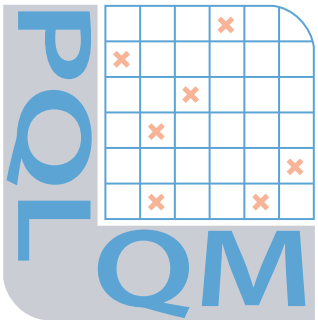
INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



Projekt PQL
Qualitätsmodell

Autor: Florian Deißenböck
Dr. Markus Pizka
Version: 09.03.2006

Technische Universität München - Institut für Informatik - Boltzmannstr. 3 - 85748 Garching
E-Mail: {deissenb, pizka}@in.tum.de - Tel.: +49 89 289-17838/17334

# Fakten	142
# Aktivitäten	27
# Atomare Aktivitäten	20
# Attribute	16
# Attributierte Fakten	181
# Synthetische attributierte Fakten	22
# Beeinflussungen	226
Σ Elemente	592
# Seiten	111
Benutzer	deissenb
Generierung	Thu Mar 09 17:44:21 CET 2006

Inhaltsverzeichnis

1 Einführung

Dieses Dokument beschreibt das Qualitätsmodell *Software-Wartbarkeit*, das im Rahmen des Projekts »Produktivitäts- und Qualitätsaspekte langlebiger Software *(PQL)*« vom Lehrstuhls Software and Systems Engineering (Prof. Dr. Dr. h.c. Manfred Broy) und der Siemens Business Services GmbH & Co. OHG erarbeitet wurde.

Ziel des Qualitätsmodells *PQL-QM* ist es, die Vielzahl der Faktoren, die für die langfristige Produktivität und Qualität in der Software-Wartung entscheidend sind aufzuschlüsseln und ihren Einfluss auf die unterschiedlichen Wartungsaktivitäten darzustellen. Erreicht wird dies durch eine konsequente Trennung zwischen den Wartungsaktivitäten, der Projektsituation und Qualitätsmerkmalen. Durch diese Trennung können die Kostentreiber identifiziert und ihr Einfluss erklärt werden.

Aufbau des Dokuments

Kapitel 2 beschreibt das dem Modell zu Grunde liegende Metamodell und erläutert dessen zentralen Konzepte anhand von Beispielen.

Kapitel 3 gibt einen Überblick über die im Modell abgebildeten Wartungsaktivitäten und zeigt auf, von welchen Systemeigenschaften sie beeinflusst werden.

Kapitel 4 erklärt die relevanten Qualitätseigenschaften und erläutert ihre Relevanz im Kontext des Modells.

Kapitel 5 bildet den »Kern« dieses Dokuments indem es alle relevanten Systemeigenschaften und ihren Einfluss auf die Wartungsaktivitäten im Detail beschreibt. Um den Zugang zum Modell zu erleichtern, bietet das Kapitel einen Überblick über die relevanten Systemeigenschaften und gliedert ihre Detailbeschreibung in organisatorische Aspekte und Aspekte, die das zu wartende Software-Produkt selbst betreffen.

Kapitel 6 fasst den Einfluss der relevanten Systemeigenschaften auf die Wartungsaktivitäten übersichtlich in Form einer Matrix zusammen.

Kapitel 7 und 8 illustrieren die im Modell beschriebenen Aktivitäten und Fakten in Form von Graphen.

Hinweis

Etwa 90% dieses Dokuments wurden vollständig auf Grundlage der Datenbank, in der das Modell verwaltet wird, generiert. Obwohl bei der Implementierung des Dokument-Generators auf höchste Qualität und Lesbarkeit des Dokuments geachtet wurde, kann ein generiertes Dokument nicht den Feinschliff eines manuell erstellten Dokuments erreichen. Dies gilt insbesondere bei Übergängen zwischen Textblöcken, die bei manuell erstellten Dokumenten entsprechend optimiert werden. Des Weiteren ist die Strukturierung des Dokuments an manchen Stellen evtl. nicht intuitiv, da es sich um die Linearisierung eines multidimensionalen Modells handelt. Wann immer sinnvoll wurde zur weitgehenden Behebung dieses Problems eine alphabetische Reihenfolge gewählt.

Da dieses Dokument mehr als Referenz und weniger als Buch, das an einem Stück gelesen wird, angelegt ist, sollten diese Tatsachen die Nutzbarkeit des Dokuments jedoch nicht mindern.

2 PQL-QM-Metamodell

Ziel des Qualitätsmodells *PQL-QM* ist es, die Vielzahl der Faktoren, die für die langfristige Produktivität und Qualität in der Software-Wartung entscheidend sind aufzuschlüsseln und ihren Einfluss auf die unterschiedlichen Wartungsaktivitäten darzustellen. Erreicht wird dies durch eine konsequente Trennung zwischen den Wartungsaktivitäten, der Projektsituation und Qualitätsmerkmalen. Durch diese Trennung können die Kostentreiber identifiziert und ihr Einfluss erklärt werden.

Die folgenden Abschnitte beschreiben das *PQL-QM* im Detail, einen detaillierten Einblick in die Entwicklung des Modells und eine Übersicht über verwandte Arbeiten gibt [1].

2.1 Aktivitäten

Die Menge der Aktivitäten enthält alle Aktivitäten, die im Rahmen der Wartung in einem bestimmten Projekt ausgeführt werden. Sie ist daher durch den konkreten Prozess definiert, den eine Organisation für ein Projekt einsetzt. Typische Aktivitäten sind *Implementierung*, *Test* oder *Build*.

Für eine realistische Bewertung des Einflusses unterschiedlicher Faktoren auf die Wartung ist es notwendig, allgemeine Aktivitäten wie *Analyse* in spezifischere Aktivitäten wie *Programm-Verstehen* und *Konzept-Lokalisierung* aufzuteilen.

Durch die sukzessive Aufteilung von unspezifischen Aktivitäten in spezifischere *Unteraktivitäten* ergibt sich ein *Aktivitäten-Baum*, dessen Blätter als *atomare Aktivitäten* bezeichnet werden.

Der Aktivitäten-Baum des *PQL-QM* ist in Kapitel 7 graphisch dargestellt; relevante Aktivitäten sind in Kapitel 3 im Detail beschrieben.

2.2 Situation

Da das *PQL-QM* im Gegensatz zu herkömmliche Qualitätsmodellen nicht nur Eigenschaften des zu wartenden Produkts, sondern auch auch Eigenschaft der Organisation, die die Wartung durchführt, beschreibt, wird hier bewusst nicht auf *Systemeigenschaften* eingegrenzt, sondern von allgemein *Fakten über die Projektsituation* gesprochen.

Analog zu den Aktivitäten wird bei der Situation vorgegangen. Die Beschreibung einer komplexen Situation wird ebenfalls durch die sukzessive Verfeinerung der, sie beschreibenden *Fakten*, erreicht.

Typische unspezifische Fakten sind z. B. *Organisation*, *Produkt*, die sich über Zwischenebenen wie *Infrastruktur* oder *Dokumentation* zu *atomaren Fakten* wie *Debugger* oder *Black-Box-Dokumentation* verfeinern lassen.

Jedoch werden bei der Verfeinerung der Fakten im Situations-Baum zwei unterschiedliche Beziehungstypen verwendet:

- *Spezialisierung.* Eine *Spezialisierungsbeziehung* zwischen zwei Fakten $f_1 \rhd f_2$ bedeutet, dass Fakt f_1 eine Spezialisierung des Fakts f_2 ist bzw. dass Fakt f_2 eine Generalisierung des Fakts f_1 ist.

  So drückt z. B. die Beziehung *FOR-Schleife* $\rhd$ *Schleife* aus, dass eine *FOR-Schleife* eine Spezialisierung einer *Schleife* ist.

- *Aggregation.* Eine *Aggregationsbeziehung* zwischen zwei Fakten $f_3 \blacktriangleright f_4$ bedeutet, dass Fakt f_3 Teil des Fakts f_4 ist bzw. dass Fakt f_4 aus dem Fakt f_3 (und möglichen anderen Fakten) besteht.

  So drücken z. B. die Beziehungen *Kopf* ► *Schleife* und *Rumpf* ► *Schleife* aus, dass eine *Schleife* aus einem *Kopf* und einem *Rumpf* besteht.

  Zu beachten ist, dass die *Aggregationsbeziehung* bei abstrakteren Fakten oft den Charakter einer *Sicht-auf*-Beziehung einnimmt. So drücken z. B. die Beziehungen *Inhalt* ► *Dokumentation* und *Darstellung* ► *Dokumentation* aus, dass die Dokumentation inhaltliche und darstellerische Aspekte hat. Im Sinne des Modells handelt es sich jedoch auch hierbei um eine Aggregationsbeziehung.

Die beiden Beziehungstypen werden im Situations-Baum frei kombiniert. Ein Beispiel zeigt Abbildung 2.1.

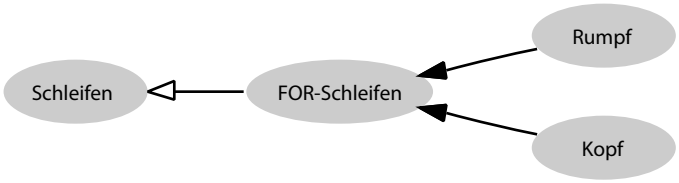


Abbildung 2.1: Beispiel-Fakten

Da es sich bei beiden Beziehungstypen um $1 : N$ (bzw. $N : 1$) -Relationen handelt, ist für den Situations-Baum sichergestellt, dass seine Baumeigenschaft (Zyklenfreiheit) erhalten bleibt.

2.3 Attribute

An den oben genannten Beispielen lässt sich leicht erkennen, dass der Zusammenhang zwischen der Situation und den Aktivitäten durch *Qualitätsmerkmale* oder *Attribute* definiert sein muss. So ist z. B. für die Analyse einer Komponente nicht nur entscheidend, ob eine Black- Box-Dokumentation vorhanden ist, sondern u. a. auch, dass diese vollständig, konsistent und kongruent zur Implementierung ist. Eine baumartige Zerlegung der Qualitätsmerkmale ist aufgrund ihrer Allgemeinheit nicht möglich. So ist z. B. das Attribute *Vollständigkeit* nicht ohne Angabe eines konkreten Kontextes verfeinerbar.

2.4 Attributierte Fakten

Da ein Fakt nur in Zusammenhang mit einem Attribut bewertbar ist, definiert das *PQL-QM attributierte Fakten* als zentrale Entität des Modells. Ein attributierter Fakt ist ein Tupel $(f, q) \in F \times Q$ wobei F die Menge der Fakten und Q die Menge der Qualitätsattribute ist. So kann z. B. der attributierte Fakt (*Klasse*, *Überflüssigkeit*) definiert werden, der nicht verwendete Klassen (d. h. *toten Code*) adressiert.

Einem attributierten Fakt ist im *PQL-QM* immer eine Bewertungsvorschrift zugeordnet, die beschreibt wie der attributierte Fakt zu bestimmen ist.

Wird ein attributierter Fakt (f, q) definiert, wobei f Wurzel einer Spezialisierungsbeziehung ist, so wird das Attribut q an die Spezialisierungen von f *vererbt*. So kann z. B. das Attribut *Überflüssigkeit* direkt auf dem Fakt *Komponente* definiert werden, der abstrakt alle Komponenten eines Softwaresystems beschreibt. Ist der Fakt *Klasse* eine Spezialisierung des Fakts *Komponente*, so ist automatisch auch der attributierte Fakt (*Klasse*, *Überflüssigkeit*) definiert. Hierbei besteht die Möglichkeit, eine attributierten Fakt als *abstrakt* zu definieren. Einem abstrakten attributierten Fakt muss keine Bewertungsvorschrift zugeordnet sein, jedoch muss jede seiner Spezialisierung eine solche definieren.

Dadurch ergibt sich, wie im obigen Beispiel gezeigt, die Möglichkeit grundsätzliche Aussagen wie »Komponenten dürfen nicht überflüssig sein« zu formulieren ohne dass die Bewertung dieser Eigenschaft präzisiert werden muss. Erst für Spezialisierungen des Fakts *Komponente* muss die Bewertungsvorschrift konkret angegeben werden. Ein abstrakter attributierter Fakt ist als $\overline{(f, q)}$ dargestellt; nicht-abstrakte attributierte Fakten werden als *konkret* bezeichnet.

