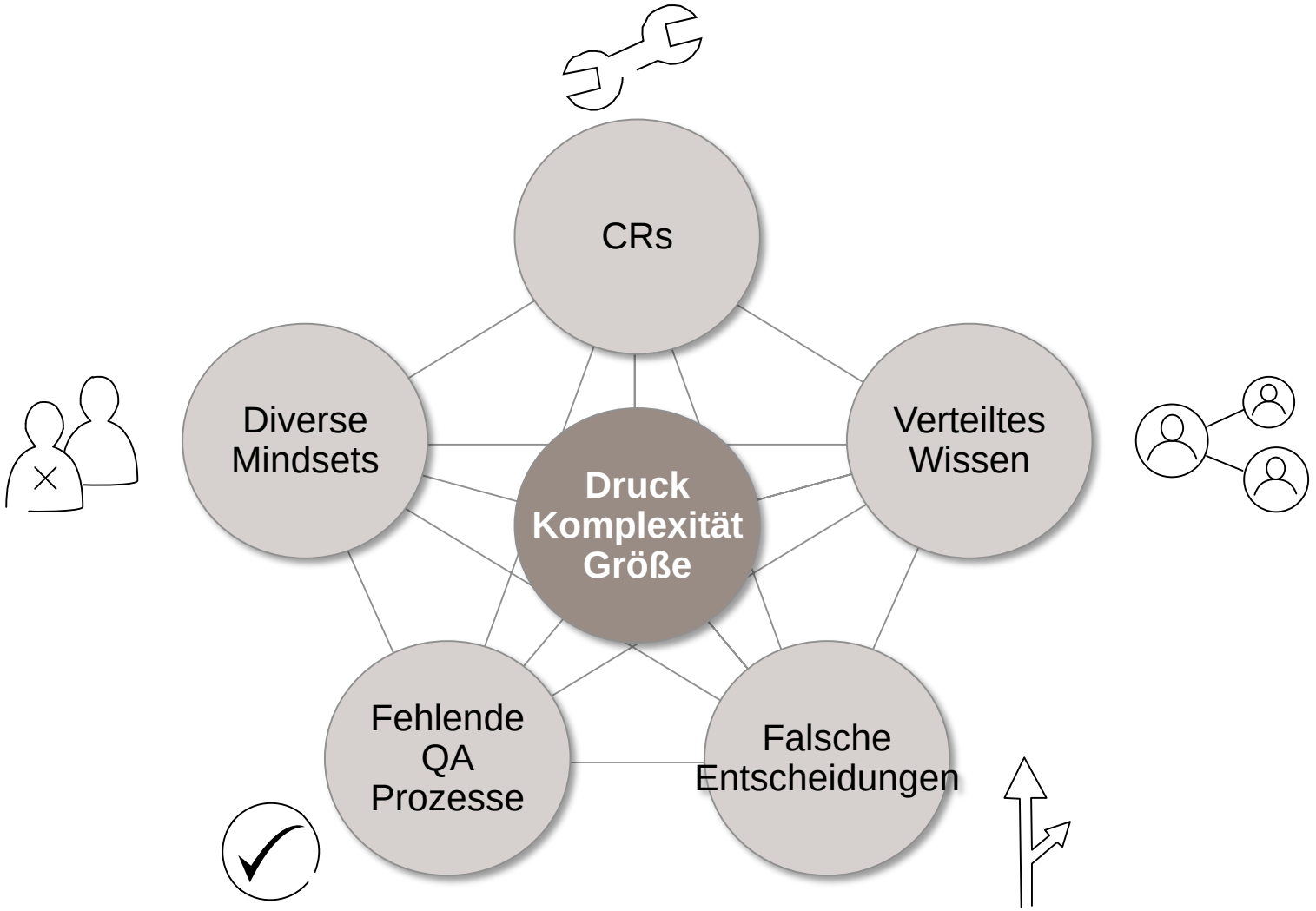


Individualsoftware

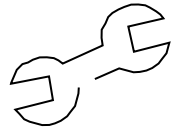
- Software wird von Grund auf selbst entwickelt
- Maßgeschneidert zur bestmöglichen Unterstützung der Kernprozesse
- Ermöglicht Differenzierung in den Kernkompetenzen eines Unternehmens
- Technische Basis kann potentiell wiederverwendet werden



Großprojekte bringen einige Herausforderungen für die zugrundeliegende Architektur mit sich



Wegen der langen Laufzeit häufen sich Änderungs-Anforderungen, die schnell umgesetzt werden müssen



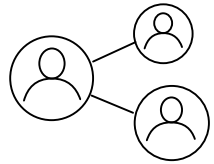
Time-to-market

- Weltweite Anforderungen können nicht alle im Voraus bedacht werden
- Anforderung ändern sich nach dem ersten Release (oder auch während)
- Änderungen sollen asap eingearbeitet werden (“Emergency CRs”)
- Führt oft zu „Hacks“ (durch ungenügendes Wissen, Druck, ...)

Gesetz der Software Entropie (Lehman)

- Eine Software, die verwendet wird, wird auch verändert
- Wenn eine Software geändert wird, erhöht sich die Komplexität, sofern nicht aktiv dagegen gesteuert wird

Die Verteilung des Wissens gestaltet sich über ein großes Team äußerst schwierig



Ursache

Übereiltes Ramp-up

- Erstellung der technischen Basis parallel zu der Entwicklung
- „Moving Target“
- Zu schneller Team-Aufbau

Beschränkung auf Dokumentation

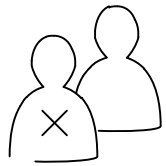
- Mitteilung von Regeln über Mails / Handbüchern / Wiki ungenügend
- Fehlende Begründung
- Entwickler haben andere Sorgen („*TAGRI*“)

Team-übergreifende Kommunikation

- z.B. bei Trennung der Teams nach Disziplinen
- Knowledge Transfer muss unterschiedliche Perspektiven genügen

- **Architektur Wissen unterschiedlich verteilt**
- **Kennt man eine Regel nicht, wird sie auch nicht befolgt**
- **„Elfenbein-Turm-Architekturen“**

Ein großes Team bringt viele unterschiedliche Meinungen mit sich, die oft im Konflikt stehen



Beispiel

Minimierung von Abhängigkeiten

- “Wenn ich es doch aber brauche, muss ich es halt kennen!”
- “Was ist falsch an zyklischen Abhängigkeiten? Das ist fachlich halt so…”

Trennung von Zuständigkeiten

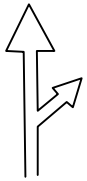
- “Es ist doch viel einfacher alles an einer Stelle zu implementieren!”
- “Man versteht das doch nicht mehr, wenn es so verteilt ist”

Schichten-Architektur

- Schichten können in einfachen Fällen künstlich wirken
- “Da wird doch kaum etwas gemacht, wieso muss das getrennt werden?!”
- “Das ist nicht effizient genug!”

