architektur SPICKER

Übersichten für die konzeptionelle Seite der Softwareentwicklung

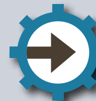
MEHR WISSEN IN KOMPAKTER FORM:
Weitere Architektur-Spicker gibt es als kostenfreies PDF unter www.architektur-spicker.de

NR. 2

embarc Software Consulting GmbH

SIGS DATACOM FACHINFORMATIONEN FÜR IT-PROFESSIONALS

Quantitative Analyse



Dieser Spicker führt in die Welt der quantitativen Analysen und zeigt wie Sie sie richtig einsetzen.

IN DIESER AUSGABE

- Bei welchen Qualitätsmerkmalen helfen quantitative Analysen weiter?
- Welche Tools und Metriken sind verbreitet und wie helfen sie Ihnen?
- Wie gehen Sie sinnvoll mit Ergebnissen um?

Worum geht's? (Herausforderungen)

- Über die Qualität Ihrer Software gibt es unterschiedliche Aussagen und Meinungen. Welche davon sind richtig?
- Sie haben Probleme mit Ihrer Software-Qualität, haben aber Schwierigkeiten, diese zu greifen
- Sie möchten ein unbekanntes System kennenlernen und dessen Verbesserungspotenziale erkennen

Quantitative Analysen geben einen unverfälschten/rationalen Blick auf Ihre Software.
Mit Tools erhalten Sie schnelle Ergebnisse. Doch wie können Sie diese positiv für Ihr Vorhaben nutzen?

Quantitative Analyse – Eine Definition

Mit quantitativen Analysen vermessen Sie die Artefakte Ihres Software-Systems. Am Häufigsten wird der Source Code betrachtet, weil dieser in unterschiedlichen Kontexten einer vergleichbaren Syntax folgt.

Es gibt sowohl statische Analysen, die nur die Artefakte als Eingabe benötigen, als auch dynamische Analysen, die das System zur Laufzeit untersuchen.

Quantitative Analyse im Überblick

Sie können quantitative Analysen für drei verschiedene Zwecke einsetzen:

Welche Eigenschaften einer Software lassen sich mit Tools bewerten?

Die ISO 25010 beschreibt die Qualitätsmerkmale einer Software. Nicht alle davon lassen sich mit quantitativen Analyse-Methoden bewerten.

Quantitative Analysen sehr gut geeignet – Andere Testmethode notwendig.

Tool-Übersicht

Eine Toolübersicht kann in diesem Bereich nie vollständig sein. Die aufgelisteten Tools begegnen uns in Projekten am häufigsten.

Tool	Plattform/ Sprachen	Integration	Verfügbar seit	Akt. Version	Stärken	Schwerpunkt	Support & Kosten
SonarQube	Java, C#, C/ C++, PL/SQL, Cobol, ABAP, JavaScript, etc.	IDE, CI	2009	6.0	Plattform bietet eine solide Basis, durch viele Plugins erweiterbar	Metriken	Freie Plattform, kostenpflichtige Plugins
→ http://www.sonarqube.org/							
Teamscale	Java, C/C++, C#, ABAP, Ada, PL/SQL, JavaScript und Python	IDE, VCS	2014	2.0	Historie wird gescannt, Feedback direkt nach dem Commit	Metriken, Strukturelle Analyse	2 Monate Testversion, danach Lizenzkosten user-basiert
→ https://www.cqse.eu/en/products/teamscale/landing/							
Structure101	Java, C# Plugins für C/ C++, PHP, SQL	IDE, CI	2006	4.2	Analysiert die Struktur ohne eine Vorgabe durch den Benutzer	Strukturelle Analyse	30 Tage Testversion, danach Lizenzkosten user- und serverbasiert
→ https://structure101.com/							
Sonargraph	Java, C#, C/C++	IDE, CI	2005	8.9	Strukturanalyse anhand von Package-Namen, zusätzlich Berechnung diverser Metriken	Metriken, Strukturelle Analyse	14 Tage Testversion, danach Lizenzkosten LOC-basiert
→ https://www.hello2morrow.com/products/sonargraph							
NDepend	C#	IDE, CI	2007	6.3	Strukturanalyse und Metriken; Vergleich von Baselines.	Strukturelle Analyse, Metriken	14 Tage Testversion, danach Lizenzkosten user- und serverbasiert
→ http://www.ndepend.com/							

