PERSPEKTIVEN FÜR UMWELT & GESELLSCHAFT umweltbundesamt

ANFORDERUNGEN AN DIE SOFTWAREQUALITÄT

Titel:	Anforderungen an die Softwarequalität
Dateiname:	Anforderungen an die Softwarequalitaet_v5
Version:	V 5.0
Datum:	06.12.2017
Autor(en):	Karin Schellner, Michael Hadrbolec, Thomas Baldauf

INHALTSVERZEICHNIS

Allgemeines

Unter Softwarequalität versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen (Ist/Soll). Diese Definition bezieht sich damit ausschließlich auf die Produktqualität und nicht die Prozessqualität. (Wikipedia)

Während die Eignung des Softwareprodukts für die in Anforderungsanalyse und Anwendungsentwurf definierten Erfordernisse als laufender Prozess in Entwicklung und Test sichergestellt werden muss, bezieht sich dieses Dokument vorrangig auf die durch formale Kriterien messbare Qualität der Architektur und des Codes der erstellten Software.

Grundsätzlich werden diese formalen Qualitätskriterien bei Projektstart (Projekt-Kickoff) festgelegt. Jedes Projekt hat vor Beginn einen QS Verantwortlichen für Software-Entwicklung (QSV-SW) festzulegen, dessen Rolle es ist, die Softwarequalität während der Entwicklung laufend zu überprüfen und gegebenenfalls Schritte zur Verbesserung einzuleiten. Der Projektleiter lässt sich regelmäßig vom QSV-SW über den aktuellen Stand der Softwarequalität in seinem Projekt berichten und ist verantwortlich für die Einhaltung der QS Kriterien in seinem Entwicklungsteam. Wenn die Kriterien nicht erreicht werden, wird der QSV-SW keine Freigabe für das Projekt geben und eine Nachbesserung der Software verlangen.

Teilaspekte der Softwarequalität können mittels Tools aufgrund von formalen Kriterien automatisch überprüft werden. Diese sind in den Nightly-Build Prozess des Hudson bzw. Jenkins Continuous Integration Servers integriert und stehen allen Beteiligten (Projektleitern, Entwicklern, Architekten, QSV-SW) laufend zur Verfügung. Der jeweilige QSV-SW eines Projekts muss diese formalen Kriterien definieren und ein automatisierter Prozess liest – bei in Entwicklung befindlichen Projekten – die aktuellen QS-Werte der jeweiligen Projekte wöchentlich aus und dokumentiert sie in den sogenannten QS-Reports.

Der QSV-SW ist angehalten, den QS Report zu interpretieren und zu bewerten, darüber im Projekt Jour Fix entsprechend Auskunft an alle Projektbeteiligten zu geben und gegebenenfalls Maßnahmen davon abzuleiten.

Qualitätskontrolle in der Software Architektur

Die Qualität der Architektur und insbesondere die Einhaltung der Architekturvorgaben sind Voraussetzung für spätere Wartung und Weiterentwicklung der Software. Darüber hinaus garantiert eine gute Architektur die Anpassbarkeit der Applikation an zukünftige Anforderungen und verlängert somit die Lebensdauer der Applikation.

Organisation

Jedem (Teil-)Projekt ist grundsätzlich ein Softwarearchitekt zugeteilt. Dieser erstellt die Softwarearchitektur und begleitet das Projekt während der Analyse- und Entwicklungsphase. Es ist anzuraten, den Architekten bereits während der Analysephase für grundlegende Architekturentscheidungen beizuziehen.

Es wird meistens Sinn machen, den oder einen der SW Architekten eines Projektes mit der Rolle des QSV-SW zu betrauen. In der Entwicklungsphase definiert dieser die QS Architekturkriterien für das vorliegende Projekt. Da für die Definition dieser Kriterien das Tool Sonargraph verwendet werden soll und aus Qualifikations- und Lizenzgründen nur wenige Personen am Umweltbundesamt damit vertraut sind, soll eine solche Person beim Prozess der Kriteriendefinition hinzugezogen werden.

