Softwarearchitektur und Qualität

„Gute Qualität kommt von Innen!“

Dr. Markus Bauer
Leiter Entwicklung CAS PIA und CAS Open

5.11.2009

Agenda

- Einführung
- Innere Softwarequalität
- Architektur-, Design- und Implementierungsrichtlinien
- Werkzeuggestützte Qualitätsanalyse
- Praxisbeispiel
- Tipps

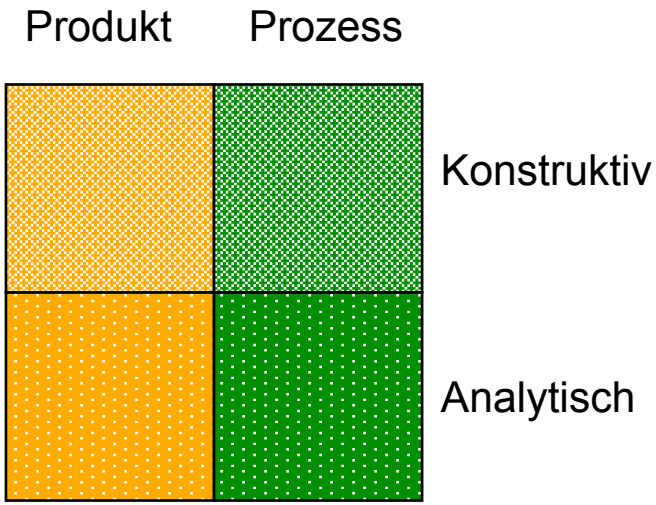
Einführung

„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
  - 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



Ein paar Zahlen...

1968

- Die NATO proklamiert die Softwarekrise: Software hat schlechte Produktqualität und verursacht immense Wartungs- und Weiterentwicklungskosten!

1994

- IBM: Umfrage unter 24 großen Softwarefirmen – 88% der Softwaresysteme benötigen ein grundlegendes Re-Design!

2000, 2001

- Gartner Group, IDC: Der überwiegende Anteil der Entwicklungszeit und –kosten wird für Wartung und Weiterentwicklung benötigt.

2002

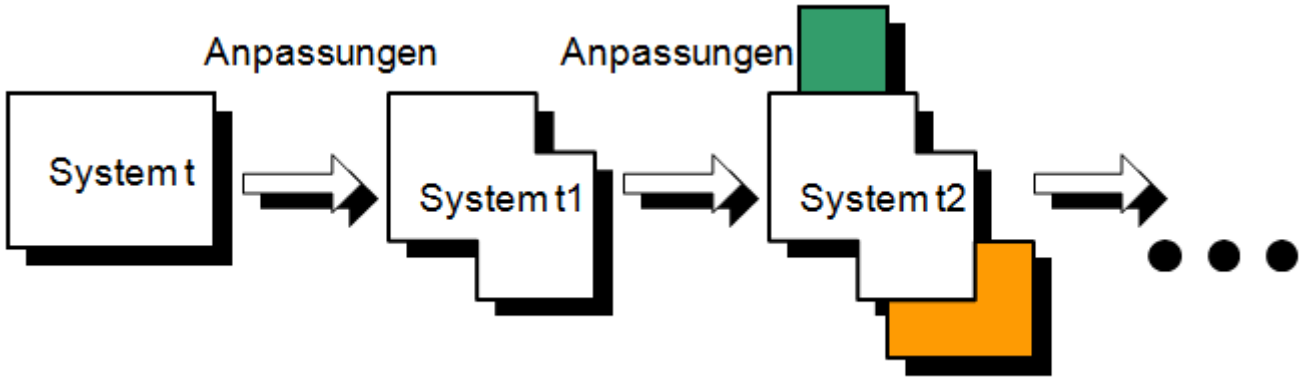
- NIST: „Software Errors Cost U.S. Economy $59.5 Billion Annually"