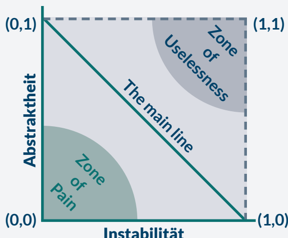
Abkürzungen: IDE = Integrated Development Environment | CI = Continuous Integration | VCS = Version Control System | LOC = Lines of Code

Stand August 2016

Verbreitete Metriken und deren Bedeutung

Die folgende Tabelle diskutiert populäre Metriken. Für alle Metriken gilt die Regel: Immer auch auf den Code dahinter schauen um die Ursache zu verstehen!

Metrik	Anwendungs-ebenen	Kurzdefinition – Was ist ein gutes Metrik-Ergebnis wirklich wert?	Typischer Korridor	Mögliche Gründe für Abweichungen
Zyklomatische Komplexität	Methode Klasse	*Anzahl möglicher Pfade durch den Code zur Laufzeit* Ein niedriger Wert ist gut, denn komplexe Elemente sind • schwierig zu verstehen • aufwändig zu testen	Max. 7-15 (Methode)	• Komplexe Fachlichkeit • Performante Algorithmen
Größe [in Zeilen]	Methode Klasse	*Umfang eines Elements* Ein niedriger Wert ist gut, denn große Elemente • sind wahrscheinlich komplex • verstoßen wahrscheinlich gegen das Single Responsibility Principle (SRP)	50-150 (Methode)	• Niedrige Komple-xität • Hohe Kohäsion
Testabdeckung	Beliebig	*Anteil des Codes, der in Tests durchlaufen wird* Hoher Wert hilft Vertrauen in die Software aufzu-bauen. Änderungen sind möglich bei vorhandenen, sinnvollen Tests. Die Metrik drückt leider nicht den Sinn der Tests aus. Variabilität von Tests wird ebenso nicht gemessen.	60%-90%	• Ausgereifte Continuous Delivery Prozesse • Gutes, feinmaschi-ges Monitoring in Produktion
Clone Coverage / Code Duplication	Projekt Komponente	*Anteil des Codes, der Kopien aus derselben Codebasis enthält.* Viele Kopien machen die Wartung von Software schwierig. Manche Frameworks erfordern jedoch das Schreiben von ähnlichen Code-Stellen („Boilerplate")	5%-15%	• Generierter Code • Lesbarkeit des Codes • Abstraktion schwierig
Paket Zyklen	Projekt Komponente	*In einem Paket-Zyklus ist ein Paket abhängig von sich selbst über beliebig viele andere Pakete.* Zyklen können Änderungen erschweren oder dazu führen, dass die Software nicht mehr gebaut werden kann, da poten-ziell viele andere Pakete betroffen sind.	Niedrig = gut Keine konkrete Vorgabe möglich	Keine
Cumulative/ Average Component Dependency (CCD/ACD)	Komponente	*Die Anzahl der mit einer Komponente direkt und indirekt verbundenen Komponenten.* Dies ist ein Indikator für Wartbarkeit, Testbarkeit und Verständlichkeit. Gemessen wird über alle betrachteten Komponenten um eine Metrik für die gesamte Software zu haben. Für die Betrachtung einzelner Komponenten siehe „Distance from the main line".	Niedrig = gut Keine konkrete Vorgabe möglich (NCCD = Vergleich mit balanciertem Binär-baum als Idealbild; Tools geben hier maximal 6,5 bis 10 vor)	*Bei Betrachtung aller Pakete:* • Komplexe Fachlichkeit *Bei Betrachtung einzelner Pakete: Das Paket enthält...* • Vermittler- • Service Locator- • Sandwiching- • ... Funktionalität
„Distance from the main line"	Paket	*Die „Distance from the main line" ist der Abstand eines Pakets zur Ideallinie (s. Grafik links) in Bezug auf Abstrakt-heit und Abhängigkeiten ausbalancierte Pakete. Abstrakt-heit: Anteil abstrakter Klassen und Schnittstellen an der Gesamtzahl der Klassen. Abhängigkeiten werden durch „Instabilität" bewertet: Anteil ausgehender Abhängigkei-ten an den gesamten Abhängigkeiten (I = Ce / (Ce + Ca)).* Ein Paket sollte ausbalanciert sein um Änderungen an ihm zu ermöglichen und gleichzeitig die Auswir-kungen richtig einschätzen zu können: • Bei vielen eingehenden Abhängigkeiten sollten die konkreten Implementierungen von den Verwen-dern durch Abstraktionen verborgen sein. • Bei wenig bis keinen eingehenden Abhängigkeiten sind konkrete Implementierungen zu empfehlen.	Möglichst gegen 0. Achtung wenn nah an „Zone of Uselessness" oder „Zone of Pain"."	„Useless" erscheinen häufig: • API-Packages • Generell: System-teile, die mit Fremdsystemen interagieren „Painful" erscheinen häufig: • Zentrale, oft verwendete Funktionalitäten • Fachlich sehr stabile Funktionalitäten