Zu beachten ist, dass die Attribut-Vererbung nur für Spezialisierungsbeziehungen, nicht für Aggregationsbeziehung definiert ist.

2.5 Einflüsse

Um den Einfluss der Projekt-Situation auf die Wartungsaktivitäten zu modellieren definiert das *PQL-QM* Einflussbeziehungen zwischen attributierten Fakten und Aktivitäten.

Somit ist die Einflussbeziehung eine Funktion

$$i : (F \times Q) \times A \to I,$$

wobei A die Menge aller Aktivitäten ist und I eine geeignete Menge zur Beschreibung des Einflusses ist. Je nach notwendiger und möglicher Granularität könnte dies beispielsweise die Menge $\mathbb{B} = \{\text{true}, \text{false}\}$ oder die Menge der reellen Zahlen $\mathbb{R}$ sein. Im Idealfall würde eine monetäre Skala verwendet, die den Einfluss der attributierten Fakten z. B. in Euro angäbe. In der vorliegenden Version den Modells wird die dreiwertige Menge $\{+, \circ, -\}$ verwendet, die es erlaubt einen positiven, neutralen oder negativen Einfluss auszudrücken:

$$i_c : (F \times Q) \times A \to \{+, \circ, -\}.$$

Wird eine Einflussbeziehung für eine nicht-atomare Aktivität a definiert, so überträgt sich die Beziehung auf alle Teilaktivitäten von a.

Erbt ein Fakt einen attributierten Fakt von einer Generalisierung, so wird auch die entsprechende Einflussbeziehung vererbt.

2.6 Integritätsbedingungen

Das Modell unterliegt folgenden Integritätsbedingungen:

- Ein Fakt kann nicht gleichzeitig Wurzel einer Spezialisierungs- und Aggregations-Beziehung sein.
- Mit jeden atomaren Fakt muss mindestens ein Attribut assoziert sein.

- Ist ein attributierter Fakt abstrakt definiert, so muss zumindest für jedes Blatt seiner Spezialisierungshierarchie dieser attributierte Fakt konkretisiert sein.
- Für jeden attributierten Fakt muss mindestens eine Einflussbeziehung definiert sein.
- Für jeden konkreten attributierten Fakt muss eine Bewertungsvorschrift angegeben sein.
- Für jede atomare Aktivität muss mindestens eine Einflussbeziehung definiert sein.

2.7 Zusammenfassung

Abbildung 2.2 veranschaulicht das Modell. Dargestellt ist eine stark vereinfachtes Modell, das über nur sieben Fakten, sieben Aktivitäten und 5 Qualitätsmerkmale verfügt. Da die Menge der Qualitätsmerkmale im Allgemeinen kleiner als die Mengen der Aktivitäten bzw. der Fakten ist, wird aus Gründen der Übersichtlichkeit auf eine vollständig dreidimensionale Darstellung verzichtet. Die Qualitätseigenschaften sind daher optisch in die Dimension der Situation mit aufgenommen. Außerdem sind neutrale Einflüsse (○) dadurch gekennzeichnet, dass entsprechende Matrix-Elemente keine Markierung tragen.

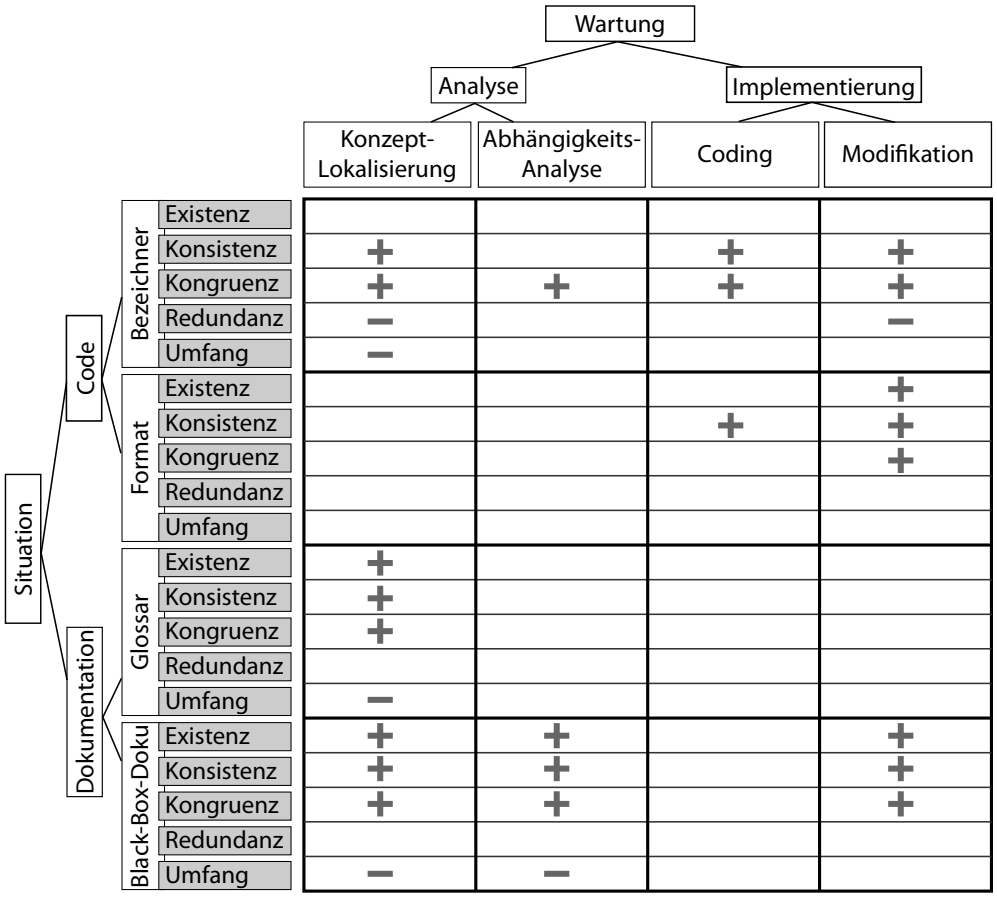


Abbildung 2.2: Beispiel-Qualitätsmodell

2.8 Beispiele

Folgende Beispiele dienen der Veranschaulichung des Modells. Die Beziehung

$$i_c((\textit{Bezeichner}, \textit{Konsistenz}, \textit{Konzept-Lokalisierung})) = +$$

drückt aus, dass konsistente Bezeichner eines Programms einen positiven Einfluss auf die Aktivität Konzept-Lokalisierung haben. Dies ist durch die typischerweise verwendeten Konzept-Lokalisierungs-Strategien begründet, die auf der Analyse der Bezeichnern beruhen.

Entsprechend drückt die Beziehung

$$i_c((\textit{Black-Box-Doku}, \textit{Umfang}, \textit{Abhängigkeits-Analyse})) = -$$

aus, dass eine der Umfang der Black-Box-Dokumentation einen negative Einfluss auf die Abhängigkeits-Analyse hat, da der Leser im Durchschnitt mehr Dokumentation lesen muss, bis das gesuchte gefunden ist, wenn der Umfang größer ist. Zu beachten ist, dass an dieser Stelle noch keine Aussage über den »richtigen« Umfang gemacht wurde. Dies würde z. B. durch ein Qualitätsmerkmal wie *Minimalität* ausgedrückt.

Die Beziehung

$$i_c((\textit{Format}, \textit{Redundanz}, \textit{Abhängigkeits-Analyse})) = \circ$$

drückt einen neutralen Einfluss auf die Abhängigkeits-Analyse aus, da das Merkmal *Redundanz* auf den Fakt *Quelltext-Formatierung* nicht sinnvoll anwendbar ist.

Obwohl das Merkmal *Konsistenz* sich zwar auf den Fakt *Quelltext-Formatierung* anwenden lässt, ist die Beziehung

$$i_c((\textit{Format}, \textit{Konsistenz}, \textit{Abhängigkeits-Analyse})) = \circ$$

trotzdem neutral, das die Abhängigkeits-Analyse meist nicht auf einer Ebene durchgeführt wird, auf der die Quelltext-Formatierung von Belang ist.

3 Aktivitäten

Dieses Kapitel beschreibt die Wartungsaktivitäten des *PQL-QM*. Abbildung 3.1 zeigt die ersten Ebenen des Aktivitätenbaums. In Kapitel 7 findet sich eine vollständige graphische Darstellung der Aktivitäten.

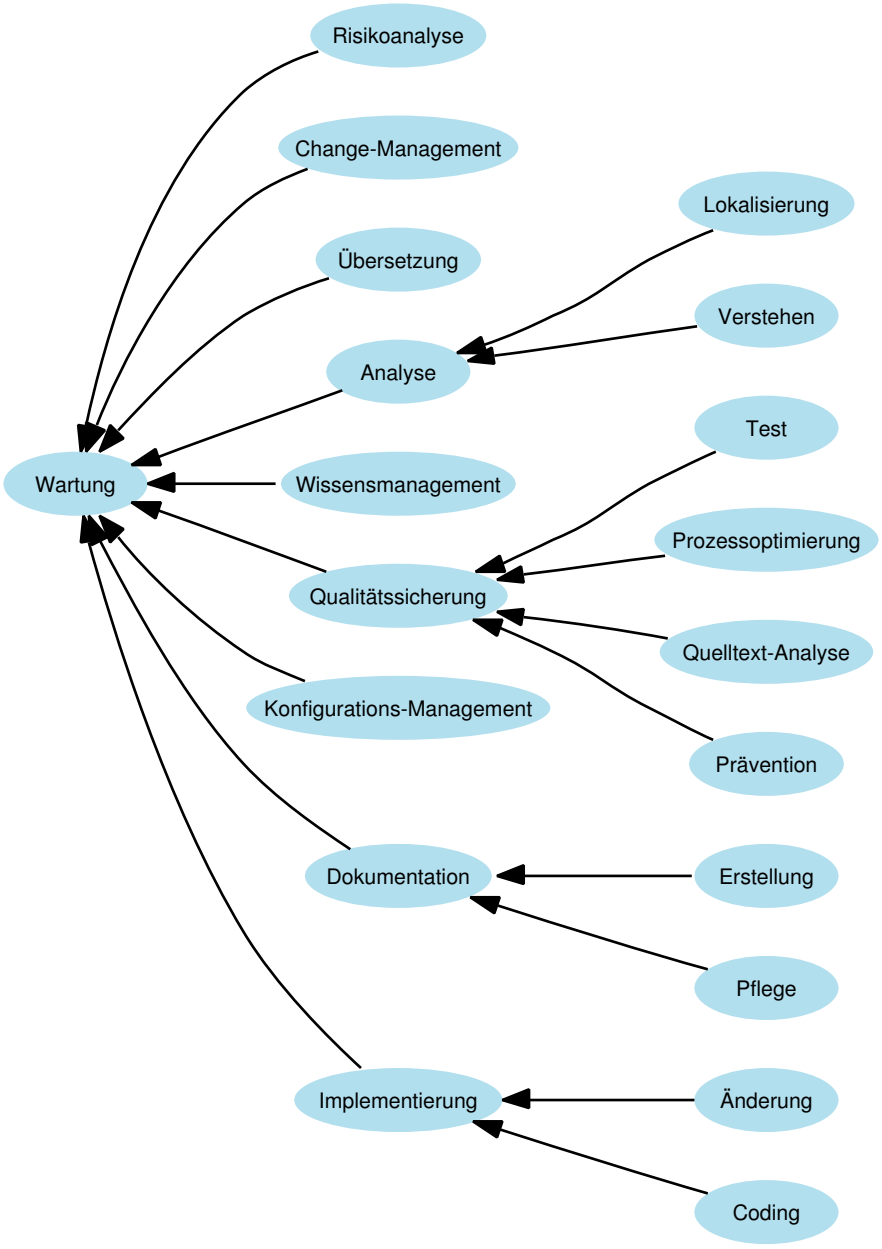


Abbildung 3.1: Aktivitäten-Überblick

Zu erkennen ist, dass sich die Wurzel-Aktivität *Wartung* in 9 Aktivitäten aufteilen lässt, wobei die wichtigstem im Folgenden beschrieben sind:.