- **Ohne Akzeptanz wird eine Architektur nicht befolgt**
- **Fehlende Motivation frustriert**

Falsche Architektur Entscheidungen haben enorme Auswirkungen



Beispiel

Schichten

Details

- Schichten ohne klare, disjunkte Zuständigkeiten
- Entwickler wissen nicht wo genau sie was implementieren müssen
- Führt zu den unterschiedlichsten Lösungsansätzen

Komponenten

- Komponentenschnitt mit zu vielen Zuständigkeiten
- Falsche Kopplung zwischen den Komponenten
- Falscher Schnittstellen-Schnitt mit Hinblick auf Performance

Qualitätssicherung wird gerade am Anfang häufig zugunsten von Ergebnissen vernachlässigt

Ursache

Fokus auf Implementierung

- "Lasst uns erst mal anfangen, wir prüfen das dann alles später"
- Automatischer Tool-Support fehlt, oder muss erst noch aufgesetzt werden

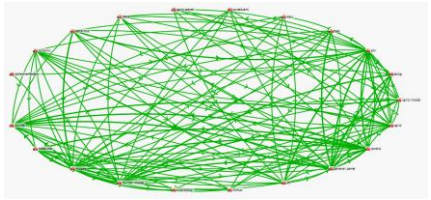
Zeitdruck

- Qualitätssicherung wird oft als erstes gestrichen, wenn die Zeit knapp wird
- Fehlende Management Awareness

- **Regeln werden verletzt**
- **Regelverletzungen breiten sich schneller aus als man denkt**

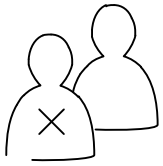
"You can't manage what you can't control, and you can't control what you don't measure" (Tom DeMarco)

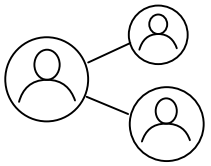
Werden diese Herausforderungen nicht gemeistert, führt dies schnell zu struktureller Erosion

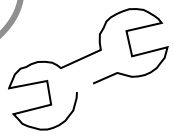


Strukturelle Erosion

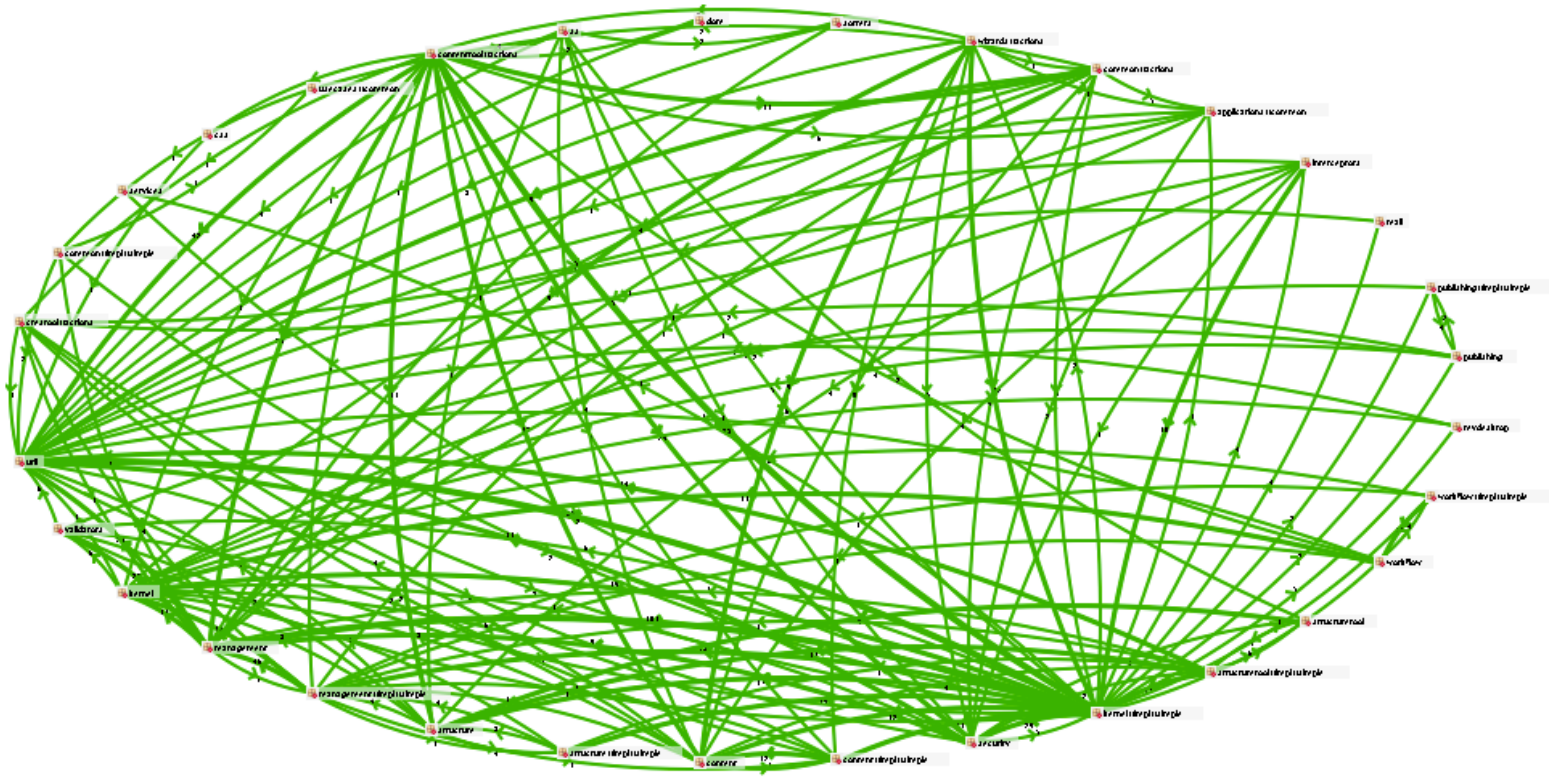
[...]







Bei struktureller Erosion verschwindet die Architektur in einem Knäuel von Abhängigkeiten



Quelle: http://www.hello2morrow.com

Strukturelle Erosion zeichnet sich durch bestimmte Merkmale aus (Robert C. Martin)

Symptom	Erläuterung
Rigidity / Fragility	▪ Anpassungen an einer Stelle wirken sich an anderen Stellen aus ▪ Regelmäßig fehlschlagende Builds auf Modul-Ebene ▪ Paralleles Arbeiten in großen Projekten immens erschwert
Opacity / Immobility	▪ Source Code ist schwer zu verstehen: wo findet sich was? ▪ Fehlende Trennung von Zuständigkeiten ▪ Wiederverwendbare Komponente sind schwer zu identifizieren und einzuführen
Viscosity	▪ Es ist leichter etwas falsch zu machen als richtig ▪ Benutzung von Code, der gegen Regeln verstößt, führt zu neuen Verstößen

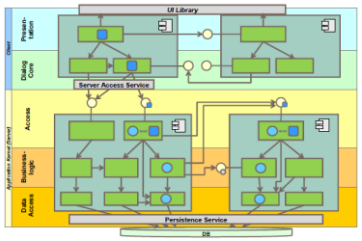
s. Agile Software Development,
Robert C. Martin, Prentice Hall 2003