Typische Zielwerte müssen immer im eigenen Kontext angepasst werden!

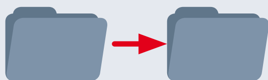
Prinzipien zum Umgang mit Metriken

- **Betrachte nur wenige Metriken.** Die Eingrenzung auf die 3-5 wichtigsten Metriken hilft den Überblick zu behalten.
- **Passe Zielwerte auf Deinen Kontext an.** Nicht „einfach so“ die Werte aus anderen Projekten übernehmen. Welche Werte sind für mich realistisch?
  1. Ziele klar festlegen und nicht nur Verbesserung in eine Richtung ausrufen.
  2. Zwischenziele zur Überprüfung einbauen und bei Erreichung entscheiden, ob noch weiter verbessert werden muss.
- **Automatisiere die Metrikauswertung.** Dies ermöglicht eine kontinuierliche Auswertung und liefert damit **Trends**, die beobachtet (und hinterfragt!) werden müssen.
- **Baue menschliche Bewertung in den Review-Prozess ein.** Überprüfe ob die Lösung auch verbessert wurde oder nur das Metrik-Ergebnis.
- **Ändere Code nicht nur, weil die Metrik dadurch verbessert wird.** Behebe Metrik-Schwächen nur an Stellen, an denen sowieso Änderungen durchzuführen sind (wegen Fehlern oder neuer Features).
- **Betrachte Metriken als Indikatoren.** Fehlalarme gibt es in jedem Bereich. Manchmal ist das Fixen teuer oder der Code nach dem Fixen nicht besser.
- **Hinterfrage regelmäßig die Prozess-Qualität.** Identifiziere die Ursache für das Problem. Mögliche Gründe: Zeitmangel, fehlendes Review, unklare Anforderung, etc.
- **Vermeide Pauschalisierungen.** Je konkreter ein Analyse-Ergebnis umso besser (z.B. nicht die Test Coverage aller Klassen anschauen, sondern sinnvoll einschränken auf die wichtigsten Module).

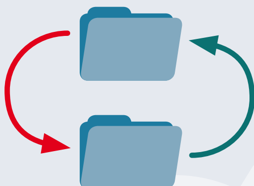
Strukturanalyse: Funktion und Möglichkeiten

Als Form der statischen Analyse hilft die Strukturanalyse den Aufbau einer Software zu verstehen und gegen ein Ziel-Modell abzugleichen. Anschließend können Änderungen eingeplant und durchgeführt werden, um die Struktur zu verbessern.

Identifizierte Probleme



Unerwünschte Abhängigkeit

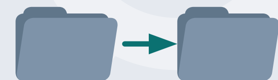


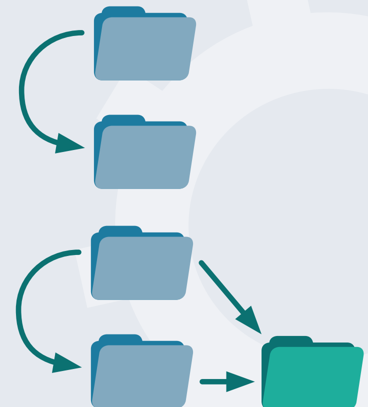
Zyklische Abhängigkeit

Möglicher Umgang

- Ausnahme definieren
- Ziel-Modell verändern
- Code verschieben
- Abstraktion oder Schnittstelle einführen (Dependency Inversion)
- Neue Komponente definieren, die Abstraktion und Schnittstelle enthält (Adapter)