- *Implementierung.* Die Implementierung befasst sich mit dem Programmieren, dazu gehören unter anderem das Schreiben von neuem *Code* und die *Änderung* existierenden Quelltextes.
- *Analyse.* Der *Analyse* kommt eine wichtige Rolle zu, da für sie während der Bearbeitung eines *Change-Requests* ca. 50% der Zeit aufgewandt werden [5]. Ihre zentralen Teilaktivitäten sind das *Programm-Verstehen* und die *Lokalisierung* von Information.
- *Dokumentation.* Die Aktivität *Dokumentation* bescheibt die *Erstellung* und *Pflege* der Dokumentation.
- *Qualitätssicherung.* Qualitätssichernde Maßnahmen wie *Testen* oder *Prozessoptimierung* sind durch die Aktivität *Qualitätssicherung* zusammengefasst.

Aktivitäten, die Teil einer Einflussbeziehung sind, werden in den folgenden Abschnitten beschrieben. Zusätzlich ist für jede Aktivität der Pfad zur Wurzel des Aktivitätenbaums und eine List der Fakten, die sie beeinflussen angegeben.

3.1 Abhängigkeitsanalyse

Bei der Änderung eines bestehenden Programms muss im allgemeinen eine *Abhängigkeits-Analyse* durchgeführt werden, um herauszufinden, welche Programmteile von der geplanten Änderung betroffen sind.

Kontext: Abhängigkeitsanalyse → Lokalisierung → Analyse → Wartung

Beeinflusst von: Klassen, Komponenten-Browser, Methoden, Nachbereich, Soziale Kompetenz, Spezialisten, Struktur, Struktur-Navigator, Technische Kompetenz, Variablen, Vermischung, Visualisierungen, Vorbereich, Wartungs-Budget

3.2 Analyse

Alle Analyseaktivitäten, die im Rahmen eines *Change-Request* durchgeführt werden.

Kontext: Analyse → Wartung

Beeinflusst von: Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.3 Änderung

Die Aktivität *Änderung* bezieht sich auf Änderungen an existierendemCode und grenzt sich daher von der Erstellung neuen Quelltextes ab.

Kontext: Änderung → Implementierung → Wartung

Beeinflusst von: Algorithmen, Ausdrücke, Bitoperationen, Boolsche Ausdrücke, Case-Zweig, Datenstrukturen, FOR-Schleifen, Fehlerbehandlung, Fliesskomma-Variablen, GOTO-Anweisungen, Glossar, Include-Direktive, Klammerung, Klassen, Kopf, Laufzeit, Laufzeit-Überprüfung, Literale, Methoden, Refactoring, Reflektion, Schnittstelle, Shift-Operator, Sicherheit, Size-of-Operator, Soziale Kompetenz, Spezialisten, Switch-Anweisung, System, Technische Kompetenz, Type-Casts, Variablen, Vergleiche, Verschränkte Rekursion, Vorzeichenlose Ausdrücke, Wartungs-Budget, Wartungs-Dokumentation, Zeiger-Arithmetik

3.4 Change-Management

Change-Management-Aktivitäten dienen der Organisation der Bearbeitung eines *Change-Requests*. Im Rahmen des PQL-QM beschreibt diese Aktivität den tatsächlichen Management-Anteil dieser Aktivität.

Kontext: Change-Management → Wartung

Beeinflusst von: Change-Management, Change-Management, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.5 Code Lesen

Das tatsächliche Lesen des Quelltext wird im PQL-QM klar von Aktivitätenwie *Änderungen am Quelltext* getrennt, um die Einflüsse derunterschiedlichen Faktoren klarer herauszuarbeiten. Die Beziehung zwischendem Lesen des Quelltextes und anderen Aktivitäten wird durch die Strukturdes Aktivitäten-Baums ausgedrückt.

Kontext: Code Lesen → Verstehen → Analyse → Wartung

Beeinflusst von: Algorithmen, Anweisungen, Ausdrücke, Boolsche Ausdrücke, Case-Zweig, Datenstrukturen, Einrückung, Format, GOTO-Anweisungen, Implementierung, Implementierung, Include-Direktive, Inhalt, Klammerung, Klassen, Komma-Operator, Komponenten, Kopf, Leerzeichen, Leerzeilen, Literale, Methoden, Nebenläufigkeit, Operatoren, Reflektion, Rekursion, Rumpf, Schnittstelle, Schnittstelle, Soziale Kompetenz, Spezialisten, Struktur, Strukturierung, Syntaxhervorhebung, System, Technische Kompetenz, Ternäre Ausdrücke, Type-Casts, Variablen, Variablen-Deklaration, Verbreitung, Vergleiche, Verhaltensbeschreibung, Vermischung, Verschränkte Rekursion, Verteilung, Vorbereich, Wartungs-Budget, Zeiger-Arithmetik

3.6 Coding

Unter *Coding* wird im PQL-QM das Schreiben von *neuem* Quelltext,also nicht die Änderung von Bestehenden, verstanden.

Kontext: Coding → Implementierung → Wartung

Beeinflusst von: Beautifier, Code-Assistenz, Code-Vervollständigung, Fehlerbehandlung, Inkrementeller Compiler, Namenskonvention, Quelltext-Konvention, Schnittstelle, Soziale Kompetenz, Spezialisten, Syntaxhervorhebung, Technische Kompetenz, Verbreitung, Wartungs-Budget

3.7 Debugging

Beim *Debugging* handelt sich ebenfalls um eine Lokalisierungs-Aktivität,die der Auffindung von Fehlern dient.

Kontext: Debugging → Lokalisierung → Analyse → Wartung

Beeinflusst von: Debugger, Nebenläufigkeit, Rekursion, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Verbreitung, Vermischung, Verschränkte Rekursion, Verteilung, Wartungs-Budget

3.8 Dokumentation

Die Aktivität *Dokumentation* beschreibt alle Aktivitäten, die mit der Dokumentations-Erstellung und Pflege aber nicht ihrer Benutzung zu tun haben.

Kontext: Dokumentation → Wartung

Beeinflusst von: Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.9 Dokumentation Lesen

Entsprechend der Aktivität *Code Lesen* bezieht sich die Aktivität*Dokumentation Lesen* ausschließlich auf das tatsächliche Lesen derDokumentation.

Kontext: Dokumentation Lesen → Verstehen → Analyse → Wartung

Beeinflusst von: Format, Funktionalität, Glossar, Index, Nebenläufigkeit, Querverweise, Soziale Kompetenz, Spezialisten, Sprache, Struktur, Technische Kompetenz, Volltextsuche, Wartungs-Budget

3.10 Erstellung

Diese Aktivität beschreibt die Erstellung der Dokumentation.

Kontext: Erstellung → Dokumentation → Wartung

Beeinflusst von: Dokumentations-Konvention, Generatoren, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.11 Implementierung

Die Implementierungsaktivität. Vereinfachend wird hierbei angenommen, dass alle Analyse-Aktivitäten vor der Implementierung durchgeführt wurden.

Kontext: Implementierung → Wartung

Beeinflusst von: Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.12 Konfigurations-Management

Das Konfigurations-Management dient der Verwaltung unterschiedlicher Versionen und Konfigurationen des Produkts.

Kontext: Konfigurations-Management → Wartung

Beeinflusst von: Konfigurations-Management, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.13 Konzept-Lokalisierung

Konzept-Lokalisierung ist eine der zentralen Aktivitäten bei der Bearbeitungeines *Change-Requests*. So erfordert z.B. der Change-Request *Erweitern Sie das Programm um die Darstellung des Datums x* u.A. dieLokalisierung der Konzepte *Darstellung* und *x*.

Kontext: Konzept-Lokalisierung → Lokalisierung → Analyse → Wartung

Beeinflusst von: Inhalt, Komponenten, Soziale Kompetenz, Spezialisten, System, Technische Kompetenz, Wartungs-Budget

3.14 Pflege

Diese Aktivität beschreibt die Pflege der Dokumentation.

Kontext: Pflege → Dokumentation → Wartung

Beeinflusst von: Generatoren, Komponenten-Dokumentation, Plege-Werkzeuge, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Variablen-Deklaration, Wartungs-Budget

3.15 Prävention

Präventive Maßnahmen dienen der Qualitätsverbesserung sind aber nicht durch Anforderung seitens des Kundens motiviert. Solche Maßnahmen können einen Wettbewerbsvorteil bedingen, da zukünftige Änderungsanträge effizienter bearbeitet werden können.

Kontext: Prävention → Qualitätssicherung → Wartung

Beeinflusst von: Konsolidierung, Qualitätssicherung, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.16 Profiling

Das *Profiling* ist eine Lokalisierung-Aktivität, die dem Auffindenvon Programmstellen mit ungenügenden Laufzeit-Eigenschaften dient.

Kontext: Profiling → Lokalisierung → Analyse → Wartung

Beeinflusst von: Laufzeit, Profiler, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Verbreitung, Verteilung, Wartungs-Budget

3.17 Prozessoptimierung

Die Prozessoptimierung dient der kontinuierlichen Verbesserung der von der Organisation verwendeten Prozesse.

Kontext: Prozessoptimierung → Qualitätssicherung → Wartung

Beeinflusst von: Change-Management, Prozessmodell, Qualitätssicherung, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.18 Qualitätssicherung

Sämtliche Qualitätssicherungsaktivitäten sind unter dieser Aktivität zusammengefasst.

Kontext: Qualitätssicherung → Wartung

Beeinflusst von: Qualitätssicherung, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.19 Quelltext-Analyse

Unter *Quelltext-Analyse* versteht das PQL-QM Analysetätigkeiten die im Rahmend er Qualitätssicherung am Quelltext durchgeführt werden. Dazu gehören sowohl automatische Analysen als auch Reviews.

Kontext: Quelltext-Analyse → Qualitätssicherung → Wartung

Beeinflusst von: If-Zero-Direktive, Metrik-Werkzeuge, Qualitätssicherung, Quelltext-Konvention, Reflektion, Richtlinien-Checker, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Verbreitung, Wartungs-Budget

3.20 Risikoanalyse

Die *Risikoanalye* dient der Aufdeckung von bisher unerkannten Risikien und der Neubewertung bekannter Risiken.

Kontext: Risikoanalyse → Wartung

Beeinflusst von: Annahmen, Risiko-Analyse, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget

3.21 Test

Im Rahmen der Aktivität *Test* werden automatisierteoder manuelle Test durchlaufen um Fehler im Produkt zu entdecken. DieTest-Durchführung ist von einer Vielzahl von Faktoren betroffen. Beispielesind der Test-Umfang, die Test-Dauer und der Automatisierungsgrad.

Kontext: Test → Qualitätssicherung → Wartung

Beeinflusst von: Ausführung, Ergebnisse, Erklärung, Priorisierung, Qualitätssicherung, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Testfälle, Vorraussetzungen, Wartungs-Budget

3.22 Übersetzung

Die *Übersetzung* ist eine Aktivität die typischerweise von einem*Compiler* durchgeführt wird. Nichtsdestotrotz ist ihre Beobachtungim Rahmen des Modells von Nöten. So können z.B. in C++ ungeschickt implementierte Inklusionsstrukturen die für die Übersetzung benötigten Zeitenerhöhen und damit die Produktivität insgesamt negativ beeinflussen.

Kontext: Übersetzung → Wartung

Beeinflusst von: Include-Direktive, Komponenten, System, Wartungs-Budget

3.23 Wartung

Wartung ist die allen Teilaktivitäten übergeordnete Aktivität und beschreibt somit den ganzen Wartungsprozess. Ihr zugeordnet sind Fakten, deren Einfluss globaler Natur ist.

Kontext: Wartung

Beeinflusst von: Wartungs-Budget

3.24 Wissensmanagement

Das *Wissensmanagement* dient der Erfassung, Organisation und Bereitstellung relevanten Wissens innerhalb einer Organisation.