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

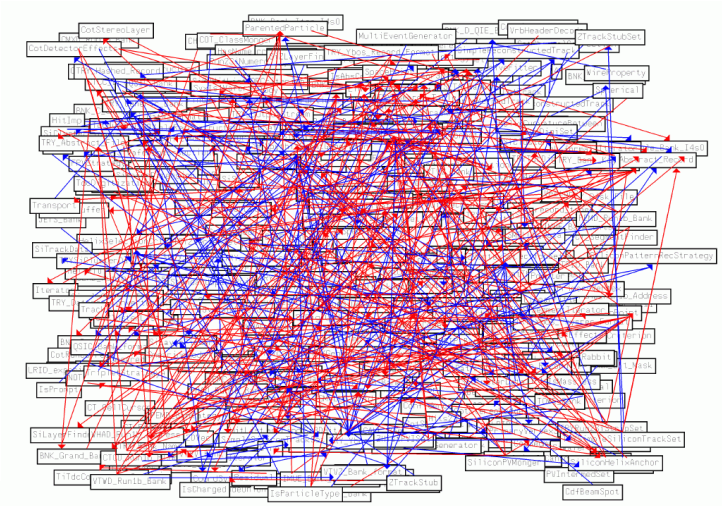
⇒ Softwarestrukturen degenerieren durch wiederholte Anpassungen