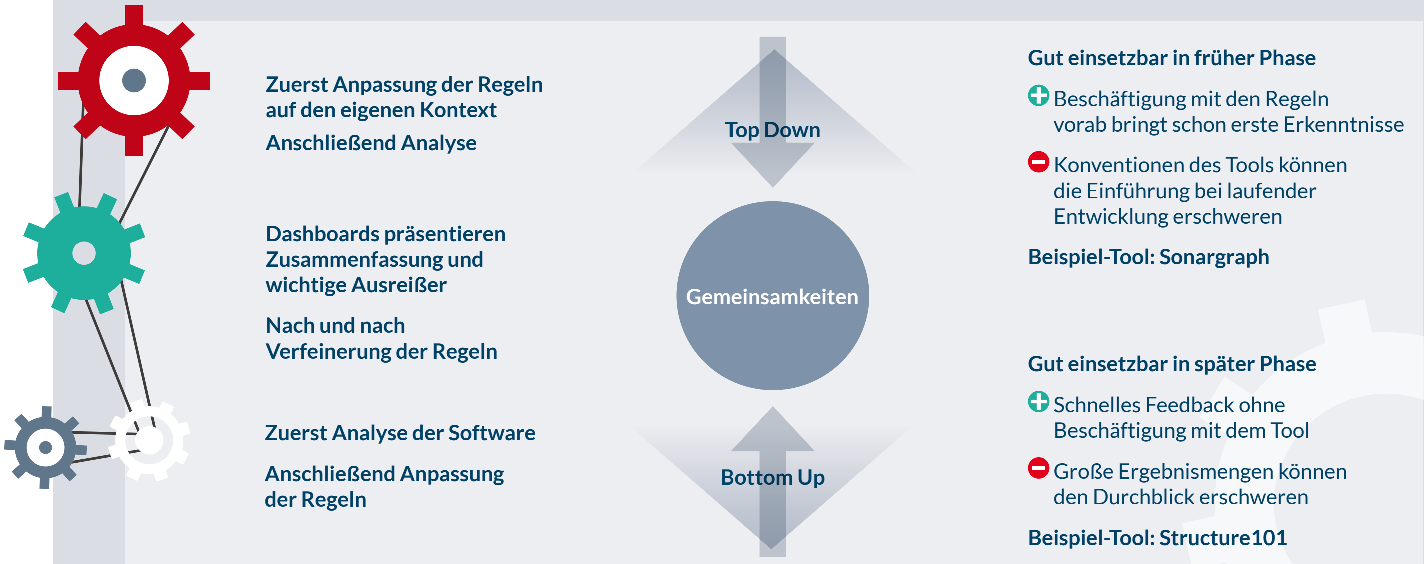
Ergebnis





Bottom-Up vs Top-Down Vorgehen

Top-Down- oder Bottom-Up-Vorgehen helfen in unterschiedlichen Situationen weiter. Die Abbildung zeigt Unterschiede und Gemeinsamkeiten der beiden Vorgehen sowie deren Vor- und Nachteile.



Dashboards zielgerichtet einsetzen

Die meisten Tools bieten eine komprimierte Aufbereitung der Analyse-Ergebnisse in Form sogenannter Dashboards an. Die folgenden Best Practices helfen Ihnen beim Einsatz von Dashboards.