Kontext: Wissensmanagement → Wartung

Beeinflusst von: Fluktuation, Soziale Kompetenz, Spezialisten, Technische Kompetenz, Wartungs-Budget, Wissensmanagement

3.25 Zurückverfolgung

Unter *Zurückverfolgung* werden Aktivitäten verstanden, deren Ziel es ist, den Zusammenhang zwischen den Anforderungen, Architekturentwurf und der aktuellen Implementierung herzustellen.

Kontext: Zurückverfolgung → Verstehen → Analyse → Wartung

Beeinflusst von: Anforderungen, Bezug, Entwurfs-Entscheidungen, Soziale Kompetenz, Spezialisten, Struktur, Technische Kompetenz, Wartungs-Budget

4 Qualitätsmerkmale

Bei der Betrachtung der Qualität eines Systems lassen sich bestimmte allgemeine Qualitätsmerkmale identifizieren, die wiederholt und in unterschiedlichen Zusammenhängen eine Rolle spielen. Diese werden in den folgenden Abschnitten beschrieben.

4.1 Angemessenheit

Das Merkmal *Angemessenheit* beschreibt, ob ein Fakt seiner Situation *angemessene* Eigenschaften aufweist. So bedeutet z.B. für einen Algorithmus *Angemessenheit*, dass er für das zu lösende Problem tatsächlich geeignet ist. Andrerseits wäre z.B. die Verwendung einer *IF-O-Direktive* für Dokumentationszwecken (in C++) *unangemessen*.

Attribut von: Algorithmen, Datenstrukturen, If-Zero-Direktive, Include-Direktive, Operatoren, Shift-Operator, Sprache

4.2 Automatisierungsgrad

Je höher der *Automatisierungsgrad* ist, desto niegriger ist der Aufwand für manuelle Leistungen.

Attribut von: Refactoring, Testfälle

4.3 Eindeutigkeit

Die Fakten einer Situation haben eine klare, eindeutige Bedeutung. Es werden keineElemente verwendet, die abhängig vom Kontext unterschiedliche Bedeutungen haben,aber ansonsten nicht zu unterscheiden sind.

Attribut von: Ausdrücke, Bitoperationen, FOR-Schleifen, Fliesskomma-Variablen, Include-Direktive, Inhalt, Variablen, Vergleiche, Vorzeichenlose Ausdrücke

4.4 Existenz

Ein Qualitätsmerkmal vieler Fakten ist deren bloße *Existenz*. Dies kann sich sowohl positiv wie negative auswirken. So ist die Existenz eines Konzept-Glossars positiv zu bewerten, die Existenz von Architektur-Verletzungen jedoch negativ.

Attribut von: Annahmen, Ausführung, Beautifier, Bezug, Change-Management, Change-Management, Code-Assistenz, Code-Vervollständigung, Debugger, Dokumentations-Konvention, Einrückung, Entwurfs-Entscheidungen, Ergebnisse, Erklärung, GOTO-Anweisungen, Generatoren, Index, Inkrementeller Compiler, Komma-Operator, Komponenten-Browser, Konfigurations-Management, Konsolidierung, Laufzeit, Laufzeit-Überprüfung, Metrik-Werkzeuge, Namenskonvention, Nebenläufigkeit,

Nebenläufigkeit, Plege-Werkzeuge, Priorisierung, Profiler, Prozessmodell, Qualitätssicherung, Quelltext-Konvention, Querverweise, Reflektion, Rekursion, Richtlinien-Checker, Risiko-Analyse, Sicherheit, Spezialisten, Struktur-Navigator, Syntaxhervorhebung, Ternäre Ausdrücke, Verhaltensbeschreibung, Vermischung, Verschränkte Rekursion, Verteilung, Visualisierungen, Volltextsuche, Vorraussetzungen, Wartungs-Dokumentation, Wissensmanagement, Zeiger-Arithmetik

4.5 Implizität

Unter *impliziter* Information versteht man Information, die mit enthaltend, mit gemeint, aber nicht ausdrücklich gesagt ist. Im Rahmen der Software-Wartung hat *Implizität* meist eine negative Konnotation. *Implizite* Information ist meist schwer zu erkennen und kann im Falle des Übersehens bei Änderungen zu unvorhergesehenen Problemen führen.

Attribut von: Literale, Schnittstelle, Type-Casts, Variablen-Deklaration, Vergleiche, Vorbereich

4.6 Kongruenz

Die *Kongruenz* beschreibt die strukturelle Übereinstimmung zwischen ausgewählten Fakten. Die Kongruenzkann sich dabei auf mehrere Merkmale der einzelnen Fakten beziehen. Ein hohes Maß an Übereinstimmungzwischen zwei Fakten senkt den Aufwand, der notwendig ist um sich in einem der beiden zurecht zu finden,wenn der andere bereits bekannt ist. Beispiel: Kongruenz zwischen Code und Dokumentation. Hier bedeutet Kongruenz, dass die Struktur derDokumentation mit der des Codes (Klassen, Methoden, etc.) übereinstimmt.

Attribut von: Change-Management, Format, Struktur, Struktur

4.7 Konsistenz

Konsistenz bezieht sich auf die Forderung nach der einheitlichen Umsetzung von Konzepten. Dies bedeutet, dass ähnliche oder gleiche Aufgaben analog gelöst werden. Dies kann sich sowohl auf das rein äußerliche Erscheinungsbild,wie auch auf semantische Elemente beziehen. Konsistenz senkt den Aufwand der notwendigist, um bestimmte Fakten zu erfassen. Es können bekannte Muster verwendet und wiedererkannt werden.

Attribut von: Einrückung, Format, Glossar, Leerzeichen, Leerzeilen, Schnittstelle, Strukturierung

4.8 Korrektheit

Die *Korrektheit* eines Faktes ist stark kontextabhängig und meist eher intuitiv definiert. Im Allgemeinen giltein Fakt als korrekt, wenn der die Realität widerspiegelt. Beispielsweise sollte für Dokumentation gefordert werden, dass sie nicht nur kongruent (strukturell übereinstimmend)mit dem dokumentierten System ist, sonder auch korrekt ist und damit das tatsächliche Verhaltendes Systems beschreibt.

Attribut von: Anforderungen, Inhalt, Size-of-Operator

4.9 Lokalität

Lokalität beschreibt wie *lokal* (im Gegensatz zu *global*) ein Artefakt oder eine Information definiert ist.

Attribut von: Klassen, Methoden, Variablen

4.10 Redundanz

Unter *Redundanz* versteht man die Existenz nicht-notwendiger Information. Redundanz hat in den meistenKontexten einen negativen Einfluss auf Wartungs-Aktivitäten, da sie den Umfang unnötig steigert. Zusätzlicherschwert sie Modifikationen, da Änderungen an redundanter Information in den meisten Fällen anmehrerer Stellen wiederholt werden müssen. Dieses Phänomen ist im Bereich der Datenbank unter demBegriff Update-Anomalie bekannt. In manchem Kontext kann sich Redundanz aber auch positiv auswirken. So ist z. B. Redundanz bzgl. spezialisierterMitarbeiter sicherlich wünschenswert.

Attribut von: Glossar, Inhalt, Klassen, Komponenten-Dokumentation, Literale, Methoden, Spezialisten, System

4.11 Regularität

Regularität beschreibt die Eigenschaft eines Faktes, sich an Gesetzmäßigkeiten (Muster) zu halten.

Attribut von: Implementierung, Implementierung, Include-Direktive, Rumpf, Schnittstelle

4.12 Überflüssigkeit

Überflüssigkeit beschreibt den Umstand, dass ein Artefakt oder Information vorhanden ist, obwohl er nicht benötigt wird.

Attribut von: Anweisungen, Ausdrücke, Bezeichner, Include-Direktive, Klassen, Komponenten, Label, Literale, Methoden, Operatoren, Präprozessor-Direktiven, Schleifen, Type-Casts, Variablen

4.13 Umfang

Viele Aktivitäten werden vom *Umfang* eines bestimmten Fakts beeinflusst. Ein einfaches Beispiel ist derUmfang des zu wartenden Systems. Je größer er ist, desto höher sind dieWartungsaufwände.

Attribut von: Algorithmen, Fluktuation, Methoden, Nachbereich, Schnittstelle, Soziale Kompetenz, System, Technische Kompetenz, Testfälle, Verbreitung, Vorbereich, Wartungs-Budget

4.14 Verbindlichkeit

Verbindlichkeit beschreibt, ob eine Regel oder Vorgabe *verbindlich*, d.h. unumgänglich definiert ist. Hierbei ist im allgemein eine technisch erzwungene und keine organisatorisch oder sozial definierte Verbindlichkeit gemeint. Ein Beispiel ist der Ausnahmebehandlungsmechanismus von Java, der dem Entwickler in bestimmten Situation einen Fehlerbehandlung zwingend vorschreibt.

Attribut von: Change-Management, Fehlerbehandlung

4.15 Vollständigkeit

Vollständigkeit ist ebenfalls eine Eigenschaft deren Definition starkvom jeweiligen Kontext abhängig ist. Pauschal lässt sich Vollständigkeitmit der Berücksichtigung aller benötigten Entitäten umschreiben.

Attribut von: Anforderungen, Annahmen, Boolsche Ausdrücke, Case-Zweig, Change-Management, Entwurfs-Entscheidungen, Funktionalität, Glossar, Index, Klammerung, Querverweise, Struktur, Struktur-Navigator, Switch-Anweisung, Visualisierungen, Volltextsuche

4.16 Zugänglichkeit

Zugänglichkeit wird im Modell mit der Bedeutung von *einfach zu benutzen*, *nicht unnötig kompliziert* verwendet.

Attribut von: Algorithmen, Boolsche Ausdrücke, Format, Implementierung, Kopf, Methoden, Querverweise, Variablen

5 Situation

Dieses Kapitel bildet den »Kern« dieses Dokuments indem es alle als relevant identifizierten Systemeigenschaften und ihren Einfluss auf die Wartungsaktivitäten im Detail beschreibt. Abschnitt 5.1 gibt einen kurzen Überblick über den Aufbau des Situationsbaums und Abschnitt 5.2 erläutert das für die Dokumentation der einzelnen Fakten verwendete Format. Die folgenden Abschnitt beinhalten die Detail-Beschreibung der einzelnen attributierten Fakten. Eine graphischen Überblick über alle Fakten vermittelt Kapitel 8.

5.1 Überblick

Um den Einstieg in das *PQL-QM* zu erleichtern, geben die folgenden Abschnitte einen Überblick über dessen Situationsbaum und erläutern zentrale Fakten. Zur Erleichterung des Einstiegs zeigt Abbildung 5.1 die ersten Ebenen des Situationsbaums.

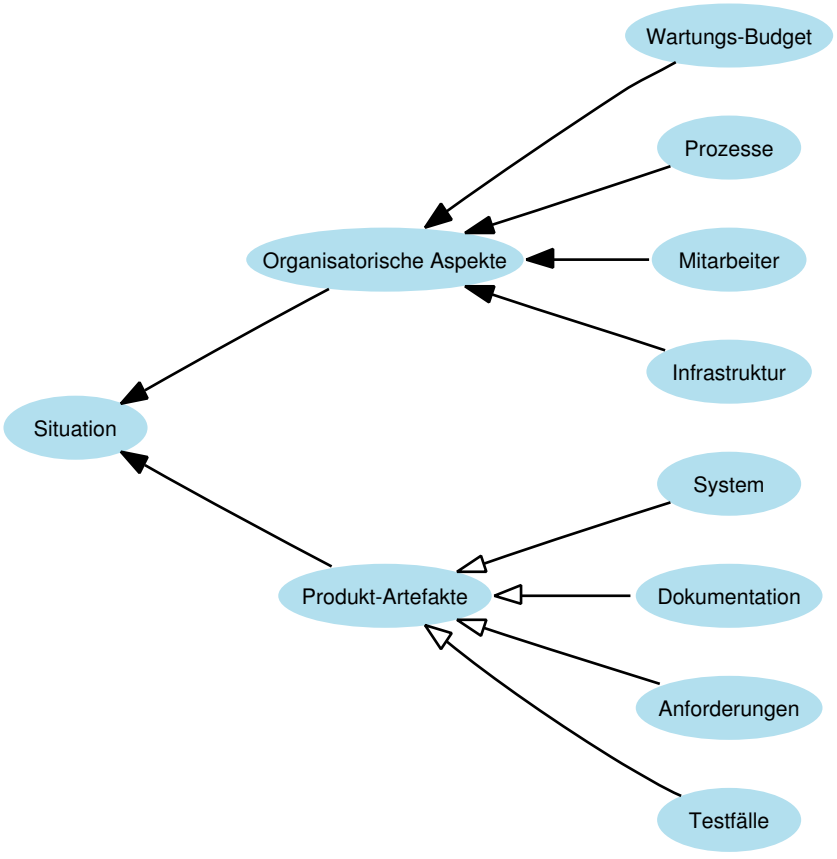


Abbildung 5.1: Situations-Überblick

Grundsätzlich unterscheidet das Modell zwischen Fakten, die sich auf das zu wartenden *Produkt* beziehen und Fakten, welche die *Organisation* beschreiben, die die Wartung durchführt.