Sonargraph erlaubt die regelmäßige Überprüfung dieser Kriterien. Die Ergebnisse dieser QS Prüfung dienen als Indikator für den SW Architekten und sollen in den regelmäßigen Projekt-Meetings (Jour Fixe) besprochen werden. Gegebenenfalls sind Maßnahmen und Aufgaben zu definieren, um die Einhaltung der vorgegebenen Architektur gewährleisten zu können.

Tools

Sonargraph

Sonargraph ist ein Architekturdefinitions- und Überwachungswerkzeug für mittlere bis große Java Projekte, das es ermöglicht, die Einhaltung festgelegter Architekturvorgaben kostengünstig und effizient zu garantieren. Sonargraph lässt sich in gängige Entwicklungswerkzeuge wie Eclipse, ANT und Maven integrieren.

Ziel des Produkts ist es, Architekturverletzungen bereits zum Zeitpunkt der Entstehung zu entdecken und zu verhindern. Sonargraph ermöglicht es, jeden Entwickler mit den nötigen Analyse- und Kontrollfeatures auszustatten, ohne dabei einen größeren Overhead zu verursachen.

Zusätzlich kann Sonargraph Reports und Metriken erzeugen, die zur Beurteilung der Architektur und der Softwarequalität dienen. Diese Reports werden täglich mit den Nightly-builds am Hudson bzw. Jenkins Continuous Integration Server erstellt.

In der aktuellen Version von Sonargraph wurde auf eine DSL (Domain Specific Language) umgestellt und gleichzeitig wird die Architektur nun nicht mehr für jedes Artefakt (api, web, …) einzeln definiert und geprüft, sondern es gibt pro Projekt eine übergreifende Architektur-Definition. Somit können auch Verletzungen zwischen den Artefakten erkannt werden.
Hinweis: Da im Tool Sonar Metriken (und somit auch die Sonargraph Architektur-Metriken) im-

mer auf Artefakt-Ebene angezeigt werden, jedoch Sonargraph nun Artefakt-übergreifende Metrik produziert, kann es nun vorkommen, dass angezeigte Architektur-Verletzungen in einem der anderen Artefakte als dem gerade angezeigten zu suchen sind!

Details zu Metriken der Software Struktur sind im Anhang B: Details zu Metriken/Kriterien in der Software Entwicklung zu finden.

Qualitätskontrolle bei der Implementierung

Die Qualität des Codes und die Einhaltung der Namens- und Code-Konventionen sind essentiell für die Verständlichkeit und Lesbarkeit des Codes und beeinflussen daher ganz unmittelbar die Aufwände für Wartung und Weiterentwicklung der Software. Des Weiteren kann nur bei ausreichender Testabdeckung sichergestellt werden, dass Codeänderungen keine unerwünschten Nebeneffekte haben.

Organisation

Für jedes Entwicklungsprojekt wird für die Implementierungsphase ein QSV-SW zur Seite gestellt. Häufig wird diese Rolle der SW-Architekt übernehmen. Der QSV-SW legt mit dem technischen Projektleiter die QS Kriterien, die im Normalfall den Standardvorgaben entsprechen sollten, fest, dokumentiert sie im Anwendungsentwurf und überprüft in regelmäßigen Abständen (ca. alle 2 Wochen) die Einhaltung derselben. Im Zuge dieser Überprüfung bewertet er den **QS Report**, der die Ergebnisse aus den formalen Kriterien zusammenfasst, Trends darstellt und laufend aktualisiert wird. Bei Abweichungen werden gemeinsam mit dem Entwicklungsteam Verbesserungsmaßnahmen vereinbart.

Genaue Details zu den einzuhaltenden Prozessen finden sich im Kapitel „Einzuhaltende QS Prozesse und Regeln“.

Tools

Es gibt bereits viele Tools, die nach entsprechender Konfiguration, automatisiert das Software Projekt bzw. den Code auf Einhaltung der konfigurierten Qualitätskriterien überprüfen und entsprechende Reports erstellen. Diese Tools erlauben auch häufig die übersichtliche Darstellung der Ergebnisse in Form von Diagrammen und das Drill-Down auf z.B. betroffene Programmzeilen.

Im Umweltbundesamt wird für die Qualitätssicherung der Implementierung die „Open Source Quality Management Plattform“ ***SonarQube*** (http://www.sonarqube.org) verwendet, die darauf ausgelegt ist kontinuierlich die Qualität des Source Codes zu analysieren und zu bewerten.