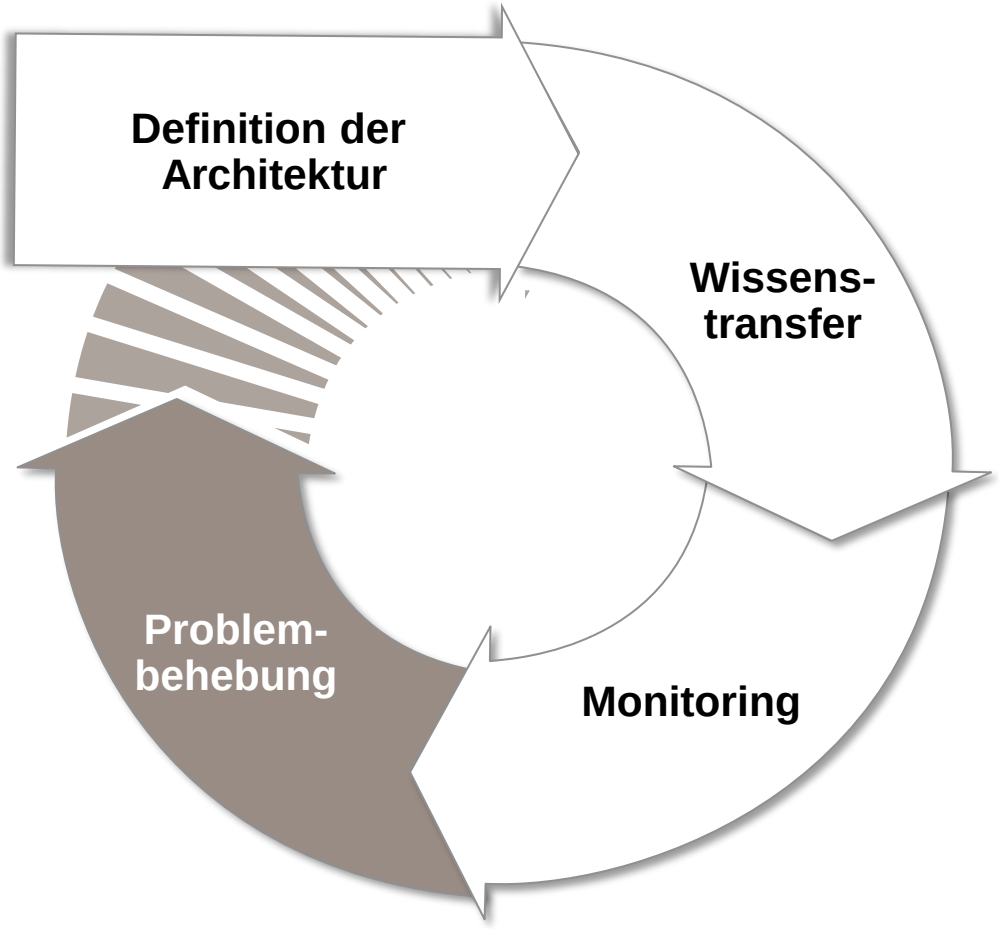
"The software starts to rot like a bad piece of meat"

Agenda

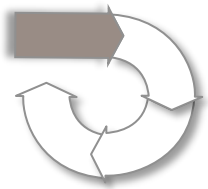
- Großprojekte und ihre Herausforderungen
- **Architekturmanagement in Großprojekten**
- Refactoring in Großprojekten
- Zusammenfassung

Großprojekte erfordern ein kontinuierliches Architekturmanagement



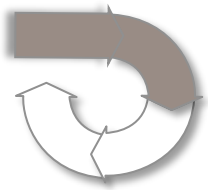


Definition der Architektur muss klar, verständlich und nachvollziehbar formuliert werden



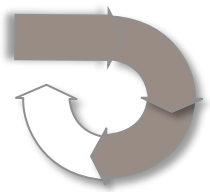
Aspekt	Zu beachten
Inhalt	▪ Angabe klar verständlicher Definitionen der Architekturelemente ▪ Definition, Hervorhebung und Motivation (!) von Regeln ▪ Entwurfsentscheidungen explizit mit Alternativen festhalten ▪ Namenskonvention zur Klassifizierung von Artefakten
Struktur	▪ Dokumentation über verschiedene Architektur-Perspektiven ▪ Technische Architektur als UML Modell ▪ Fachliche Architektur als Word Dokument
Nur ein Startpunkt	▪ Architekturen sind nicht in Stein gemeißelt ▪ Was nicht funktioniert, muss geändert werden ▪ Änderungen sollten mit Bedacht erfolgen und mit dem Management abgestimmt werden

Ein geordneter Wissenstransfer für die Architektur ist unabdingbar



Aspekt	Zu beachten
Veröffentlichung der Architektur	▪ Einfacher Zugang für jeden Entwickler ▪ Verknüpfung zu anderen Dokumentationen (z.B. Entwicklerhandbuch) ▪ Bereitstellung eines expliziten Regelkatalogs mit Beispielen
Architektur-Schulungen	▪ Veröffentlichung alleine nicht ausreichend ▪ Schulungen zur Vermittlung des Inhalts vorbereiten ▪ Motivation für die Architektur (d.h. die Regeln) hervorheben ▪ Raum für Diskussionen lassen – aber gut vorbereitet sein ;-)

Architektur-Verletzungen müssen frühzeitig und kontinuierlich zumindest erfasst werden



Aspekt	Zu beachten

Überprüfung der Regeln	▪ Auch der beste Wissenstransfer verhindert nicht alle Regelverletzungen
	  ▪ Regeln können oft nicht durch die Sprache forciert werden
	  ▪ Bei komplexen Implementierungen oftmals vernachlässigt
	▪ Erfassung und Dokumentation von Regel-Verletzungen

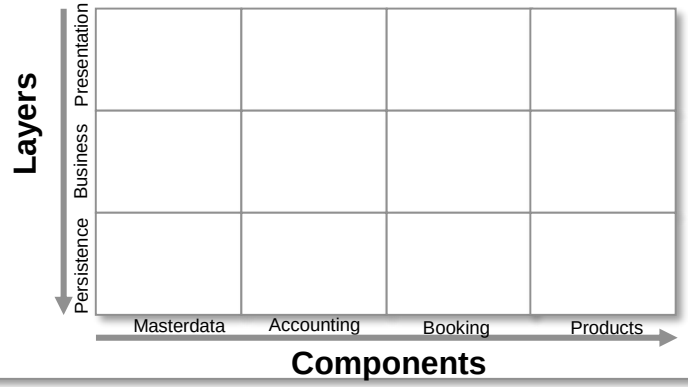
Tool-Unterstützung	▪ Wenn möglich sollte Regelprüfung automatisch erfolgen, z.B. über
	  ▪ Sonargraph
	  ▪ Sonarqube
	▪ Sollte Teil der Continuous Integration sein