5.1.1 Produkt

Die Beschreibung des Produkts wird entsprechend in Fakten bzgl. der *Dokumentation* und Fakten bzgl. der *Software-Systems* selbst unterteilt.

Dokumentation

Ein für die Wartung entscheidender Punkt ist die *Dokumentation* des Systems. Zur detaillierteren Beurteilung wird hierbei zwischen dem *Inhalt* und der *Aufbereitung* der Dokumentation unterschieden.

- *Inhalt.* Der *Inhalt* der Dokumentation setzt sich aus den unterschiedlichen Dokumentations-Typen zusammen.
  - *Komponenten-Beschreibung.* Die *Komponenten-Beschreibung* bietet für jede Komponente des Systems eine eigene Beschreibung an. Typischerweise unterteilt sich ein System in Komponenten-Typen wie *Subsysteme*, *Klassen* oder *Methoden*. Bei allen Komponenten-Typen sollten die Innen- und die Außensicht getrennt dokumentiert sein.
    - *Black-Box-Dokumentation.* Unter der *Black-Box-Dokumentation* einer Komponente wird die Beschreibung der Schnittstelle (Außensicht) verstanden. Diese Art der Dokumentation ist wichtig um die kognitive Belastung des Lesers möglichst gering zu halten. So kann der Leser nach der Konsultation der Außensicht einer Komponente entscheiden, ob für ihn auch die Innensicht von Relevanz ist.
    - *Glass-Box-Dokumentation.* Die *Glass-Box-Dokumentation* einer Komponente beschreibt ihre Innensicht. Dadurch wird dokumentiert welche Sub-Komponenten die Komponente enthält und wie diese in Verbindung stehen. Wichtig hierbei ist, dass die Glass-Box-Sicht im Gegensatz zur Black-Box-Sicht zwar die Innensicht in eine Komponente bietet, nicht aber die Innensicht in die in ihr enthaltene Komponenten. Von diesen steht wiederum nur eine Black-Box-Sicht zur Verfügung.
  - *Konzept-Glossar.* Jedes Software-System besteht aus bzw. beschreibt eine Reihe von *Konzepten* [7]. Konzepte können sowohl fachlicher Natur (z. B. Kontonummer) als auch technischer Natur (z. B. Speicherverwaltung) sein. Das *Konzept-Glossar* enthält eine Liste aller verwendeten Konzepte mit entsprechenden Erklärungen und hilft damit, ein gemeinsames Verständnis der Konzepte zu etablieren.
  - *Identifier Dictionary.* Auf Konzepte wird im Programmtext meist in Form von *Bezeichnern* verwiesen. Das *Identifier Dictionary* stellt sicher, dass eine wohldefinierte Abbildung von Bezeichnern auf Konzepte existiert. Siehe dazu auch [3].
- *Aufbereitung.* Entscheidend für den tatsächlichen Nutzen der Dokumentation ist nicht nur ihr Inhalt, sondern auch ihre Aufmachung. Die »Benutzerfreundlichkeit« der Dokumentation ist entscheidend für ihre Akzeptanz und ihren Nutzen.
  - *Format.* Ein entscheidendes Kriterium für »Benutzerfreundlichkeit« der Dokumentation ist das verwendete Format. Die zentralen Punkte hierbei sind die Konsistenz des Formats und die grundsätzliche Entscheidung zwischen einer sequentiellen, Buch-ähnlichen Dokumentation bzw. einer ungeordneten, mit Hyperlinks verbundenen, Struktur.

- *Navigation.* Neben dem Format haben die *Navigationsmöglichkeiten* einen großen Einfluss auf die Effizienz bei der Dokumentations-Nutzung. Hier kann eine Hypertext-Format stark unterstützen.

Software-System

Selbstverständlich hat das *Software-System* selbst einen entscheidenden Einfluss auf seine Wartung. Hier wird grundlegend zwischen einer *statischen* und einer *dynamischen* Sicht auf das System unterschieden.

- *Statik. Statische* Aspekte eines Systems sind solche, die sich analysieren lassen ohne das System auszuführen. Sie bilden meistens den Einstiegspunkt zu Verstehens-Prozessen [9, 4, 2].
  - *Komponenten.* Der Begriff *Komponente* wird hier in einer sehr allgemeinen Form verwendet und steht für einen rekursiven Container. Beispiele sind *System* ⊃ *Subsystem* ⊃ *Modul/Paket* ⊃ *Klasse* ⊃ *Methode* ⊃ *Anweisung* ⊃ . . ..

    Für viele dieser Komponente gelten ähnliche Eigenschaft. So ist z. B. generell von Interesse, ob eine Komponenten überflüssig oder redundant ist. Darüberhinaus gibt es eine Reihe von Eigenschaften, die nur für bestimmte Komponenten relevant sind. So betont das Modell z. B. bei Klassen und Methoden (Funktion) die Unterscheidung zwischen einer Black-Box- und einer Glass-Box-Sicht. Bei speziellen Konstrukten, wie z. B. Fallunterscheidungen, sind dagegen andere Eigenschaften, z. B. deren Vollständigkeit, im Zentrum des Interesses.
  - *Sprache.* Unterschiedliche Eigenschaften der verwendeten Programmiersprache haben direkten Einfluss auf die Wartungsaktivitäten.
    - *Fehlerbehandlung.* Ein wichtiger Punkt ist, ob die Sprache eine geeignetes *Fehlerbehandlungs-Konzept* bietet und den Entwickler dazu zwingt, dieses zu verwenden.
    - *Laufzeit-Überprüfungen. Laufzeit-Überprüfungen* wie das Überschreiten von Array-Grenzen können, speziell in Kombination mit einem entsprechenden Fehlerbehandlungs-Konzept, die Modifikation von existierendem Code vereinfachen.
    - *Sprach-Mix.* Manche Sprache (z. B. Jsp) zwingen zu einem Mix aus mehreren Sprachen (z. B. Html und Java). Ein solcher *Sprach-Mix* ist der Wartung im Allgemeinen abträglich.
  - *Programmtext.* Dieser Fakt beschreibt ein rein textuelle Sicht auf das System. Entscheidend ist hier vor allem das *Format* des Quelltextes, das einen deutlichen Einfluss auf die Lesbarkeit [6] hat.
  - *Struktur.* Von zentraler Bedeutung ist die reale *Struktur* des Systems. So ist z. B. von Interesse, ob diese Struktur zu der ursprünglichen Architekturbeschreibung passt.
- *Dynamik. Dynamische* Aspekte des Systems haben einen starken Einfluss auf das Programm-Verstehen.
  - *Nebenläufigkeit. Nebenläufige* System sind um mehrere Grade schwerer zu verstehen und zu analysieren als nicht-nebenläufige Systeme.
  - *Rekursion.* Für nicht speziell geschulte Entwickler erschweren *rekursive* Konstruktionen das Programm-Versehen erheblich, da die kognitiven Modelle, die zum Verständnis benötigt werden, sehr komplex sind. Dies betrifft insbesondere rekursive Konstruktion, die sich über mehrere Komponenten verteilen und kaskadenartige bzw. verschränkte Rekursionen.

5.1.2 Organisation

Neben dem zu wartenden Produkt hat natürlich die Organisation, welche die Wartung durchführt, einen entscheidenden Einfluss auf die Produktivität und Effizienz der Wartungsaktivitäten. Zur detaillierten Beschreibung wird im Folgenden zwischen den *Mitarbeitern* der Organisation, der *Infrastruktur* und *Prozessen* unterschieden.

Mitarbeiter Die Mitarbeiter nehmen im Rahmen der Wartung eine entscheidende Rolle ein. Dabei sind folgende Punkte von zentraler Bedeutung.

- *Ausbildung.* Die Ausbildung der Mitarbeiter hat offensichtlich eine direkten Einfluss auf die Effizienz der Wartung. Hierbei ist sowohl die Ausbildung, die sich auf technische Aspekte bezieht als auch jene, die die soziale und psychologische Kompetenz verbessert, von Belang. Siehe dazu auch [10].
- *Fluktuation. Mitarbeiter-Fluktuation* kann u. U. einen sehr negativen Einfluss auf die Wartungsaktivitäten haben und muss daher entsprechend berücksichtigt werden.
- *Spezialisierung.* Eine *Spezialisierung* der Mitarbeiter, z. B. auf fachliche oder Domänen-spezifische Aspekte, kann im Rahmen der Wartung einen positiven Einfluss auf Wartungsaktivitäten haben.

Infrastruktur Neben den Mitarbeitern hat die *Infrastruktur*, die diesen zur Verfügung steht, einen entscheidenden Einfluss.

- *Werkzeuge.* Viele Wartungsaktivitäten werden von Werkzeugen unterstützt bzw. erst ermöglicht. So ist z. B. die Redundanz-Analyse ohne Werkzeuge äußerst aufwendig und das Übersetzen des Quelltextes ohne einen Compiler praktisch unmöglich.
- *Richtlinien.* Die Existenz von Projekt-weiten Richtlinien ist Voraussetzung für die Erlangung und Einhaltung von Konsistenz bzgl. unterschiedlichster Sachverhalten.
- *Hardware.* Am Beispiel der Übersetzung lässt sich erkennen, dass die Leistungsfähigkeit der Hardware einen entscheidenden Einfluss haben kann. So ist es im Rahmen der Wartung von entscheidendem Einfluss, ob die Übersetzung einer Komponente wenige Minuten oder einige Stunden dauert.

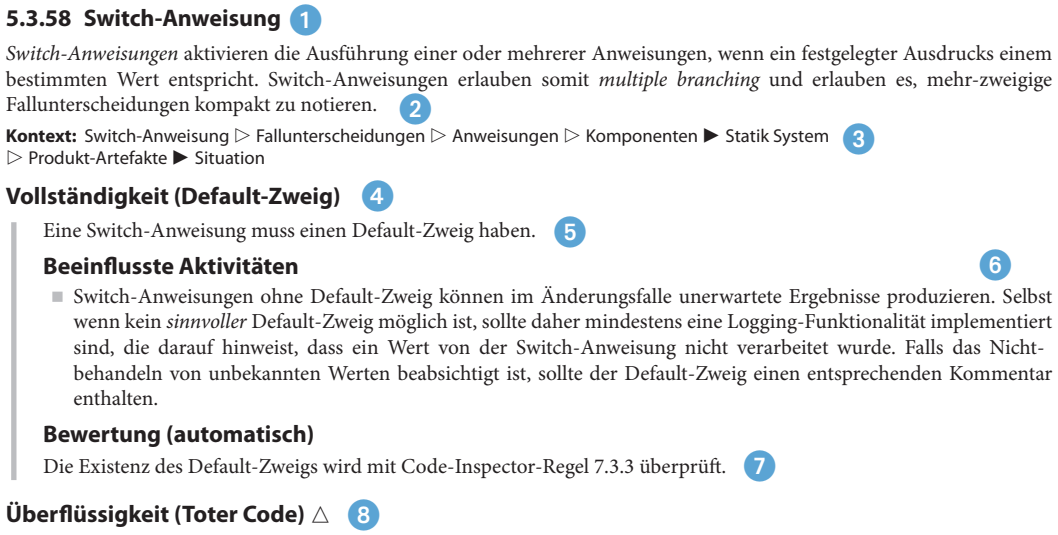
Prozesse Um die Wiederholbarkeit sicherzustellen bzw. die Möglichkeit zur Optimierung zu schaffen ist eine klare Definition der zentralen Prozesse entscheidend. Daher finden sich Fakten bzgl. der Prozessdefinitionen auch im Situationsbaum wieder, obwohl die Durchführung der Prozesses natürlich in From von Aktivitäten im Aktivitäten-Baum modelliert ist.