SonarQube wird vom Hudson bzw. Jenkins CI (Continuous Integration) Server im Rahmen der Nightly-Builds angestoßen und damit stehen aktuelle QS Ergebnisberichte täglich zur Verfügung.

Achtung: Wenn die Projekte am Hudson bzw. Jenkins CI Server nicht erstellt werden können, dann laufen die QS Tests nicht und die QS Ergebnisberichte werden auch nicht erzeugt. Da diese Ergebnisberichte von den Architekten und QSV-SW benötigt werden, ist für funktionierende Builds zu sorgen. (Die Verantwortung dafür trägt der Entwicklungsteamleiter.)

SonarQube

SonarQube hat mittlerweile eine eigene Rule engine für Java und verwendet aber auch einige vorhandene bekannte Libraries zur Qualitätssicherung, wie z.B. Checkstyle, PMD

Weiters erlaubt SonarQube die Integration von Sonargraph und ermöglicht die Darstellung der Ergebnisse der Sonargraph Überprüfung im SonarQube Dashboard.

SonarQube bietet die Möglichkeit verschiedene Quality Profiles zu erstellen, die die für bestimmte Projekte sinnvollen Vorgaben (Rules) beinhalten. Für die Java Software Projekte im Umweltbundesamt soll das Profil „UBA Vorgabe“ verwendet werden. Dieses Profil beinhält ca. 570 Coding Rules, die von den SonarQube Plugins für Java, Checkstyle, PMD und Sonargraph stammen. Im Moment konzentriert sich das Profil „UBA Vorgabe“ hauptsächlich auf Regelverletzungen des Schweregrads „blocker“, „critical“, „major“ mit einigen wenigen Regeln mit Severity „minor“ und „info“.

Alle im Qualitätsprofil definierten Regeln können am Umweltbundesamt Server devtoolprod unter diesem Link:

https://devtoolprod.umweltbundesamt.at/sonar/profiles/show?key=java-uba-vorgabe-72609 angesehen werden.

Checkstyle

Checkstyle ermöglicht es, die Java Coding Standards zu überprüfen. Bei neuen Projekten ist zumindest die im SonarQube Qualitätsprofil „UBA Vorgabe“ definierte Konfiguration bindend. Für jede Konfiguration gibt es auch eine Entsprechung für Eclipse, die das Formatieren des Codes erleichtert (siehe Uclipse-Installation = Eclipse Installation fürs Umweltbundesamt)

Bei den Checkstyle Coding Rules wird z.B. auf die Existenz von nicht verwendeten Import-Statements (Unused Imports) geprüft, oder kontrolliert, ob bei überschriebener equals() Methode auch die hashCode() Methode überschrieben wurde.

PMD Report

PMD überprüft den Java Source Code und entdeckt potentielle Probleme:

- Mögliche Bugs - leere try/catch/finally/switch Bedingungen
- “Dead Code” - ungenutzte lokale Variablen, Parameter und private Methoden
- Suboptimaler Code – ineffiziente Verwendung von String/StringBuffer
- Verkomplizierte Ausdrücke - unnötige if-Bedingungen, for-Schleifen, die durch while-Schleifen ersetzt werden könnten

Einzuhaltende QS Prozesse und Regeln

QS Prozess

- Für jedes Entwicklungsprojekt wird ein **QSV-SW definiert**, diese Rolle kann z.B. der SW Architekt übernehmen
- QS Ziele werden vor Projektstart vereinbart und Aufwände geschätzt (TPV-IT[1], SWA, ETL, TGL)
- Die Dokumentation der QS Vorgaben ist im Anwendungsentwurf vorzunehmen und an alle Projektbeteiligten zu kommunizieren (SWA).
- Der Status zur QS Ziel-Erreichung soll vom Entwicklungsteamleiter bzw. SW Architekten (SWA) im Projekt Jour Fix regelmäßig präsentiert werden. Der automatisch aus den SonarQube Metriken generierte **QS Report** soll dabei vom QSV-SW bewertet und kommentiert werden.
- Dieser **QS Report** soll die Einhaltung der formalen Kriterien beleuchten, **Trends** darstellen und wird laufend aktualisiert, um die Verbesserung der Codequalität nachvollziehbar zu machen.
- Durchführung von Code Reviews
- Bei **Abweichungen** werden gemeinsam mit dem Entwicklungsteam **Verbesserungsmaßnahmen** vereinbart.