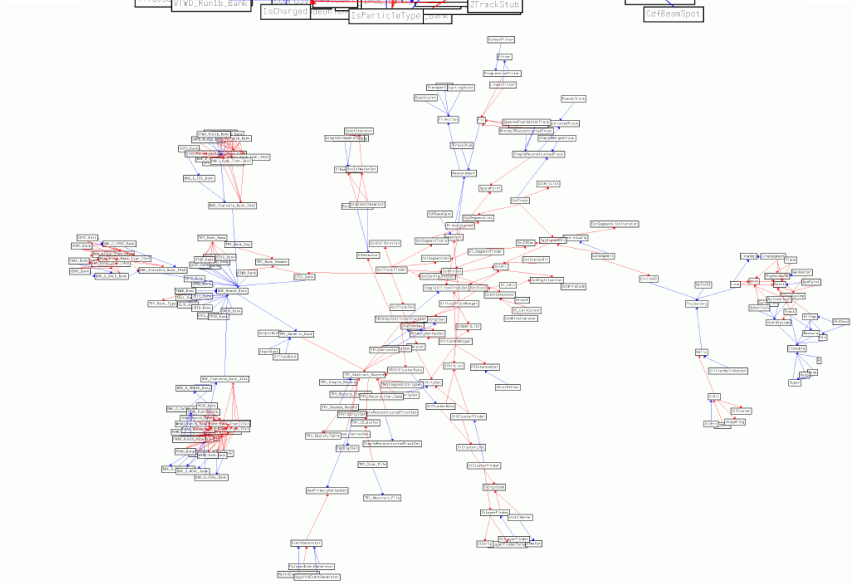


Grundprinzipien guter innerer Qualität

- **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

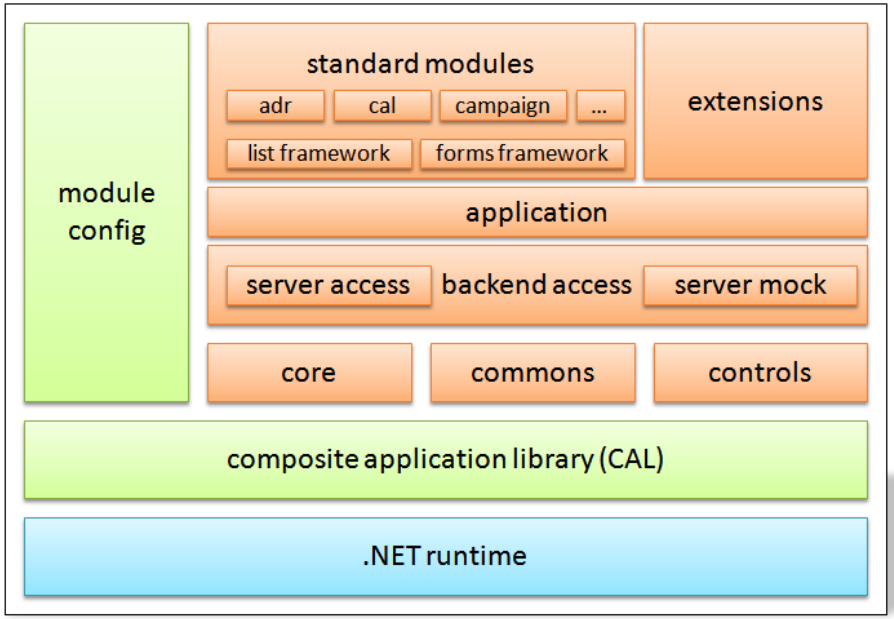
Implementierung

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

Beispiel einer modernen Architektur

Charakteristika

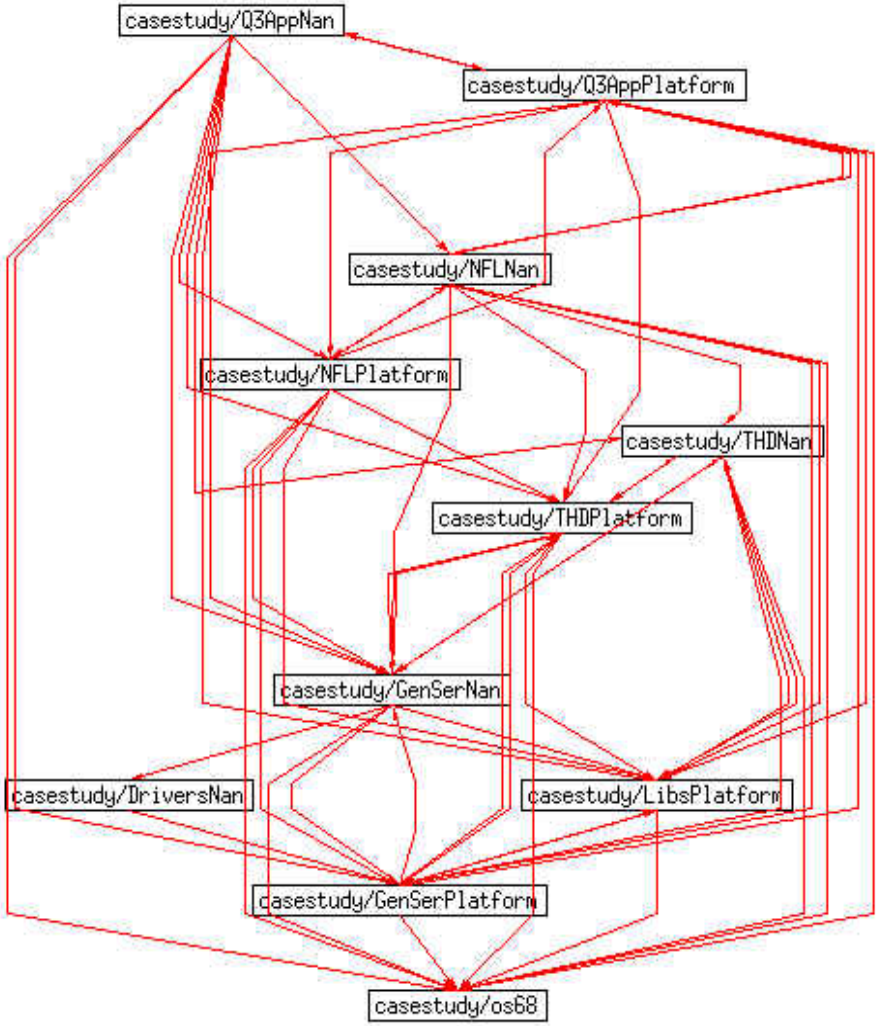
- Klare Schichtung
- Durchgängiges Modulkonzept, austauschbare Komponenten
- Inversion of Control, Dependency Injection
  - Betonung von Schnittstellen
  - Framework steuert die Modulabhängigkeiten: Dienstanbieter registriert sich beim Framework, Dienstabnehmer fragt Dienst beim Framework an.
  - Erst zur Laufzeit werden die Module aneinander gekoppelt



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:

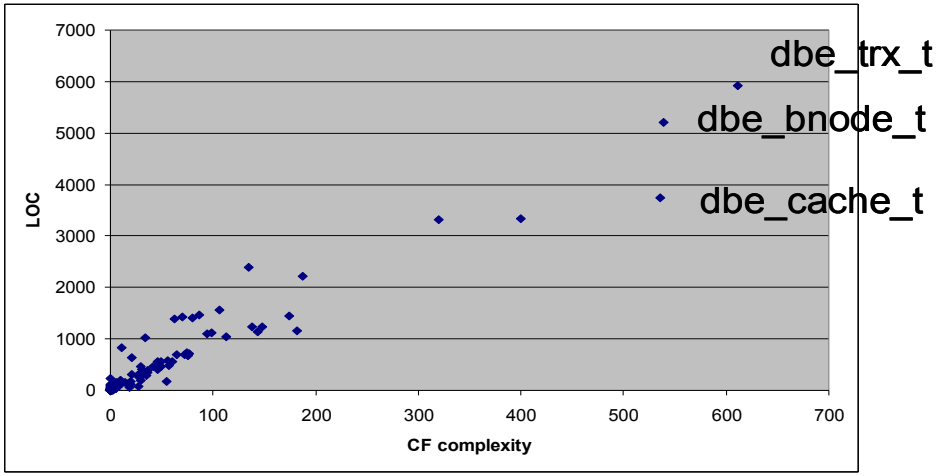
- Schichtung z.T. verletzt: LibsPlatform -> Q3AppPlatform
- Abbhängigkeiten von Plattformcode zum Produktcode: Q3Platform -> Q3AppNan