- *Change-Management.* Der zentrale Prozess der Wartung betrifft das *Change-Management*. Es ist von höchster Wichtigkeit, dass dieser Prozess definiert, eingehalten und optimiert wird. Nur so kann sichergestellt werden, dass die Bearbeitung von *Change-Requests* standardisiert, koordiniert und dokumentiert ist.
- *Konfigurations-Management.* Die Verwaltung des Quelltextes nimmt eine entscheidende Rolle ein. Hierzu müssen Rollen, Vorgehensweisen, Berechtigungen und Werkzeuge definiert sein. Ebenso wichtig ist eine definierte Verzahnung mit dem Change-Management-Prozess.
- *Wissens-Management.* Die Verwaltung und Bereitstellung von impliziten und expliziten Wissen innerhalb eines Projektes oder einer Organisation hat einen entscheidenden Einfluss auf sämtliche Wartungsaktivitäten, da nicht nur Wissen bzgl. der Domäne sondern auch »Meta-Wissen« bzgl. der eigenen Prozesse, Benutzung der Werkzeuge o. Ä. verwaltet wird.

- *Qualitäts-Controlling.* Ein kontinuierliche, zeitnahe Überwachung (und entsprechende Reaktion) definierter Qualitätsparameter ist unerlässlich. Dies ist zum einem notwendig, da die Kosten zur Behebung vieler Qualitätsprobleme mit der Zeit steigen [8]. Zum anderen fördert ein kontinuierliches *Qualitäts-Controlling* die Disziplin der Mitarbeiter und wirkt daher präventiv auf die Schaffung neuer Qualitätsproblem.

5.2 Erläuterung

Der folgende Abschnitte beschreiben alle im *PQL-QM* enthaltenen attributierten Fakten und ihre Einflüsse auf die Wartungsaktivitäten im Detail.



5.3.58 Switch-Anweisung 1

Switch-Anweisungen aktivieren die Ausführung einer oder mehrerer Anweisungen, wenn ein festgelegter Ausdrucks einem bestimmten Wert entspricht. Switch-Anweisungen erlauben somit *multiple branching* und erlauben es, mehr-zweigige Fallunterscheidungen kompakt zu notieren. 2

Kontext: Switch-Anweisung ▷ Fallunterscheidungen ▷ Anweisungen ▷ Komponenten ▶ Statik System 3
▷ Produkt-Artefakte ▶ Situation

Vollständigkeit (Default-Zweig) 4

Eine Switch-Anweisung muss einen Default-Zweig haben. 5

Beeinflusste Aktivitäten 6

- Switch-Anweisungen ohne Default-Zweig können im Änderungsfalle unerwartete Ergebnisse produzieren. Selbst wenn kein *sinnvoller* Default-Zweig möglich ist, sollte daher mindestens eine Logging-Funktionalität implementiert sind, die darauf hinweist, dass ein Wert von der Switch-Anweisung nicht verarbeitet wurde. Falls das Nicht-behandeln von unbekannten Werten beabsichtigt ist, sollte der Default-Zweig einen entsprechenden Kommentar enthalten.

Bewertung (automatisch)

Die Existenz des Default-Zweigs wird mit Code-Inspector-Regel 7.3.3 überprüft. 7

Überflüssigkeit (Toter Code) △ 8

Abbildung 5.2: Fakten-Beschreibung

Hierbei wird das in Abbildung 5.2 beschriebene Format verwendet:

1. Name des betroffenen Fakts
2. Beschreibung des Fakts
3. Einordnung des Fakts im Situationsbaum (▷ beschreibt eine Spezialisierungsrelation, ▶ eine Aggregationsrelation)
4. zugeordnetes Qualitätsmerkmal (attributierter Fakt)
5. Beschreibung des attributierten Fakts
6. Liste der beeinflussten Aktivitäten
7. Bewertungsvorschrift
8. weitere Qualitätsmerkmale. Ist ein konkretes Attribut vererbt, wird die Beschreibung hier nicht wiederholt (ausgezeichnet durch △).

5.3 Organisatorische Aspekte

Dieser Fakt beschreibt Aspekte, die nicht das zu wartenden System selbst, sondern die Organisation, die das System wartet, betreffen.

5.3.1 Beautifier

Werkzeug zur automatischen Quelltextformatierung.

Kontext: Beautifier ► Entwicklungsumgebung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es ein Werkzeug zur automatischen Quelltextformatierung?

Beeinflusste Aktivitäten

- *Coding*: Ein Beautifier nimmt dem Entwickler die Arbeit, den Quelltext zu formatieren, ab. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.2 Change-Management

Der Change-Management-Prozess.

Kontext: Change-Management ▷ Prozessdefinitionen ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Existiert ein definierter Change-Management-Prozess?

Beeinflusste Aktivitäten

- *Change-Management*: Eine klare Definition des Change-Management-Prozess macht ein strukturiertes Change-Management erst möglich. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Verbindlichkeit

Ist die Einhaltung des Change-Management-Prozesses verpflichtend?

Beeinflusste Aktivitäten

- *Change-Management*: Eine Change-Management-Strategie kann nur erfolgreich umgesetzt werden, wenn es keinen Möglichkeiten gibt »am Prozess vorbei« zu arbeiten. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.3 Change-Management

Ein Change-Management-Werkzeug erleichtert das Change-Management indem es den Change-Management-Prozess abbildet und die beteiligten Benutzer durch den Prozess führt.

Kontext: Change-Management ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

> Existiert ein Change-Management-Werkzeug?
>
> **Beeinflusste Aktivitäten**
> - *Change-Management (positiv)*
>
> **Bewertung (manuell)**
> Diese Eigenschaft muss manuell überprüft werden.

Kongruenz

> Lässt sich der Change-Management-Prozess des Projektes mit dem Change-Management-Werkzeug abbilden?
>
> **Beeinflusste Aktivitäten**
> - *Change-Management*: Für eine effiziente Durchführung des Change-Managements muss das Change-Management den definierten Prozess unterstützen. *(positiv)*
>
> **Bewertung (manuell)**
> Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

> Bietet das Change-Management-Werkzeug die Möglichkeit, Change-Management-Aktivitäten nachzuvollziehen.
>
> **Beeinflusste Aktivitäten**
> - *Prozessoptimierung*: Um Prozessoptimierungen durchzuführen, muss das Change-Management-Werkzeug die Möglichkeit bieten, alle Aktivitäten nachzuvollziehen. *(positiv)*
> - *Change-Management*: Das Change-Management-Werkzeug muss die Möglichkeit bieten, alle Aktivitäten nachzuvollziehen um ein effizientes Change-Management durchzuführen. *(positiv)*
>
> **Bewertung (manuell)**
> Diese Eigenschaft muss manuell überprüft werden.

5.3.4 Code-Assistenz

Code-Assistenz bietet die Möglichkeiten bestimmte Assisstenz-Funktionen während der Eingabe von Quelltext aufzurufen. So wird der Entwickler z.B. bei der Auswahl von Methoden, die ein Objekt anbietet, unterstützt.

Kontext: Code-Assistenz ► Entwicklungsumgebung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Bietet die Entwicklungsumgebung Code-Assisstenz?

Beeinflusste Aktivitäten

- *Coding*: Code-Assisstenz beschleunigt die Eingabe und hilft, Fehler zu vermeiden. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.5 Code-Vervollständigung

Die automatische Code-Vervollständigung werden Eingabe.

Kontext: Code-Vervollständigung ► Entwicklungsumgebung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Bietet die Entwicklungsumgebung Code-Vervollständigung?

Beeinflusste Aktivitäten

- *Coding*: Code-Vervollständigung beschleunigt die Eingabe und hilft, Fehler zu vermeiden. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.6 Debugger

Ein Debugging-Werkzeug *(Debugger)* dient der gezielten Fehlersuche in Programmen.

Kontext: Debugger ► Laufzeit ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Ist ein *Debugger* für die verwendeten Programmiersprachen vorhanden?

Beeinflusste Aktivitäten

- *Debugging*: Ein Debugging-Werkzeug unterstützt die Fehlersuche. Bestimmte Fehlersuch-Aktivitäten sind realistischerweise nur mit einem Debugger durchzuführen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.7 Dokumentations-Konvention

Die Dokumentations-Konvention beschreibt Format-Vorgaben für die Dokumentation.

Kontext: Dokumentations-Konvention ► Richtlinien ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es eine Konvention für das Dokumentations-Format?

Beeinflusste Aktivitäten

- *Erstellung*: Die Erstellung von Dokumentation wird durch Formatdefinitionen und Vorlagen unterstützt. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.8 Fluktuation

Die Mitarbeiter-Fluktuation.

Kontext: Fluktuation ► Mitarbeiter ► Organisatorische Aspekte ► Situation

Umfang

Das Ausmaß der Personal-Fluktuation.

Beeinflusste Aktivitäten

- *Wissensmanagement*: Das Wissensmanagement wird von der Personal-Fluktuation beeinflusst. Je größer die Fluktuation ist, desto wichtiger ist es, dass das Wissensmanagement sich nicht auf das persönliche Wissen bestimmter Mitarbeiter stützt. *(negativ)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.9 Generatoren

Dokumentations-Generatoren erlauben es, bestimmte Teile der System-Dokumentation auf Grundlage des Quelltextes zu generieren.

Kontext: Generatoren ► Dokumentation ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Sind Dokumentations-Generatoren für die verwendeten Sprachen vorhanden?

Beeinflusste Aktivitäten

- *Erstellung*: Die Dokumentationserstellung wird durch Generatoren wie *JavaDoc* stark vereinfacht. *(positiv)*
- *Pflege*: Das »Nachziehen« der Dokumentation wird durch Dokumentationsgeneratoren vereinfacht. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden. Für Java kann JavaDoc, für C++ Doxygen verwendet werden.

5.3.10 Inkrementeller Compiler

Ein inkrementeller Compiler muss nicht bei jeder Übersetzung das vollständige Programm übersetzten, sonder erlaubt es auch nur Teile davon zu übersetzen. Die Granularität variiert hierbei von Compiler zu Compiler. Manche Compiler können nur ganze Dateien übersetzen, manche sogar einzelne Anwendungen.

Kontext: Inkrementeller Compiler ► Entwicklungsumgebung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Ist ein inkrementeller Compiler vorhanden?

Beeinflusste Aktivitäten

- *Coding*: Ein inkrementeller Compiler, wie er z.B. bei Eclipse genutzt wird um den Quelltext bereits während der Eingabe zu übersetzen, steigert die Produktivität, da Syntaxfehler sofort erkannt werden. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.11 Komponenten-Browser

Ein Werkzeug, mit dessen Hilfe der Entwickler sich schnell einen Überblick über die Komponenten des Systems verschaffen kann.

Kontext: Komponenten-Browser ► Reverse Engineering ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es einen Komponenten-Browser.

Beeinflusste Aktivitäten

- *Abhängigkeitsanalyse*: Die Abhängigkeitsanalyse wird durch einen *Komponenten-Browser* stark erleichtert. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.12 Konfigurations-Management

Das Konfigurationsmanagement.

Kontext: Konfigurations-Management ▷ Prozessdefinitionen ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Gibt es einen definierten Konfigurations-Management-Prozess?

Beeinflusste Aktivitäten

- *Konfigurations-Management*: Für ein erfolgreiches Konfigurations-Management ist eine entsprechende Prozessdefinition erforderlich. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.13 Konsolidierung

Definierter Prozess zur Durchführung von Software-Konsolidierungen.

Kontext: Konsolidierung ► Präventive Maßnahmen ▷ Prozessdefinitionen ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Gibt es eine Prozessdefinition für Konsolidierungsmaßnahmen? Diese Definition sollte vor allem beschreiben, unter welchen Umständen eine Konsolidierung angezeigt ist.