Werden die vereinbarten Kriterien grob verletzt oder Ziele nicht eingehalten, erfolgt keine **Freigabe aus QS Sicht**.

QS und Test

- Allgemein gilt, dass QS Regeln 1-4 (siehe QS Regeln) zu den Themen Code, Architektur Verletzungen und Code Coverage **vor Teststart** (ETL) für die jeweiligen zu testenden Module weitestgehend erfüllt sein müssen.
- In einer Projektsitzung ca. 2 Wochen vor geplantem Testbeginn ist "Code Complete" der an den Test zu übergebenden Module in Hinblick auf SW Quality Gates einzuschätzen. Fällt diese Einschätzung negativ aus, müssen entsprechende Schritte eingeleitet werden (Verschiebung des Testbeginns, etc.)
- Prinzipiell werden QS-Fehler mit entsprechenden Fehlerklassen im Zuge des Tests durch die Test-Abteilung als Bugs erfasst, siehe Anhang D: QS Regeln und Fehlerklassen.
- Werden während der Entwicklung bereits einzelne Module getestet, ist die SonarQube Metrik, die sich auf das gesamte Projekt/Artefakt bezieht, noch nicht entsprechend aussagekräftig und es erfolgt noch keine Erfassung im Bugtracker-Tool.
    - In diesen Fällen muss der Code dieser Module manuell, stichprobenartig durch SW Architekten und Entwicklungsteamleiter (ETL) auf ausreichende Erfüllung hin überprüft und eingeschätzt werden.
    - Der QSV-SW (SWA) dokumentiert diese Einschätzung im QS-Report (anhand eines eigenen Excel-Tabellenblatts pro Modul)

[1] Eine Rolle, die unterstrichen dargestellt ist, bedeutet, dass diese Rolle sich um die Ausführung dieser Tätigkeit kümmert.

- QS-Fehler werden erst bei Übergabe des letzten Moduls eines Projekts an den Test im Bugtracker-Tool erfasst
- Dafür wird das QS-Ergebnis für die einzelnen Module durch den SW Architekten (SWA) und Entwicklungsteamleiter vor jedem Testbeginn bewertet und mit dem Testverantwortlichen und dem Projektleiter abgestimmt und im QS-Report festgehalten (TPV-IT).

QS Regeln

Qualitätssichernde Regeln, die laufend (mit jedem Nightly-Build) von dem Tool SonarQube auf den Projektcode angewandt werden und deren Einhaltung vom QSV-SW alle 2 Wochen überprüft und im QS Report zusammengefasst werden sollen:

1) Es darf keine Verletzungen der im Qualitätsprofil „UBA Vorgabe“ definierten Regeln (Rules) der Kategorien „Blocker“ und „Critical“ geben. (Regelverletzungen aus den Kategorien „Major“, „Minor“ und „Info“ dienen als Information und sind geduldet.)
   - ⇨ Violations: Verletzungen von Regeln der Kategorien „Blocker“ und „Critical“ = 0

2) Einhaltung der Architekturvorgaben, im speziellen Vermeidung von Zyklen, Kopplung.
   - ⇨ Sonargraph Architecture Violations = 0 (werden als Blocker angeführt)
   - ⇨ package cycle groups, cyclic packages = 0
   - ⇨ unassigned types bzw. components = 0
     (Gibt es unassigned types bzw. components, muss die Sonargraph Konfiguration dieses Projekts entsprechend aktualisiert/erweitert werden!)

3) Verwendung von JUnit Tests mit hoher Testabdeckung
   - ⇨ 80% Code Coverage (bei api packages)
   - ⇨ Ausgenommen werden dürfen, z.B. model classes, mappings, container objects und generierter Code

4) Kein doppelter Code
   - ⇨ < 5% Duplications
   - ⇨ Konkrete Code Duplications müssen vom QSV-SW überprüft werden:
     Doppelter Code im Sinne von kopierten Codeblöcken muss komplett vermieden werden. Die Toleranzgrenze von bis zu 5% bei den Duplications resultiert aus der Tatsache, dass sich z.B. Methoden wie

```
public String getGLN() {
return this.gln;
}
```

     in verschiedenen Klassen durchaus wiederholen können.