Code Reviews	▪ Falls kein Tool vorhanden, sollte dies Teil von Code Reviews sein (z.B. über Checkliste) ▪ Durchführung von einem erfahrenen Entwickler
Frühzeitig und kontinuierlich	▪ Verletzungen verbreiten sich oft rasant ▪ Je später erfasst, desto schneller steigen die technischen Schulden

Sonargraph bietet eine einfache Möglichkeit Architekturen zu modellieren und zu kontrollieren

Architektur Modell

- Unterstützung von zwei Dimensionen:
  - Technische Architektur (Schichten)
  - Anwendungs-Architektur (Slice = Komponente)
- Explizite und Implizite Abhängigkeiten
- Flexiblere Modellierung über DSL in neuester Version

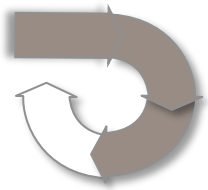


Quellcode

- Zuweisung von Architekturelementen über Namenskonvention (s. Architektur-Definition)
- Best Practice: [company].[system].[component].[layer]

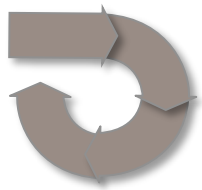
```
/*
 * Copyright 2012 Johann Gyger
 */
package name.gyger.jmoney.service;

import name.gyger.jmoney.model.Account;

@Service
@Transactional
public class OptionsService {

    private static final Logger log = LoggerFactory.getLogger(OptionsService.class);

    private static final Entry.Status[] entryStates = {null, Entry.Status.RECONCILING, Entry.Status.CLEARED};

    @PersistenceContext
    private EntityManager em;

    private Session session;

    private net.sf.jmoney.model.Session oldSession;

    private Map<net.sf.jmoney.model.Category, Category> oldToNewCategoryMap = new HashMap<net.sf.jmoney.model.Category, Category>();

    private Map<Entry, net.sf.jmoney.model.Category> entryToOldCategoryMap = new HashMap<Entry, net.sf.jmoney.model.Category>();

    private Map<net.sf.jmoney.model.DoubleEntry, Entry> oldToNewDoubleEntryMap = new HashMap<net.sf.jmoney.model.DoubleEntry, Entry>();

    @Inject
    private SessionService sessionService;

    public void init() {
        removeOldSession();
        Session session = new Session();
        initCategories(session);
        em.persist(session);
```

Sonargraph bietet Features, die für das Architekturmanagement äußerst nützlich sind



Feature	Details
Verletzungen	▪ Abgleich der Quellcode-Abhängigkeiten mit Architekturmodell ▪ Identifikation und Auflistung der Abhängigkeitsverletzungen
Tasks	▪ Tasks für virtuelle Refactorings (Verschiebung, Umbenennung, etc.) ▪ Tasks zur Auflösung von Abhängigkeiten ▪ Erlaubt die virtuelle Behebung der oben aufgeführten Verletzungen
Package-Zyklen	▪ Identifikation von Zyklen zwischen Packages ▪ Vorschläge zum Auflösen von Zyklen
Metriken	▪ Code Duplizierung ▪ ACD / NCCD ▪ Komplexitäts-Metriken

Die Behebung der Probleme und die Rückkopplung der Erkenntnisse müssen aktiv gemanagt werden



Aspekt	Zu beachten
Problembehebung	▪ Priorität sollte mit Bedacht und nach eingehender Analyse erfolgen ▪ Verletzungen breiten sich schnell aus (Viskosität) ▪ Finden der Balance zwischen Weiterentwicklung und Behebung der strukturellen Schulden
Feedback Schleife	▪ Konkrete Verletzungen als Beispiel für Wissenstransfers heranziehen ▪ Prüfen der Architektur-Definition mit evtl. Anpassung falls erforderlich

Agenda

- Großprojekte und ihre Herausforderungen
- Architekturmanagement in Großprojekten
- **Refactoring in Großprojekten**
- Zusammenfassung

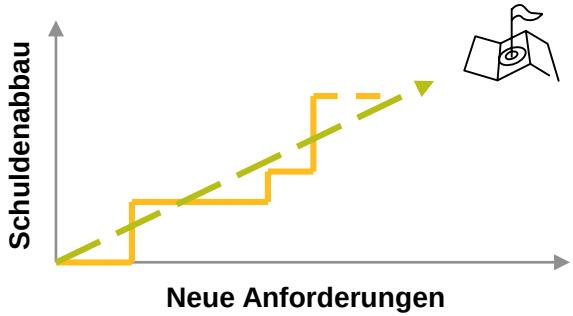
Technische Schulden sind in Großprojekten unvermeidlich und müssen in Rahmen von Refactorings aufgelöst werden

Metapher

- Formuliert von Ward Cunningham
- **Entstehung von Schulden**:
  - Unvollständiges Verständnis der Systemanforderungen führt zu „leicht falschem“ Code
  - Fachliche Architektur passt z.B. nicht zu den Anforderungen
- **Auswirkung**:
  - wie bei finanziellen Schulden müssen Zinsen gezahlt werden
  - Neue Anforderungen lassen sich schwerer umsetzen
- Inzwischen wird auch die Code-Qualität als Teil der technischen Schulden gesehen (anders als von Cunningham gedacht)

Abbau von Schulden

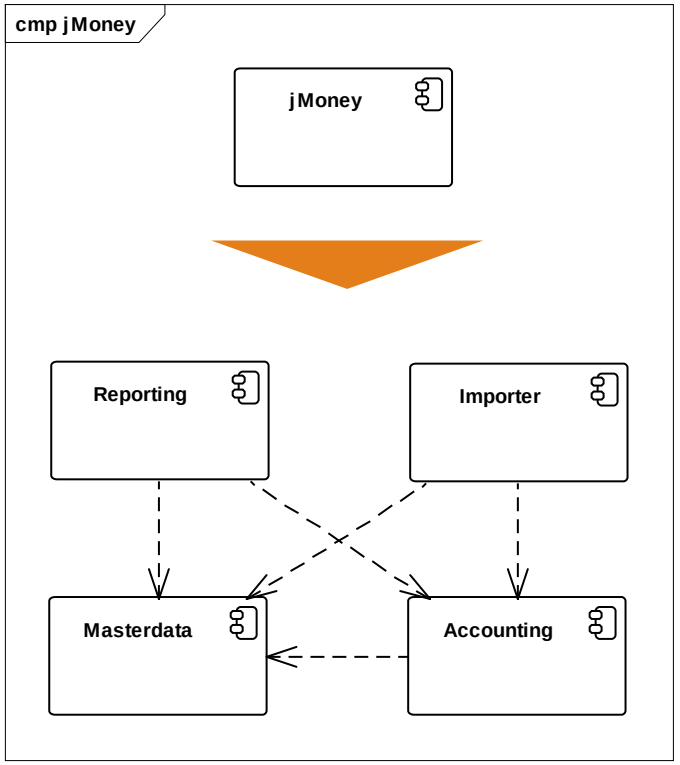
- Refactorings zur Korrektur der Architektur
- Kann fachliche aber auch technische Architektur betreffen
- Sinnvolle Bewertung notwendig:
  - dort ansetzen, wo es für die Zukunft hilfreich ist
  - Planung inkl. Schätzung als Grundlage
  - Zielbild zur Orientierung