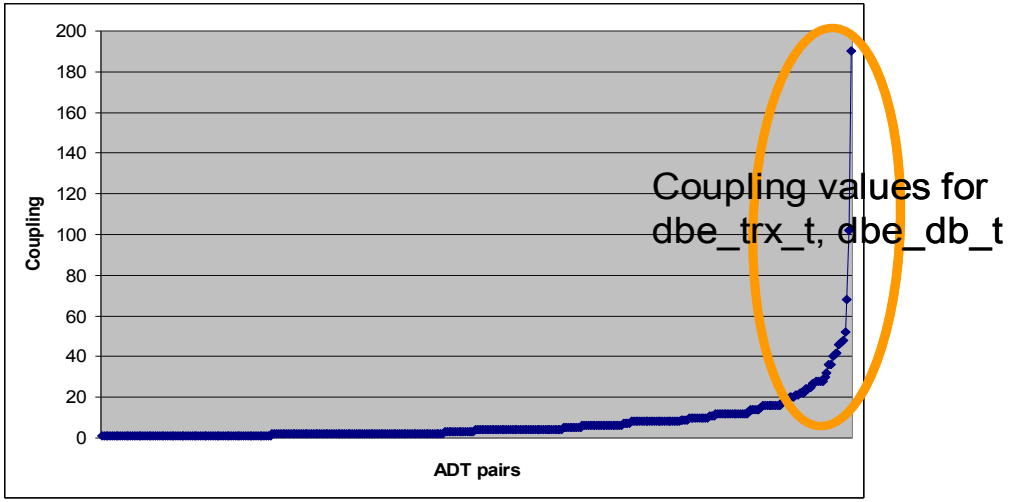
Korrektur von Abhängigkeiten:

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

Beispiel: Interpretation von Softwaremaßen

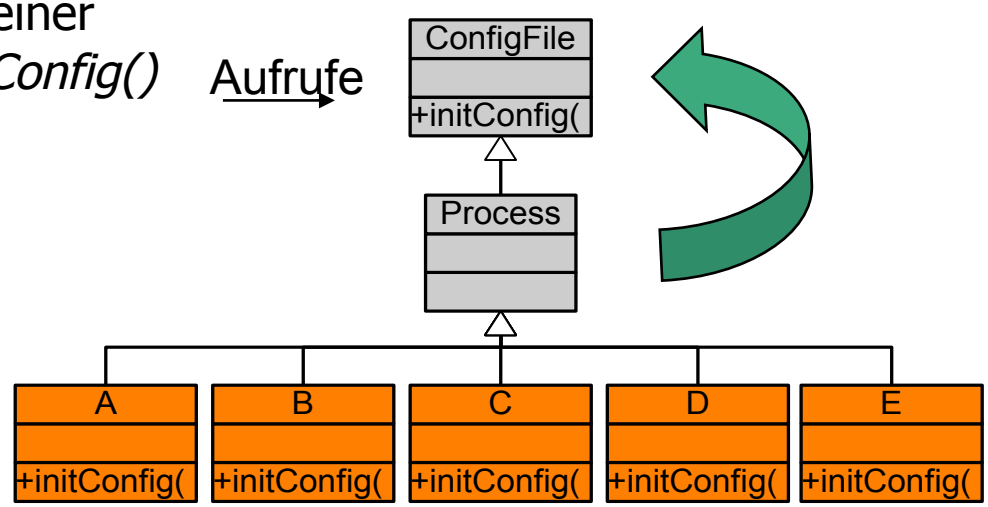
- Komplexitätsanalysen
  Nur wenige Datentypen sind sehr komplex
- Kopplung
  Nur wenige Paare von Datentypen mit hohen Kopplungswerten
- Probleme:
  - Einige Datentypen komplex und hoch mit anderen gekoppelt
  - Diese repräsentieren zentrale Konzepte; wahrscheinlich kann dies nicht verhindert werden