Qualtitätssichernde Maßnahmen und Regeln, die durch die Durchführung von Code Reviews überprüft werden müssen:

- Klassendesign (z.B. Cohesion und Coupling)

- Verwendung von Interfaces
- Lesbarkeit des Codes (z.B. Klassen- und Methodengröße)
- Komplexe Algorithmen (punktuell auch fachlich, wenn es erforderlich ist, z.B. häufige Fehler in dem Bereich)
- Querschnittthemen: Logging, Aspekte, Security, etc.
- Unit Testfälle
- Qualität der Dokumentation (Javadoc)

    - ⇨ Vollständige Dokumentation aller Packages, Klassen und aller öffentlichen Methoden (public methods) ausgenommen einfache setter/getter Methoden
    - ⇨ Dokumentation aller anderen Methoden, wenn sie nicht ohnehin selbsterklärend sind
    - ⇨ Dokumentation von komplexen Codeteilen, deren Nachvollziehbarkeit schwierig ist
    - ⇨ Dokumentation von Feldern, deren Bedeutung oder Wertannahme nicht auf einen Blick ersichtlich ist
    - ⇨ Kommentare sind grundsätzlich in englischer Sprache zu verfassen

Anhang A: Qualitätsüberprüfung mit SonarQube

Jedes Projekt, sollte am Hudson bzw. Jenkins CI einen FULL-Job konfiguriert haben, der für jedes Artefakt (API, Web, …) den Sonarqube Aufruf beinhält; dadurch , scheinen die Artefakte des Projekts dann in der SonarQube Projektliste auf:

https://devtoolprod.umweltbundesamt.at/sonarJedes Projekt hat eine Dashboard- Ansicht mit Details .

Hier erhält man eine Zusammenfassung der QS Ergebnisse (siehe Abbildung 1).