Ein typisches Beispiel ist die Auftrennung einer zu groß gewordenen Komponente

Beispiel

- **Problem**: Größe einer Kernkomponente
  - Größe wurde vom Design nicht antizipiert
  - Neue Zuständigkeiten im Laufe der Zeit hinzugefügt
  - Abhängigkeiten in der Komponente nicht mehr überschaubar
- **Ergebnis**: parallele Weiterentwicklung der Komponente in mehreren Teams nicht möglich
- **Lösungsansatz**:
  - Auftrennung in kleinere Komponenten mit klar definierten Zuständigkeiten
  - Behebung von anderen Architekturverletzungen „*along-the-way*“



Die Refactoring Planung sollte methodisch und tool-unterstützt erstellt werden, um Fallstricke zu vermeiden

Phase:	**Ziel Definition**	Sonargraph Modell	Virtuelle Verteilung	Definition Abhängig-keiten	Kompo-nenten Trennung	Schichten Trennung
Cont.:	▪ Komponenten Modell ▪ Rein fachlich orientiert ▪ Ohne Abhängigkeiten	▪ Quell-Komponente beibehalten ▪ Neue Komponenten definieren ▪ Ohne Abhängigkeiten	▪ Packages + Klassen umbenennen ▪ Dadurch: Zuweisung zu neuen Komponenten ▪ Tasks immer mit einem Ticket verknüpfen	▪ Können empirisch ermittelt werden ▪ Optimierung anhand von Verletzungen ▪ Anschließend fachliche Validierung	▪ Schichten-verletzungen ignorieren ▪ Refactoring Tasks zu über-geordneten Aufgaben bündeln (JIRA) ▪ Controlling über Sonargraph	▪ Aufgaben bündeln pro Typ und Komponente ▪ Controlling über Sonargraph

Erstellung eines Komponentenmodells als Leitbild

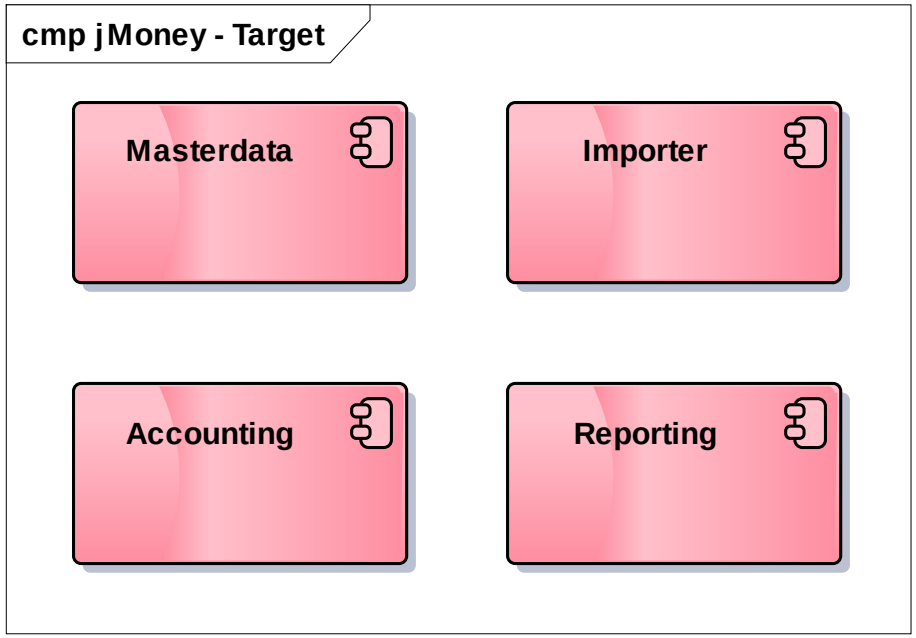
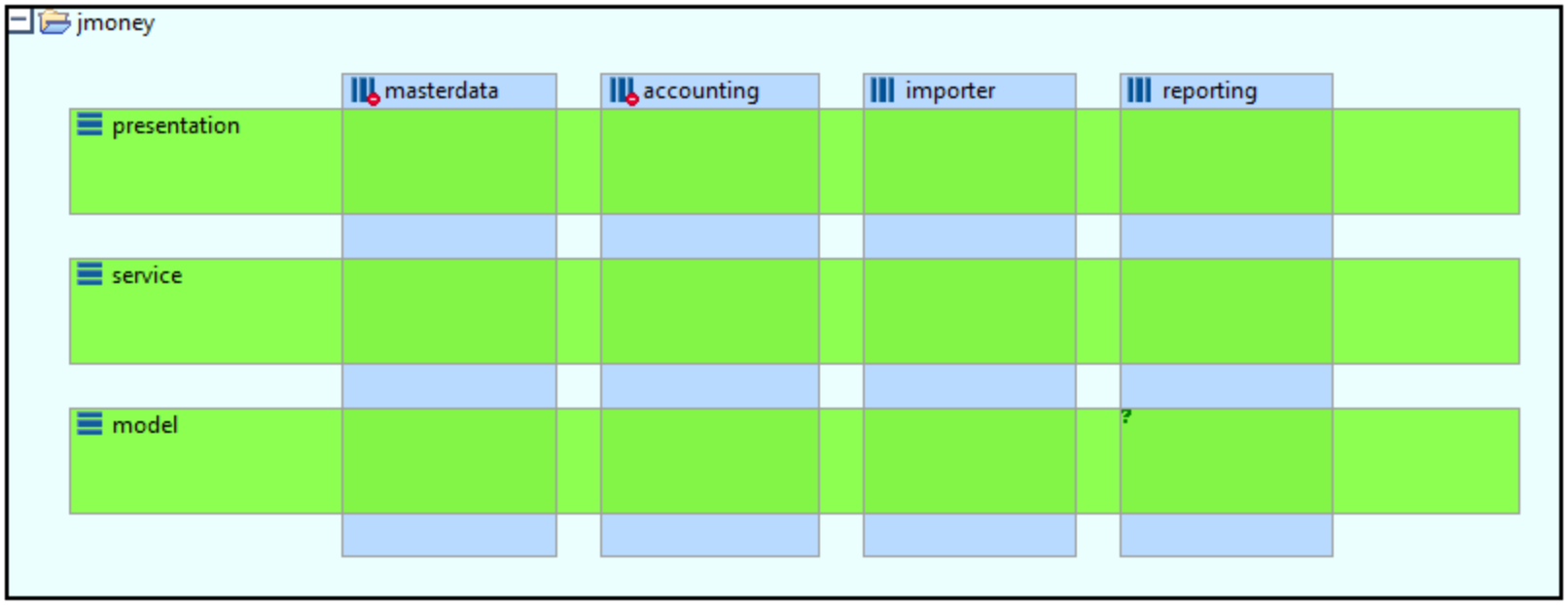
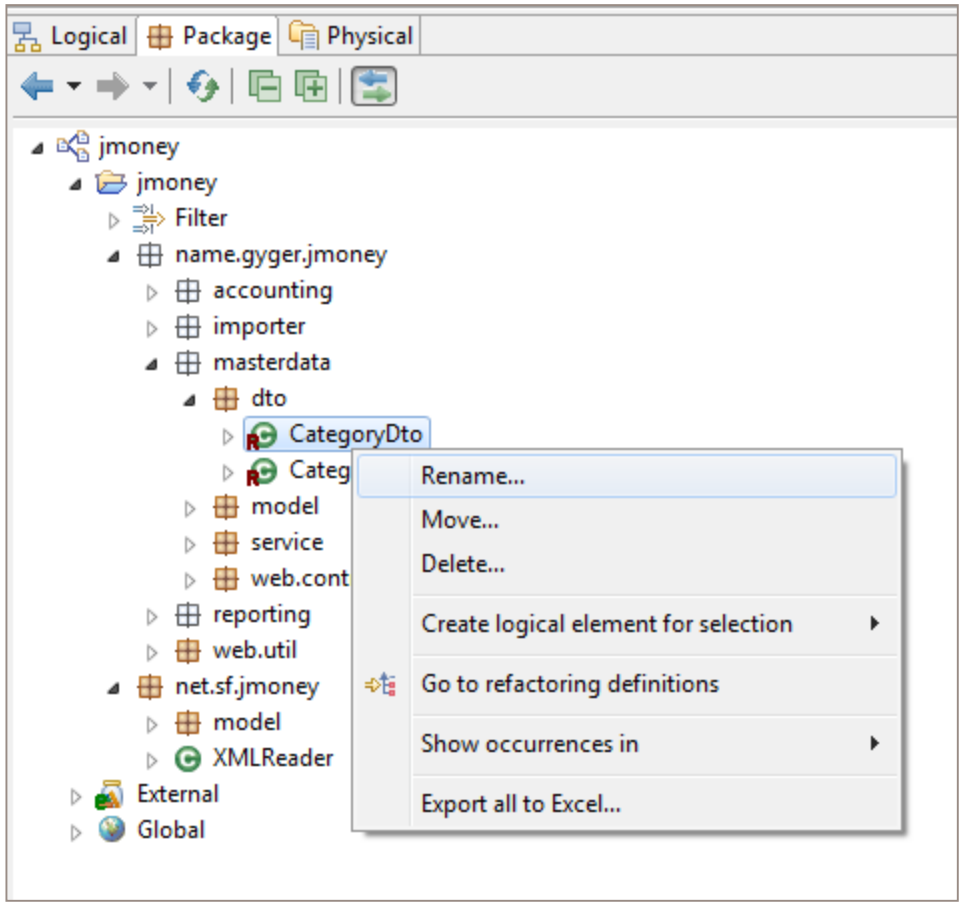
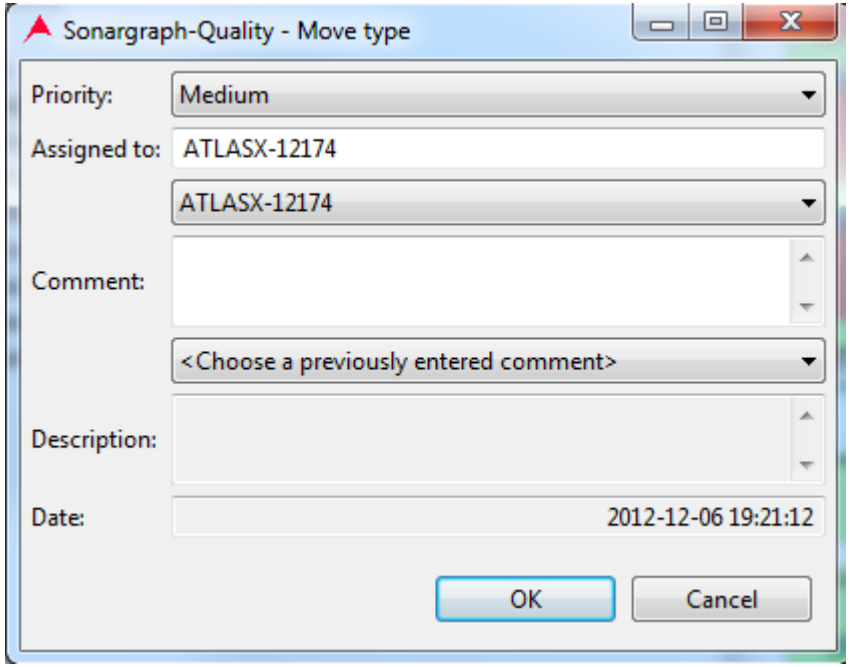