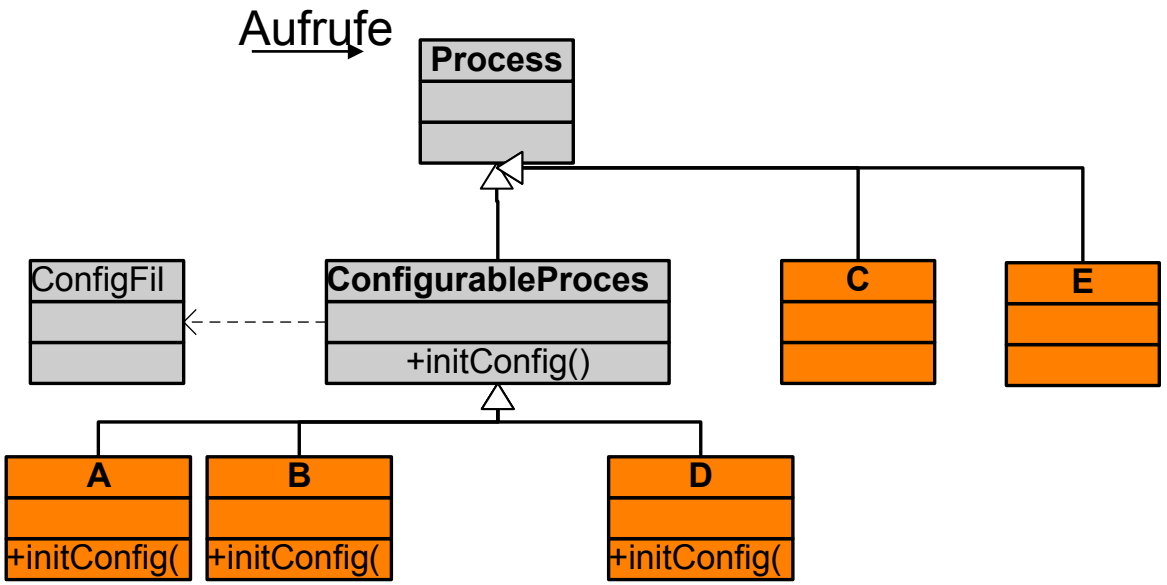
Beispiel: Musterbasierte Schwachstellensuche

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



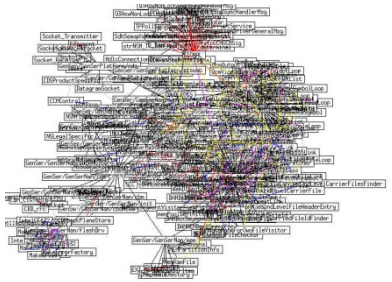
- Spezielle Prozesse *A* bis *E* überschreiben *initConfig():*
  - *C* und *E* : Implementierung von *initConfig()* ist leer
  - *A, B* und *E*: Implementierungen von *initConfig()* orientieren sich an der von *ConfigFile* (Codeduplikation!); zudem sind sie recht komplex.
- Viele Aufrufstellen verwenden 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



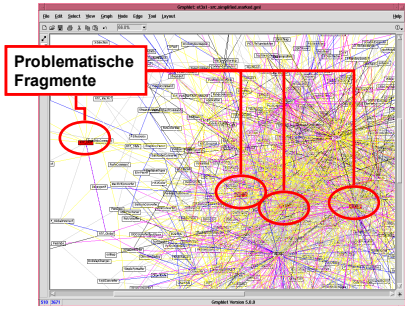
Visualisierung

Abstraktion

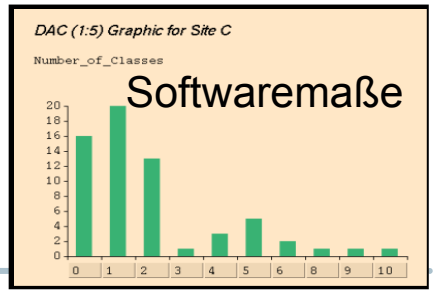
Quellcode

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!

Einstiegswege für Studierende und Absolventen

Gesuchte Fachrichtungen	Informatik, Wirtschaftsinformatik
Mögliche Tätigkeitsbereiche	Entwicklung, Forschung, Produktmanagement
Einstiegsmöglichkeiten	Praktikum, Abschlussarbeit, Werkstudententätigkeit, Direkteinstieg
Erwünschte Zusatzqualifikationen	Positive und kundenorientierte Denkweise, Pragmatismus
Kontakt	CAS Software AG Eva Erdl Human Resources Wilhelm-Schickard-Str. 8-12 76131 Karlsruhe Tel. 0721/96 38 -779 E-Mail: jobs@cas.de

Kontakt

Markus Bauer
Markus.Bauer@cas.de
CAS Software AG
Wilhelm-Schickard-Str. 8-12
76131 Karlsruhe