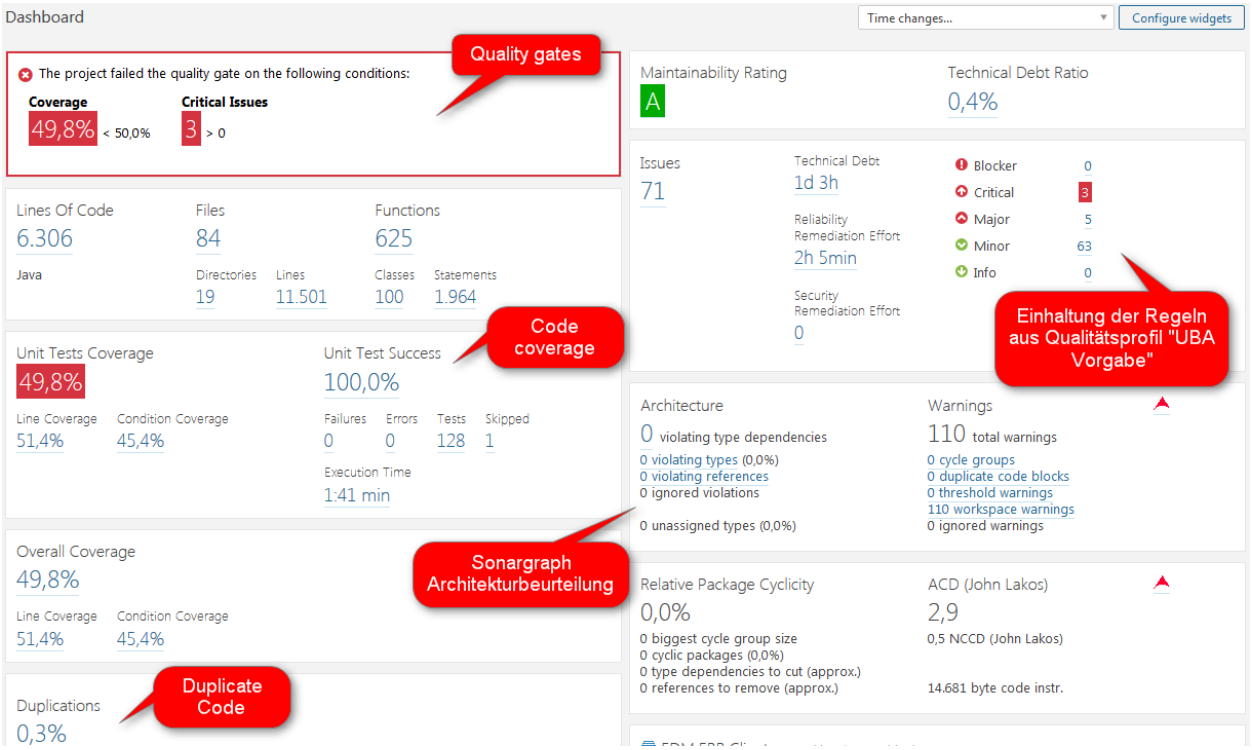


Abbildung 1: SonarQube Dashboard für ein bestimmtes SW Projekt

Die einzelnen Blöcke des Dashboards behandeln Gruppen von Metriken und Kriterien für die Codequalität des betreffenden Projekts. Durch Drill-Down kann man sich die Details zu den einzelnen Metriken und Kriterien bis auf die Ebene der einzelnen Codezeilen hinunter anzeigen lassen.

Die in jedem Fall vom QSV-SW zu prüfenden Aspekte sind in Kapitel „QS Regeln“ im Detail aufgelistet und inkludieren u.a.:

- Einhaltung der im Qualitätsprofil „UBA Vorgabe“ definierten Rules
- Sonargraph Verletzungen (Zyklen, Kopplungsgrad)
- Duplications
- Code coverage und Test success

Abbildung 2 zeigt die Detailansicht nach dem Drill-Down in die Verletzung der Regel „Catch Exception instead of Throwable“. SonarQube ermöglicht es, bis zur entsprechenden Codezeile zu gelangen.

Abbildung 2: SonarQube – Detailansicht einer „Violation“ nach Drill-Down

Achtung: Die in Sonarqube eingebetteten Metriken zu Sonargraph (=Architekturbeurteilung) sind ab Sonargraph Version 8 nicht mehr nur für das spezielle Artefakt (z.B. API oder Web), sondern für das gesamte Java-Projekt zu sehen. D.h. Architekturverletzungen, die in einem Web-Artefakt angezeigt werden, könnten sich in einem anderen (z.B. dem API) Artefakt befinden!

Anhang B: Details zu Metriken/Kriterien in der Software Entwicklung

Einige der wichtigsten Punkte für die Gewährleistung von Codequalität in Softwareprojekten sind:

1. Jedes nicht triviale Projekt benötigt eine klar definierte zyklenfreie logische Architektur, die zumindest die Schichten und wenn möglich auch die fachlichen Komponenten der Anwendung und die erlaubten Abhängigkeiten zwischen diesen definiert. Der Anwendungs-Code muss diese Struktur ohne Verletzungen reflektieren (Verletzung = regelwidrige Abhängigkeit).
2. Zyklen zwischen Packages sind nicht erlaubt (Zyklen auf höheren Ebenen werden durch Regel 1 ausgeschlossen).
3. Der Kopplungsgrad ist niedrig zu halten (siehe Abschnitt über CCD, NCCD, ACS und rACD).
4. Codezeilen in einer Komponenten sollten einen gewisse Anzahl nicht überschreiten (z.B. # Codezeilen pro Komponente < 700)
5. Zyklomatische Komplexität von Methoden sollte gering gehalten werden (z.B. < 25)
6. Vermeidung von doppeltem Code
7. Vollständige Dokumentation
8. Hohe Testabdeckung

Verwendung von Interfaces

Ein wichtiges Designprinzip liegt darin, im Abhängigkeitsgraphen unten liegende Knoten möglichst abstrakt zu halten, d.h. man sollte dort mit Interfaces anstatt mit konkreten Klassen arbeiten. Die Motivation dafür besteht einfach darin, dass Interfaces in aller Regel erheblich stabiler als Klassen sind und außerdem zu einer Verringerung des Kopplungsgrades beitragen. Die erfolgreiche Umsetzung dieses Prinzips lässt sich recht leicht mithilfe der weiter unten definierten "Distance"-Metrik überprüfen.