Abbildung des Komponentenmodells in Sonargraph

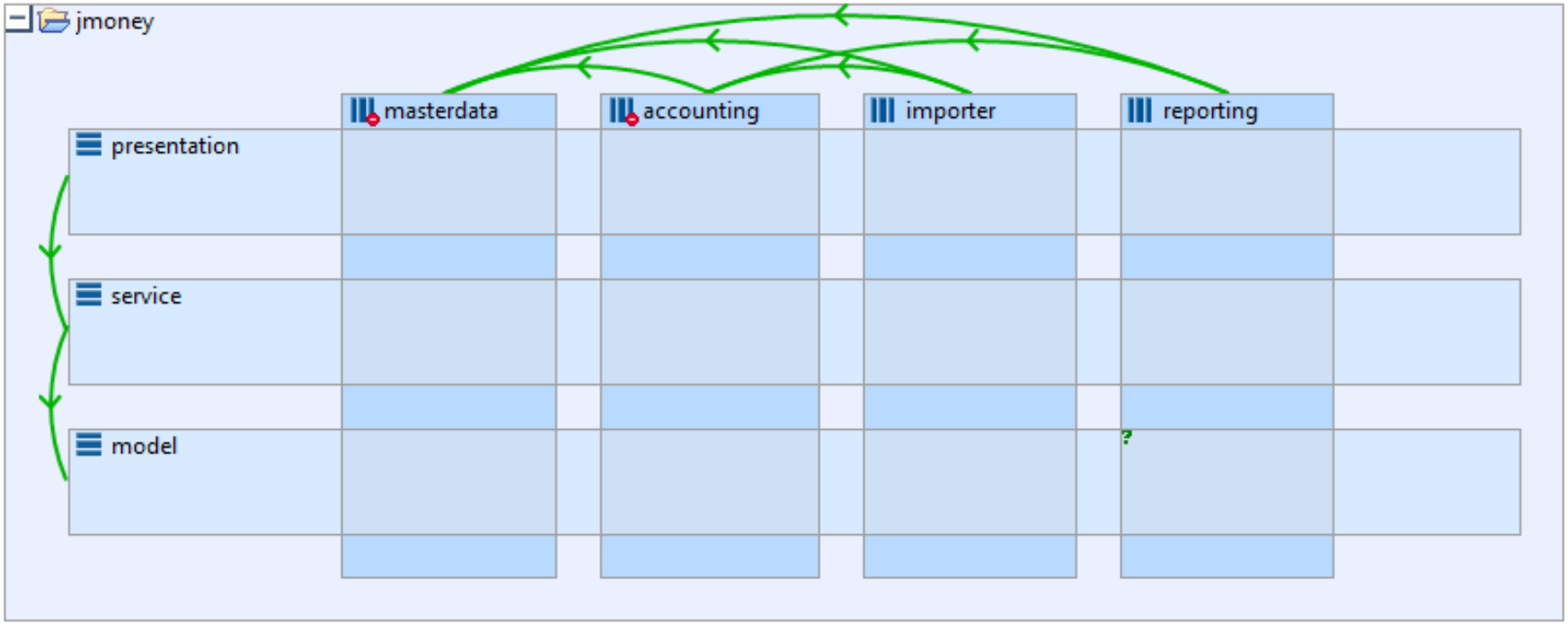


Virtuelle Verteilung der Klassen und Packages auf die Komponenten

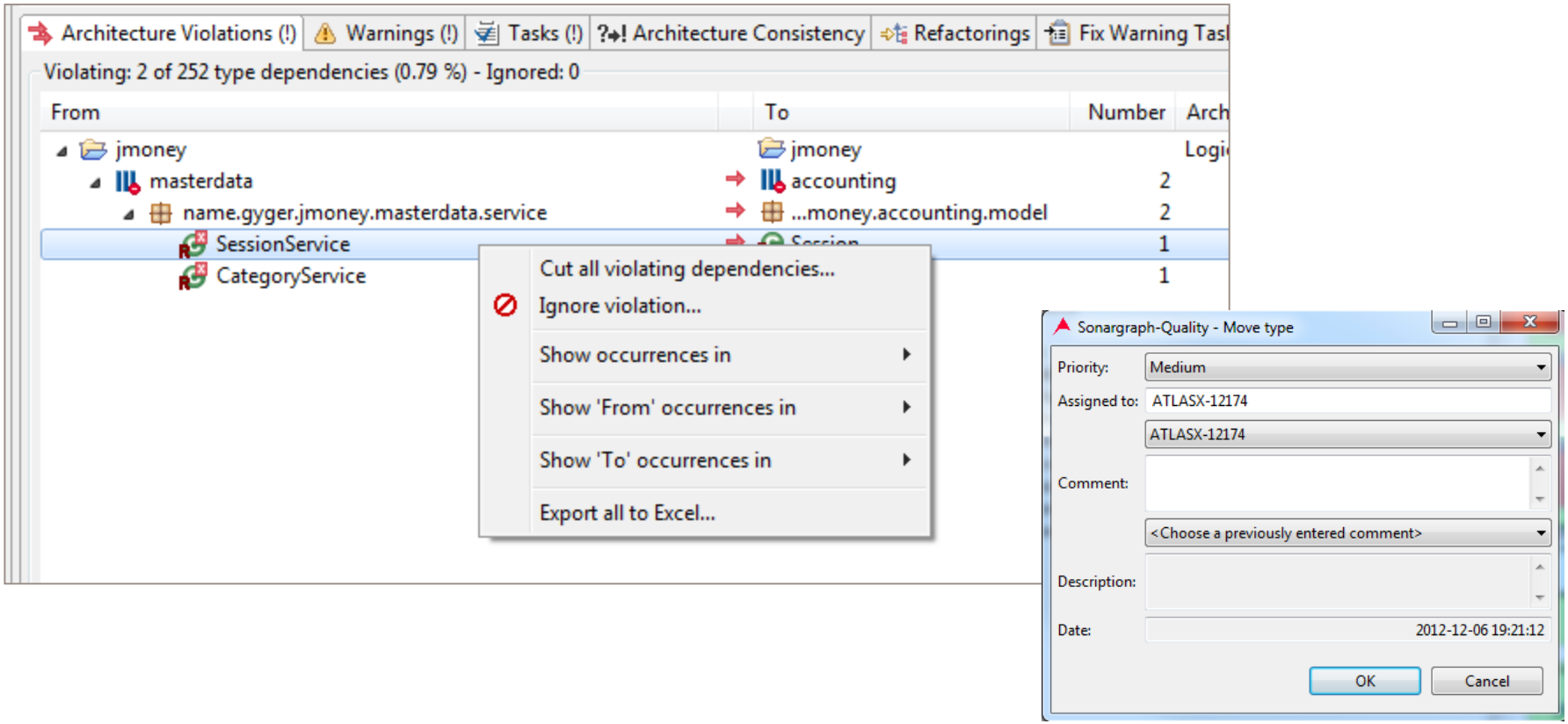




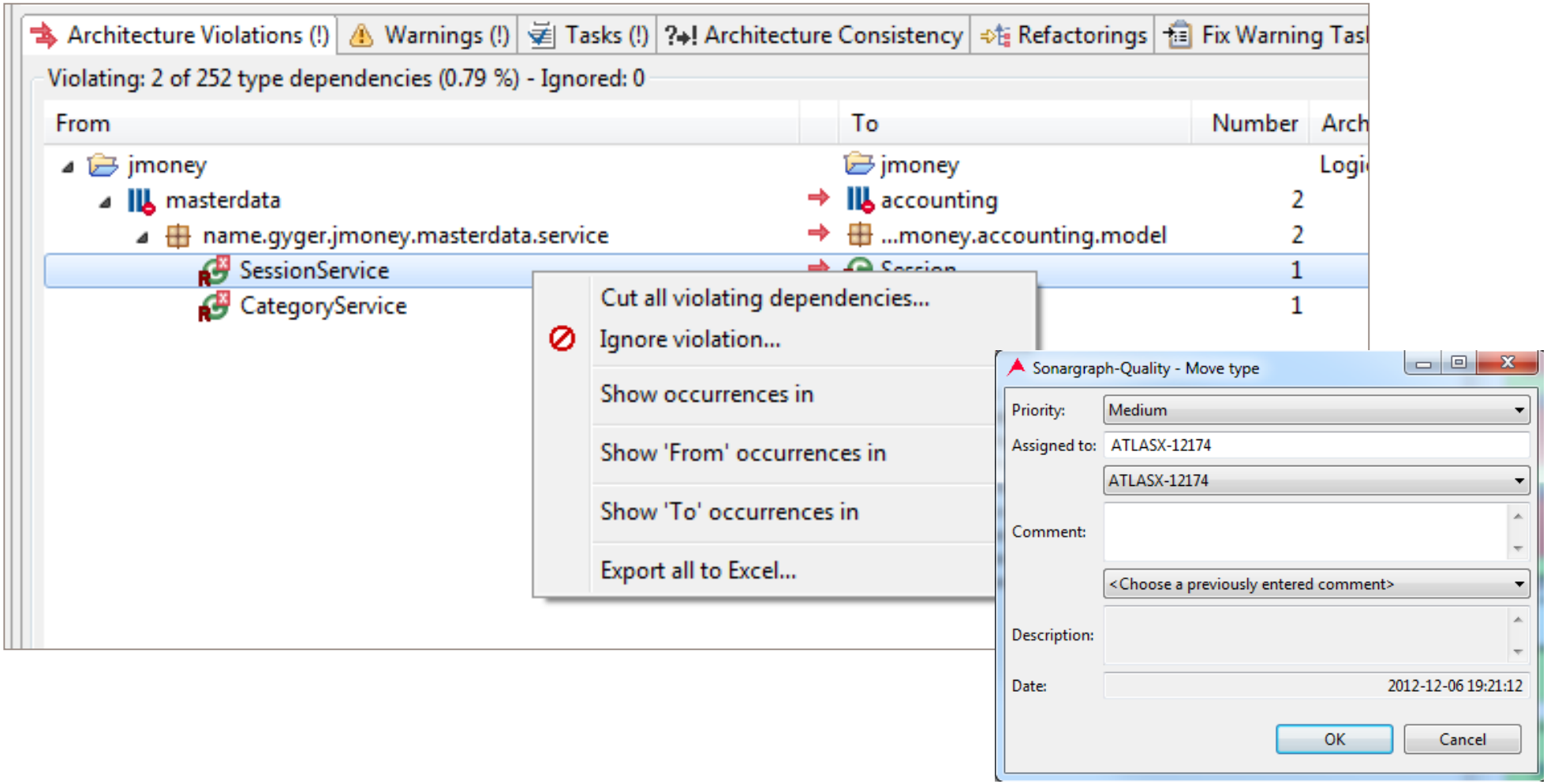
Definition der fachlichen Abhängigkeiten durch hybride Soll-Ist-Analyse