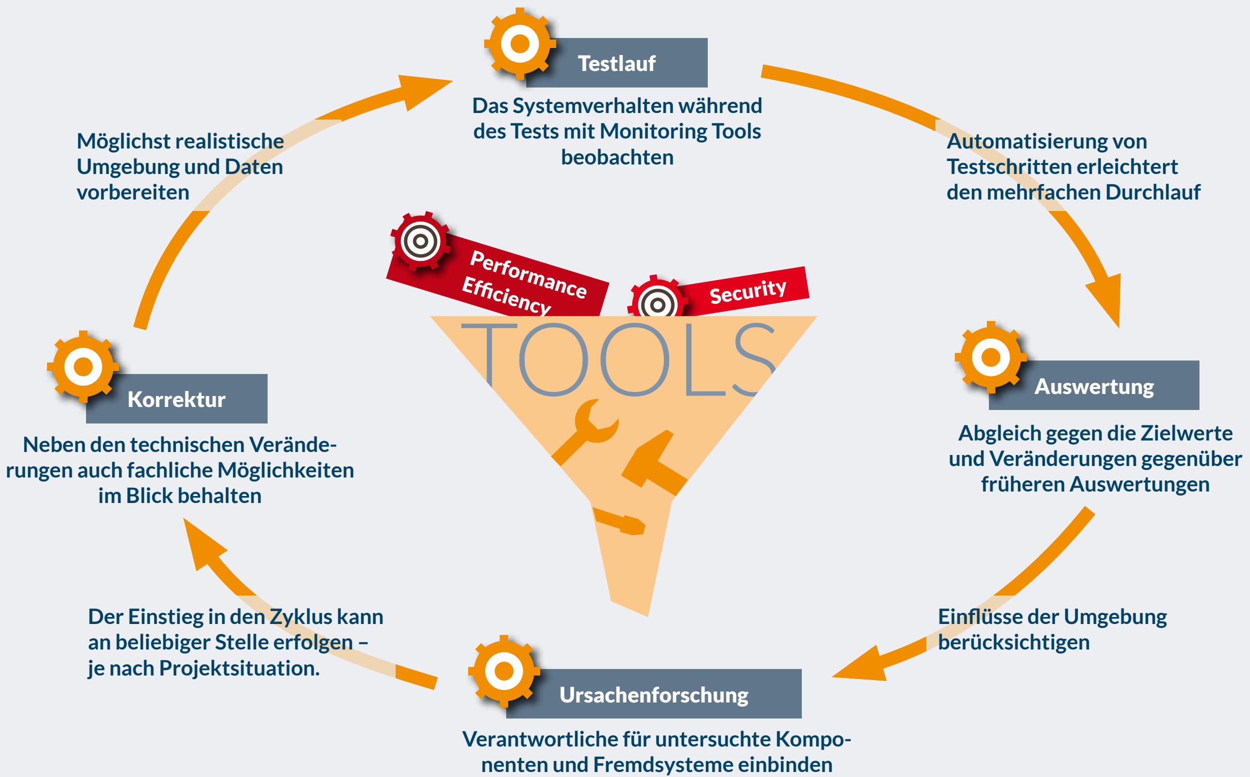
Vorgehen für direkte Zielüberprüfung

Die Erfüllung bestimmter Anforderungen kann direkt am laufenden System überprüft werden:

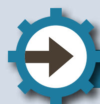
- Durch funktionale Tests: Funktionalität (Beispiel-Aspekte: Vollständigkeit, Korrektheit, Angemessenheit), Gebrauchstauglichkeit (Beispiel-Aspekte: Verwendbarkeit, Fehlertoleranz, Benutzerfreundlichkeit, Flexibilität)
- Durch Tools: Performance Efficiency, Security

Statische Analysen bieten häufig Abgleich gegen Anti-Patterns in den Bereichen Performance Efficiency und Security und damit Indikationen. Gewissheit über die Erreichung der Ziele bieten nur dynamische Analysen.

Das folgende iterative Vorgehensmodell hat sich für uns in verschiedenen Projekten bewährt.

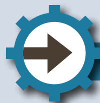


Weitere Informationen



Beispiele und weiterführende Informationen zu den aufgeführten Tools

- SonarQube Demo: http://nemo.sonarqube.org/
- Teamscale Demo: https://demo.teamscale.com/
- Structure 101 Tutorials: https://structure101.com/resources/#videos
- Sonargraph Webcasts: https://www.hello2morrow.com/products/sotoarc/tutorial
- NDepend Demo und Tutorial: http://www.ndepend.com/docs/getting-started-with-ndepend



Der Autor dieses Spickers

- Harm Gnoyke ist Softwareentwickler und -architekt bei embarc in Hamburg.
Kontakt: harm.gnoyke@embarc.de
Twitter: @HarmGnoyke

http://www.embarc.de
info@embarc.de

http://www.sigs-datacom.de
info@sigs-datacom.de