Beeinflusste Aktivitäten

- *Prävention*: Um Konsolidierungen durchzuführen, muss eine Prozessdefinition für Konsolidierungsmaßnahmen existieren. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.14 Metrik-Werkzeuge

Metrik-Werkzeuge dienen der Bestimmung unterschiedlichster Software-Produkt-Metriken.

Kontext: Metrik-Werkzeuge ► Qualitätssicherung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Stehen Metrik-Werkzeuge zur Verfügung?

Beeinflusste Aktivitäten

- *Quelltext-Analyse*: Ist eine Metrik automatisch erhebar, erleichtert ein entsprechendes Werkzeug die Quelltextanalyse. Tatsächlich ist es bei typischen Quelltextmetriken nahezu unmöglich, eine Erhebung ohne entsprechende Werkzeuge durchzuführen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.15 Namenskonvention

Die Namenskonvention beschreibt Richtlinien für die Vergabe von Namen. Dies geht über die allgemeinen, im Fakten-Baum beschriebenen, Vorgaben hinaus und erlaubt die Definition von Projekt-spezifischen Eigenschaften.

Kontext: Namenskonvention ► Richtlinien ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es Namenskonventionen?

Beeinflusste Aktivitäten

- *Coding*: Beim Schreiben von neuen Code muss der Entwickler immer wieder Namen für neue Entitäten *entwerfen*. Diese Aktivität wird durch Namens-Konvention vereinfacht und gestaltet sich weniger fehlerträchtig. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.16 Plege-Werkzeuge

Dokumentations-Pflege-Werkzeuge unterstützen die (meist aufwändige) Pflege der Dokumentation. Aktuelle Entwicklungsumgebungen bieten hier zum Teil einige Unterstützung indem sie z.B. Refactoring-Schritte wie Umbennungen nicht nur im Quelltext, sondern auch in der Dokumentation durchführen.

Kontext: Plege-Werkzeuge ► Dokumentation ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es Werkzeuge zur Pflege der Dokumentation?

Beeinflusste Aktivitäten

- *Pflege*: Eine manuelle Pflege der Dokumentation ist sehr aufwändig, wenn das System großen Änderungen unterliegt. Werkzeuge die helfen, die Dokumentation *nachzuziehen* können diesen Aufwand stark reduzieren. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.17 Profiler

Ein Profiling-Werkzeug *(Profiler)* dient der Laufzeitanalyse des Systems (oder Teilen davon). Typischerweise untersuchte Aspekte sind Ausführungszeiten und Speicherverbrauch.

Kontext: Profiler ► Laufzeit ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Ist ein *Profiler* für die verwendeten Programmiersprachen vorhanden?

Beeinflusste Aktivitäten

- *Profiling*: Die Analyse des Laufzeitverhaltens ist in vielen Fällen ohne ein spezielles Profiling-Werkzeug extrem schwierig bis unmöglich. Speziell zur Aufdeckung von *Memory-Leaks* ist ein Profiler von großen Nutzem. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.18 Prozessmodell

Ein Prozessmodell ist die Grundlage einer präzisen Prozessdefinition.

Kontext: Prozessmodell ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Gibt es ein, den Prozessdefinitionen, zu Grunde liegendes Prozessmodell?

Beeinflusste Aktivitäten

- *Prozessoptimierung*: Ein klar definiertes Prozessmodell stellt die Basis für strukturierte Prozessoptimierungen dar. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.19 Qualitätssicherung

Qualitätssicherungsprozess

Kontext: Qualitätssicherung ▷ Prozessdefinitionen ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Gibt es eine Prozessdefinition für den Qualitätssicherungsprozess?

Beeinflusste Aktivitäten

- *Qualitätssicherung*: Ein definierter Qualitätssicherungsprozess macht die Qualitätssicherung kontrollier-, steuer- und bewertbar. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.20 Quelltext-Konvention

Die Quelltext-Konvention beschreibt Format-Vorgaben für den Quelltext.

Kontext: Quelltext-Konvention ► Richtlinien ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es eine Quelltext-Format-Konvention?

Beeinflusste Aktivitäten

- *Quelltext-Analyse*: Ein Analyse des Quelltexts hinsichtlich des Formats kann nur durchgeführt werden, wenn das gewünschte Format spezifiziert ist. *(positiv)*
- *Coding*: Der Entwickler wird durch Richtlinien unterstüzt, da er keinen eigenen Aufwand in die Definition des Formats investieren muss. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.21 Refactoring

Die automatische Unterstützung von Refactorings ist entscheidend für den produktiven Einsatz von Refactorings. Refactorings, die von aktuellen Entwicklungsumgebungen automatisch unterstützt werden:

- Rename Variable/Constant/Method/Class
- Extract Method
- Inline Method
- Extract Constant
- Pull Up Method
- Push Down Method

Kontext: Refactoring ► Entwicklungsumgebung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Automatisierungsgrad

Wie hoch ist der Automatisierungrad für Refactoring-Aufgaben?

Beeinflusste Aktivitäten

- *Änderung*: Viele Refactorings sind manuell nur sehr aufwändig durchzuführen. Hier kann entsprechende Werkzeugunterstützung die Effizienz deutlich erhöhen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.22 Richtlinien-Checker

Unter *Richtlinien-Checker* versteht man Werkzeuge, die der Überprüfung der Einhaltung von Richtlinien, wie sie in diesem Dokument definiert sind, dienen.

Kontext: Richtlinien-Checker ► Qualitätssicherung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es ein Werkzeug zur Überprüfung von Richtlinien?

Beeinflusste Aktivitäten

- *Quelltext-Analyse*: Die Quelltextanalyse kann durch automatische Konventions-Checker effizienter durchgeführt oder erst ermöglicht werden. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.23 Risiko-Analyse

Analyse möglicher Risiken.

Kontext: Risiko-Analyse ► Präventive Maßnahmen ▷ Prozessdefinitionen ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Gibt es einen Prozess zur Bewertung von Risiken. Beispiele zu betrachtender Faktoren sind:

- Verwendung unbekannter Technologien
- Einbindung neuer Mitarbeiter.

Beeinflusste Aktivitäten

- *Risikoanalyse*: Die Risiko-Analyse sollte durch einen entsprechenden Prozess definiert sein. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.24 Soziale Kompetenz

Dieser Fakt beschreibt die soziale Kompetenz *(Soft Skills)* der Mitarbeiter. Dazu gehören:

- Teamfähigkeit
- Kritikfähigkeit
- Flexibilität
- Motivation

Kontext: Soziale Kompetenz ► Fähigkeiten ► Mitarbeiter ► Organisatorische Aspekte ► Situation

Umfang

> Wie hoch ist die soziale Kompetenz der Mitarbeiter?
>
> **Beeinflusste Aktivitäten**
> - *Risikoanalyse*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Wissensmanagement*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Qualitätssicherung*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Konfigurations-Management*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Implementierung*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Change-Management*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Analyse*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
> - *Dokumentation*: Wie alle manuellen Aktivitäten wird auch diese von der sozialen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
>
> **Bewertung (manuell)**
> Diese Eigenschaft muss manuell überprüft werden.

5.3.25 Spezialisten

Auf bestimmte Bereiche spezialisierte Mitarbeiter. Die Spezialisierung kann zB erfolgen nach:

- Technologien
- Komponenten des Systems
- Abstraktionsebenen (Modellierung)
- Phasen des Entwicklungsprozesses

Kontext: Spezialisten ► Mitarbeiter ► Organisatorische Aspekte ► Situation

Existenz

Gibt es Mitarbeiter, die auf bestimmte Aufgabe spezialisiert sind?

Beeinflusste Aktivitäten

- *Risikoanalyse*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Wissensmanagement*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Qualitätssicherung*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Konfigurations-Management*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Implementierung*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Change-Management*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Analyse*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*
- *Dokumentation*: Da unterschiedliche Mitarbeiter unterschiedliche Qualifikationen haben, kann der Einsatz von *Spezialisten* die Effizienz erhöhen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

Redundanz

Gibt es für jedes Spezialgebiet mehrere Spezialisten?

Beeinflusste Aktivitäten

- *Risikoanalyse*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Wissensmanagement*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Qualitätssicherung*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Konfigurations-Management*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Implementierung*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Change-Management*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Analyse*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*
- *Dokumentation*: Um die Abhängigkeit von einzelnen Spezialisten zu veringern, sollte sicher gestellt sein, das es für die jeweiligen *Spezialtätigkeiten* mehr als einen Experten gibt. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.26 Struktur-Navigator

Ein Werkzeug, das es erlaubt im System entsprechend seinen Strukturen zu navigieren. Ein Beispiel ist das verfolgen von Kontrollflüssen des Systems.

Kontext: Struktur-Navigator ► Reverse Engineering ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es einen Struktur-Navigator?

Beeinflusste Aktivitäten

- *Abhängigkeitsanalyse*: Eine strukturelle Navigation, wie z.B. bei Eclipse, erleichtert das Aufdecken von Abhängigkeiten. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Deckt der Struktur-Navigator alle zu untersuchenden Strukturen ab. Ist dies nicht gegeben, ensteht Misstrauen auf Seiten der Benutzer.

Beeinflusste Aktivitäten

- *Abhängigkeitsanalyse*: Werden bestimmte Navigationspfade vom Werkzeug zur strukturellen Navigation nicht untersützt, kann dies zu fehlerhaften Ergebnissen der Abhängigkeitsanalyse führen. Außerdem wir das Vertrauen der Entwickler in das Werkzeug negativ beeinflusst. *(positiv)*

Bewertung (semi-automatisch)

Obwohl diese Eigenschaft durch Stichproben prinzipiell manuell untersucht werden musst, besteht die Möglichkeit der Semi-Automatisierung indem Vergleiche zu anderen Werkzeugen gebildet werden.

5.3.27 Syntaxhervorhebung

Syntaxhervorhebung bezeichnet die Möglichkeit eines Editors, bestimmte Wörter und Zeichenkombinationen in einem Text abhängig von seiner Bedeutung in unterschiedlichen Farben, Schriftarten und -Stilen darzustellen. Hervorgehoben werden Schlüsselwörter und andere Sprachelemente.

Kontext: Syntaxhervorhebung ► Entwicklungsumgebung ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Bietet die Entwicklungsumgebung Syntax-Hervorhebung?

Beeinflusste Aktivitäten

- *Code Lesen*: Syntaxhervorhebung verbessert die Lesbarkeit von Texten: Strukturen im Text sind leichter zu erkennen, das »Querlesen« wird vereinfacht, Kommentare erscheinen abgesetzt vom eigentlichen Code. *(positiv)*
- *Coding*: Durch Syntaxhervorhebung fallen Tippfehler schneller auf, da in vielen Fällen ein Tippfehler zu einer veränderten Darstellung führt. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.28 Technische Kompetenz

Dieser Fakt beschreibt die technische Qualifikation der Mitarbeiter für Wartungsaufgaben.

Kontext: Technische Kompetenz ► Fähigkeiten ► Mitarbeiter ► Organisatorische Aspekte ► Situation

Umfang

Wie hoch ist die technische Qualifizierung der Mitarbeiter?

Beeinflusste Aktivitäten

- *Risikoanalyse*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Wissensmanagement*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Qualitätssicherung*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Konfigurations-Management*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Implementierung*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Change-Management*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Analyse*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*
- *Dokumentation*: Wie alle manuellen Aktivitäten wird auch diese von der technischen Kompetenz der Mitarbeiter beeinflusst. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.3.29 Visualisierungen

Visualisierungs-Werkzeuge dienen der Visualisierung unterschiedlicher System-Eigenschaften. Betrachtete Aspekte sind z.B.:

- System-Struktur
- Kontrollfluss
- Datenfluss
- Änderungshäufigkeiten

Kontext: Visualisierungen ► Reverse Engineering ► Werkzeuge ► Infrastruktur ► Organisatorische Aspekte ► Situation

Existenz

Gibt es Werkzeuge zur Visualisierung.

Beeinflusste Aktivitäten

- *Abhängigkeitsanalyse*: Abhängigkeitsanalyse kann durch entsprechende Visualisierungen unterstützt werden. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Sind die Darstellungen der Visualisierungs-Werkzeuge vollständig. Ist dies nicht gegeben, ensteht Misstrauen auf Seiten der Benutzer.