Vermeidung von Zyklen

Neben der Erhöhung des Kopplungsgrades haben Zyklen folgende negative Auswirkungen zur Folge:

- Zyklenteilnehmer können nicht mehr getrennt voneinander getestet werden. Damit wird das Erreichen einer guten Testabdeckung erheblich schwieriger, da man immer alle Zyklenteilnehmer berücksichtigen muss.
- Das Codeverständnis wird erschwert, da man tendenziell alle Zyklenteilnehmer gemeinsam verstehen muss. Die zyklische Abhängigkeit verhindert, dass man sich von unten nach oben hocharbeiten kann.
- Zyklen erhöhen die Komplexität.
- Auch für den Betrieb sind Zyklen potenziell unangenehm, insbesondere wenn sie mehrere fachliche Komponenten umfassen. Denn in diesem Fall kann keine der Komponenten ohne die anderen in Betrieb genommen werden. Für eine gute logische Architektur empfiehlt sich daher folgende Designregel:

  Zyklische Abhängigkeiten sind ab der Package-Ebene aufwärts verboten.

Kriterien/Metriken

Primäres Kriterium bei der Beurteilung der tatsächlichen (physischen) Software-Struktur einer Applikation ist die Einhaltung der vom zuständigen Architekten entworfenen logischen Architektur.

⇨ **Ziel: Anzahl Architekturverletzungen = 0**

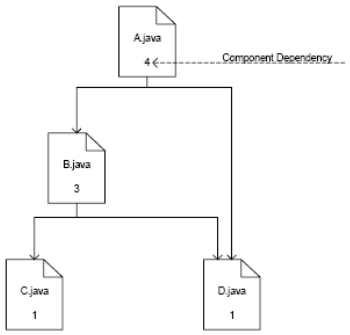
Allgemein versucht eine gute Software-Architektur (skalierbar, wiederverwendbar, testbar, robust)

- die Anzahl der Komponenten n zu maximieren
- den Kopplungsgrad der Komponenten (CCD) zu minimieren
- und die Abstraktion (A) von Komponenten tieferer Schichten zu maximieren

Cumulative Component Dependency (CCD)

- Summe aller Komponenten (= Compilation Units n) von denen eine Komponente direkt und indirekt abhängt, kumuliert über alle Komponenten des Strukturelementes
- Jede Komponente hat einen CCD von 1 + die Anzahl der eigenen Kinder bis zu den Blättern
- Zyklen => CCD → n^2 bei n Komponenten

⇨ **Ziel: CCD = n * log(n)**



Normalized Cumulative Component Dependency (NCCD)

- Verhältnis CCD (Cumulative Component Dependency) eines Subsystems mit n Komponenten zum CCD eines vergleichbaren balancierten Binärbaums

  < 1 horizontal (unabhängige Komponenten)

  1 balancierter Binärbaum

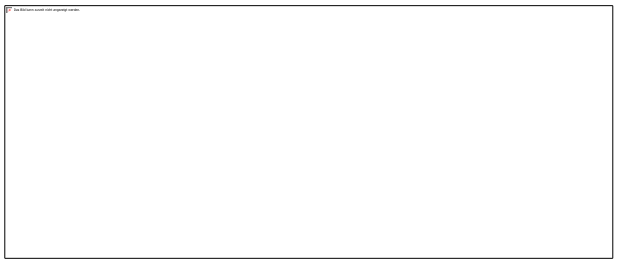
  > 1 zyklische Abhängigkeiten

⇨ **Ziel: NCCD = 1**

Average Component Dependency (ACD)

- durchschnittlicher CCD eines Subsystems
- ACD = CCD / Anzahl Komponenten
- Indikator für Anzahl Komponenten, die im Durchschnitt für einen Change Request geändert / getestet werden müssen

⇨ **Ziel: ACD = log(n)**



Relative Average Component Dependency (rACD)

- Von relativem ACD (rACD) spricht man, wenn der ACD selbst noch einmal durch die Anzahl der Knoten dividiert wird, um einen vergleichbaren Wert zu bekommen.
- Der Kopplungsgrad sollte immer deutlich langsamer als die Anzahl der Komponenten wachsen:

$$rACD_{max} <= 1{,}5 * \frac{1}{2^{\log_2(n)}} \quad \text{(für } n >= 200\text{)}$$

n ist dabei die Anzahl der Komponenten und sollte bei dieser Näherung mindestens bei 200 liegen.