Virtuelle Trennung der fachlichen Komponenten



Abschließend: Schichtenverletzung ebenfalls betrachten (optional)



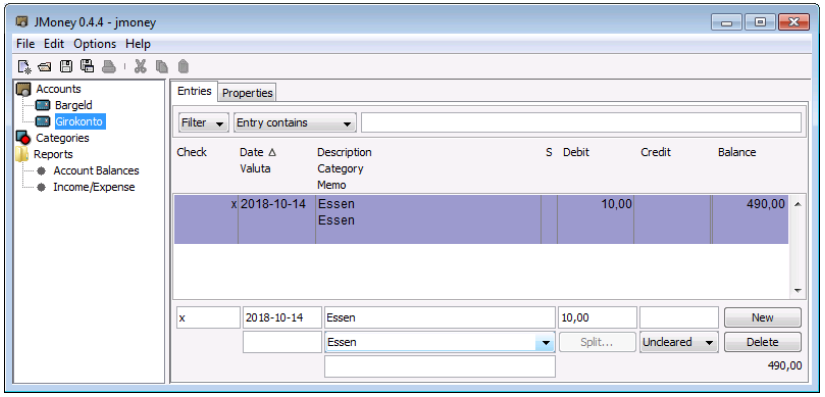
Die Durchführung des Refactoring sollte in Phasen erfolgen, um ein paralleles Arbeiten zu ermöglichen

Phase:	**Package Refactoring**	**Komponenten Trennung**	**Build Unit Refactoring**	**Aufräumen**
Cont.:	▪ Ausführung der Rename und Move Tasks ▪ Build Unit bleibt unberührt ▪ Ermöglicht paralleles Refactoring und erhöht Stabilität	▪ Ausführung der Tasks zum Auflösen der unerlaubten Abhängigkeiten ▪ Ein Bearbeiter pro Komponente sinnvoll	▪ Build Unit auftrennen ▪ Ermöglicht die Einführung harter Abhängigkeiten, die nicht mehr verletzt werden können	▪ Ausführung der restlichen Refactoring Tasks

Sonargraph am Beispiel: JMoney (OpenSource)

Details

- Kleine Swing-Anwendung zur Verwaltung der persönlichen Finanzen (
- Setup:
  - Konten für die Erfassung von Buchungen (Bargeld, Bankkonto, etc.)
  - Kategorienverwaltung zur Klassifizierung von Buchungen
- Buchungen können für die angelegten Konten durchgeführt und kategorisiert werden
- Optionen zur Initialisierung und Import von Daten aus der alten Swing Applikation



Quelle: http://jmoney.sourceforge.net/

Agenda

- Großprojekte und ihre Herausforderungen
- Architekturmanagement in Großprojekten
- Refactoring in Großprojekten
- **Zusammenfassung**

Zusammenfassung

- Aktives Architekturmanagement ein MUSS zur Wahrung der technischen Qualität
- Prozess sollte frühzeitig (am besten am Anfang) aufgesetzt werden
- Transparenz über den aktuellen Stand schaffen, um Prioritäten klären zu können
- Toolunterstützung wie Sonargraph (wenn möglich) nutzen
- Bei Erosion: toolgestütztes Refactoring durch Sonargraph mit methodischer Durchführung

Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

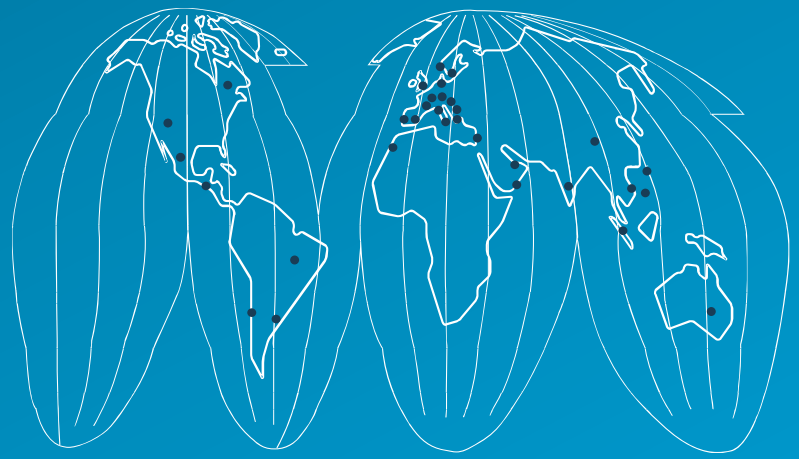
People matter, results count.

Über Capgemini

Mit mehr als 140.000 Mitarbeitern in über 40 Ländern ist Capgemini einer der weltweit führenden Anbieter von Management- und IT-Beratung, Technologie-Services sowie Outsourcing-Dienstleistungen. Im Jahr 2013 betrug der Umsatz der Capgemini-Gruppe 10,1 Milliarden Euro.

Gemeinsam mit seinen Kunden erstellt Capgemini Geschäfts- wie auch Technologielösungen, die passgenau auf die individuellen Anforderungen zugeschnitten sind. Auf der Grundlage seines weltweiten Liefermodells Rightshore® zeichnet sich Capgemini als multinationale Organisation durch seine besondere Art der Zusammenarbeit aus – die Collaborative Business Experience™.

Rightshore® ist eine eingetragene Marke von Capgemini



www.de.capgemini.com