⇨ **Ziel:** **n > 200: rACD < 15 %**
n > 1.000: rACD < 7,5 %
n > 5.000: rACD < 4 %.

Efferent coupling (CE)

- Anzahl der Typen außerhalb der Komponente, von der Typen innerhalb der Komponente abhängig sind (CE → 0 „Unabhängigkeit“).

Afferent coupling (CA)

- Anzahl der Typen außerhalb der Komponente, die von Typen innerhalb der Komponente abhängig sind (CA → 0 „keine Verantwortung“).

Abstractness (A)

- Anzahl abstrakter Typen dividiert durch die Gesamtanzahl der Typen in einer Komponente.

Instability (I)

- I = CE / (CA + CE)
- Je größer die Abhängigkeit von anderen Komponenten und je geringer die Verwendung in anderen Komponenten, desto größer die Instabilität.

Distance (D)

- D = A + I – 1
- Die Distanz wächst mit der Instabilität (Abhängigkeit von anderen Komponenten) und der Abstraktheit (Verwendung von Interfaces) einer Komponente.

Cyclomatic Complexity (CC)

- Die zyklomatische Komplexität bezeichnet die Anzahl möglicher Ablaufpfade in einer Methode. Diese Anzahl entspricht wiederum der Anzahl der Testfälle, die für eine hundertprozentige Testabdeckung benötigt werden.

  ⇨ **Ziel: CC < 25**

Anhang C: JavaDoc Sun Conventions

Es sollten möglichst die Javadoc Sun Conventions (How to Write Doc Comments for Javadoc) verwendet werden, z.B.:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Anhang D: QS Regeln und Fehlerklassen

QS Regel	QS Ziel	Ergebnis	Fehlerklasse
QS Regel 1: Code Violations	Verletzungen von Regeln der Kategorien „Blocker" und „Critical" = 0	> 0	1
QS Regel 2: Architektur Verletzung	SonarGraph Architecture Violations = 0 (werden als Blocker angeführt)	> 0	1
	Package cycle groups, cyclic packages = 0		
	Unassigned types bzw. components = 0		
QS Regel 3: JUnit Testabdeckung	80% Code Coverage (nur bei API packages!) - Unit Tests Coverage	75-79,9%	4
		70-74,9%	3
		50-69,9%	2
		< 50%	1
	100% success rate - Unit test success	<100%	1
QS Regel 4: Duplication	< 5% Duplications	5-6%	4
		6,1-9,9 %	3
		10-19,9%	2
		>= 20%	1

Anmerkung: Die bisherige QS Regel 5 für Javadoc („Public documented API: mindestens 95%") ist in den neuen Sonarqube Versionen nicht mehr als Metrik verfügbar und wird deshalb durch die Sonarqube Rule „Public types, methods and fields (API) should be documented with Javadoc" ersetzt. Diese Rule wurde auf Kategorie „Critical" gestuft und wird damit in der QS Regel 1 mit überprüft.

Änderungs-Verzeichnis

Versions#	Datum	Grund	Kurzbeschreibung	Kapitel
Version 1	Datum vor 2011-11-18	Ersterstellung durch Thomas Baldauf, Michael Hadrbolec		gesamtes Dokument
Version 2	2012-07-16	Aktualisierung durch Karin Schellner	Restrukturierung und Aktualisierung des gesamten Dokuments. Neudefinition QS Prozess und QS Regeln (Metriken)	gesamtes Dokument
Version 3	2013-12-11	Aktualisierung durch Karin Schellner	Änderungen bei den QS Regeln (Duplications, Javadoc)	QS Regeln
Version 4	2016-06-09	Aktualisierung durch Karin Schellner	Änderungen in den Prozessen und bei den QS Regeln (Comments)	Allgemeines Qualitätskontrolle bei der Implementierung Einzuhaltende QS Prozesse und Regeln
Version 5	2017-12-06	Aktualisierung durch Karin Schellner	Änderungen bezüglich neuer Adresse des Sonarqube Servers und Sonarqube Änderungen bezüglich Javadoc Überprüfungsmöglichkeiten und neuer Sonargraph Version.	gesamtes Dokument