Beeinflusste Aktivitäten

- *Abhängigkeitsanalyse*: Unvollständige Visualisierung der Abhängigkeiten könnne zu fehlerhaften Analyseergebnissen führen. *(positiv)*

Bewertung (semi-automatisch)
Obwohl diese Eigenschaft durch Stichproben prinzipiell manuell untersucht werden musst, besteht die Möglichkeit der Semi-Automatisierung indem Vergleiche zu anderen Werkzeugen gebildet werden.

5.3.30 Wartungs-Budget

Ein Wartungs-Budget ist ein Budget das speziell für Wartungs-Aktivitäten bestimmt ist.

Kontext: Wartungs-Budget ► Organisatorische Aspekte ► Situation

Umfang

Welchen Umfang hat das speziell für Wartungsaktivitäten vorgesehene Budget?

Beeinflusste Aktivitäten

- *Wartung*: Die Größe des Wartungsbudgets hat einen Einfluss auf alle Wartungsaktivitäten. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.3.31 Wissensmanagement

Strategien, Organisation und Lösungen, die dazu dienen, lokales organisationsspezifisches Wissen und globales Wissen zu verwalten und in einfacher Weise verfügbar zu machen

Kontext: Wissensmanagement ▷ Prozessdefinitionen ► Prozesse ► Organisatorische Aspekte ► Situation

Existenz

Gibt es eine Prozessdefinition für den Wissensmanagement-Prozess?

Beeinflusste Aktivitäten

- *Wissensmanagement*: Eine klare Definition des Wissensmanagement-Prozesses macht eine strukturiertes Wissensmanagement erst möglich. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4 Produkt-Artefakte

Die Artefakte, aus denen sich das Produkt zusammen setzt.

5.4.1 Algorithmen

Obwohl viele betriebliche Informationssystem relativ wenig Algorithmik enthalten, haben Algorithmen aufgrund ihrer Komplexität eine wichtige Bedeutung im Kontext der Wartung. Dies gilt inbesondere, da Algorithmen zur Abbildung von komplexer und kritische Funktionalität des Systems dienen.

Kontext: Algorithmen ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Unangemessener Algorithmus)

Stellt der verwendete Algorithmus eine angemesse Lösung für das behandelte Problem dar? Typische Probleme sind:

- Es wird ein *falscher* Algorithmus verwendet (z.B. *Sortierung* statt *Selektion*).
- Die Laufzeiteigenschaften sind ungenügend (z.B. *Bubble-Sort* statt *Merge-Sort*).
- Durch die Evolution der Software ist der Algorithmus unübersichtlich komplex geworden *(mehr Sonderfälle als Algorithmus*).

Beeinflusste Aktivitäten

- *Änderung*: Nur ein Alogrithmus, der seinem Problem angemessen ist, kann einfach verändert werden. *(positiv)*

Bewertung (manuell)

Ob ein Algorithmus angemessen ist oder nicht, kann nur manuell überprüft werden. Die Überprüfung sollte sich dabei an den, in der Beschreibung gegebenen, Punkte orientieren.

Umfang

Welchen Umfang hat ein Algorithmus?

Beeinflusste Aktivitäten

- *Code Lesen*: Je umfangreicher ein Algorithmus ist, desto schwerer ist er zu lesen. Das gilt speziell für extrem lange Algorithmen (mehrere kLOC). *(negativ)*
- *Änderung*: Überlange Algorithmen behindern deren Modifikation, da sie schlecht zu überblicken sind. *(negativ)*

Bewertung (manuell)

Es können unterschiedliche Metriken zur Bestimmung des Umfang verwendet werden:

- LOC
- # Fallunterscheidungen
- Menge der verarbeiteten Daten

Zugänglichkeit

Ist ein Algorithmus so *schlank* wie möglich gestaltet? In manchen Fällen lassen sich kompliziertere Algorithmen auf mehrere einfachere Teilalgorithmen aufteilen.

Beeinflusste Aktivitäten

- *Änderung*: Algorithmen, die nur ihre zentrale Funktionalität erfüllen und nichts Überflüssiges leisten, sind leichter zu ändern. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.4.2 Anforderungen

Die an das System gestellten Anforderungen werden als Artefakte aufgefasst, da sie üblicherweise in From von Dokumenten verwaltet werden.

Kontext: Anforderungen ▷ Produkt-Artefakte ► Situation

Korrektheit

Sind die Anforderungen korrekt dokumentiert?

Beeinflusste Aktivitäten

- *Zurückverfolgung*: Eine korrekte Dokumentation der Anforderungen erleichtert es, die aktuelle Implementierung mit den Anforderungen in Bezug zu setzen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Sind die Anforderungen vollständig dokumentiert?

Beeinflusste Aktivitäten

- *Zurückverfolgung*: Eine vollständige Dokumentation der Anforderungen erleichtert es, die aktuelle Implementierung mit den Anforderungen in Bezug zu setzen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.4.3 Annahmen

Während der Entwicklung werden die unterschiedlichsten Annahmen getroffen, deren Gültigkeit in vielen Fällen nur von begrenzter Dauer ist. Beispiele für solche Annahmen sind:

- Annahmen bzgl. des Benutzerkreises (Lokalisierung)
- Annahmen bzgl. der Anzahl der Benutzern
- Annahmen bzgl. der technischen Umgebung
- Annahmen bzgl. des möglichen Bereichs für bestimmte Datentypen

Kontext: Annahmen ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Sind grundlegende Annahmen dokumentiert?

Beeinflusste Aktivitäten

- *Risikoanalyse*: Für eine Risiko-Analyse ist es wichtig, für das System getroffenen, Annahmen zu berücksichtigen. Daher sollten diese dokumentiert sein. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Sind *alle* Annahmen dokumentiert?

Beeinflusste Aktivitäten

- *Risikoanalyse*: Die Ergebnisse der Risiko-Analyse hängen davon, wie vollständig die, für das System getroffenen, Annahmen dokumentiert sind. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden. Realistischerweise können nicht *alle* Annahmen dokumentiert sein. Daher erfordert die Überprüfung die Expertise eines Experten.

5.4.4 Anweisungen

Die im Programm verwendeten Anweisungen.

Kontext: Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Überflüssigkeit △

Überflüssige Anweisungen entstehen meist durch die Software-Evolution. Beispiele dafür sind:

- Inkrementierung einer Variablen, die danach nicht mehr gelesen wird.
- Anweisungen innerhalb einer Fallunterscheidung, die nie erreicht werden können.
- Anweisungen, die Funktionalität mehrfach implementieren, z.B. das mehrmalige Schließen einer Datei.
- Zuweisung, die keinen Effekt haben *(a=a)*

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssige Anweisungen stören das Lesen des Quelltexts. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)

Nicht alle überflüssigen Anweisungen können automatisch entdeckt werden, jedoch bietet z.B. die Eclipse-Entwicklungsumgebung relativ weitreichenden Unterstützung. So weist sie z.B. auf Zuweisungen ohne Effekt und in bestimmten Fällen auf nicht erreichbaren Code hin.

5.4.5 Ausdrücke

Ausdrücke setzen sich aus Operanden und Operatoren zusammen.

Kontext: Ausdrücke ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit

Der Wert eines Ausdrucks muss, unabhängig von der Auswertungsreihenfolge, die der C++-Standard erlaubt, eindeutig sein.

Beeinflusste Aktivitäten

- *Code Lesen*: Nicht-eindeutige Ausdrücke sind schwer zu lesen und können zu Fehlinterpretationen führen. *(positiv)*

- *Änderung*: Veränderung am Code können zur Folge haben, dass *instabile* Ausdrücke unvorhergesehene Fehler erzeugen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Überflüssigkeit △

Ist ein Ausdruck überflüssig?

Beeinflusste Aktivitäten
- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.6 Ausführung

Dieser Teil der Testfall-Dokumentation beschreibt wie der Testfall auszuführen ist.

Kontext: Ausführung ► Testfall-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Existiert für den Testfall eine Ausführungs-Dokumentation?

Beeinflusste Aktivitäten
- *Test*: Eine sauber Beschreibung der vorgesehenen Test-Durchführung erleichtert dem Tester die Arbeit. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.7 Bezeichner

Die im Programm verwendeten Bezeichern.

Kontext: Bezeichner ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Überflüssigkeit △

Ist ein Bezeichner überflüssig?

Beeinflusste Aktivitäten
- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)
Das *Identifier Dictionary* unterstützt bei der Suche nach überflüssigen Bezeichnern.

5.4.8 Bezug

Dieser Teil der Komponenten-Dokumentation gibt an, zu welcher Anforderung diese Komponente beiträgt.

Kontext: Bezug ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Ist dokumentiert zu welcher Anforderung eine Komponente beiträgt.

Beeinflusste Aktivitäten
- *Zurückverfolgung*: Um den Bezug zu den Anforderungen herzustellen zu können, muss dokumentiert sein, welche Anforderunge(n) einen Komponente umsetzt. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell bestimmt werden.

5.4.9 Bitoperationen

Bitoperationen sind Operationen, die auf der Bit-Darstellung primitiver Datentypen arbeiten.

Kontext: Bitoperationen ▷ Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit

Bit-Operationen dürfen nicht auf vorzeichenbehaftete Integers durchgeführt werden, da Ergebnis von Compiler und der Laufzeitumgebung abhängen kann.

Beeinflusste Aktivitäten
- *Änderung*: Veränderung am Code können zur Folge haben, dass *instabile* Ausdrücke unvorhergesehene Fehler erzeugen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Überflüssigkeit △

5.4.10 Boolsche Ausdrücke

Boolsche Ausdrücke sind Ausdrücke, deren Typ Bool ist. Zumeist enthalten diese Ausdrücke boolsche operatoren.

Kontext: Boolsche Ausdrücke ▷ Ausdrücke ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit △

Vollständigkeit (Klammerung)

Boolesche Ausdrücke sind vollständig zu klammern.

Beeinflusste Aktivitäten

- *Code Lesen*: Geklammerte Ausdrücke sind leichter zu lesen, da die Auswertungsreihenfolge der Operatoren nicht beachtet werden muss. *(positiv)*
- *Änderung*: Änderungen an Ausdrücken die nicht vollständig geklammert sind können leicht zu unerwünschten Effekten führen. *(positiv)*

Bewertung (potentiell automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Zugänglichkeit (Nebenwirkungen)

Innerhalb eines Ausdrucks mit dem Type Bool sollten keine Zuweisungen gemacht werden.

Beeinflusste Aktivitäten

- *Code Lesen*: Zuweisungen innerhalb eines boolschen Ausdrucks sind schwer zu lesen, da hier zwei unterschiedliche Ziele (Auswertung des Ausdrucks, Zuweisung) gleichzeitig erkannt werden müssen. *(positiv)*
- *Änderung*: Zuweisungen innerhalb eines boolschen Ausdrucks können bei Änderungen leicht so modifziert werden, dass sie zu fehlerhaften Ergebnissen führen. *(positiv)*

Bewertung (potentiell automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Überflüssigkeit △

5.4.11 Case-Zweig

Ein *Case-Zweig* beschreibt einen Zweig einer *Switch-Anweisung*.

Kontext: Case-Zweig ► Switch-Anweisung ▷ Fallunterscheidungen ▷ Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Vollständigkeit △

Der Code nach einem *case*-Label muß - abgesehen von den folgenden Ausnahmen - immer mit 'break' abgeschlossen werden. Ausnahmen:

- Wenn mehrere Fälle exakt dieselbe Bearbeitung haben, können die *case*-Labels unmittelbar aufeinander folgen. In diesem Falle sollte jedoch der Kommentar *fall-through* auf dieses Verhalten aufmerksam machen.
- Hinter einem Statement, das ein Verlassen der *switch*-Anweisung zur Folge hat (z.B. *return*), braucht kein *break* zu stehen.

Beeinflusste Aktivitäten

- *Code Lesen*: Ein klar abgeschlossener Case-Zweig lässt sich leichter lesen als unübersichtliche *Fall-Through*-Konstruktionen. *(positiv)*
- *Änderung*: Bei einem klar abgeschlossenen Case-Zweig ist das Risiko, Fehler während der Änderung zu erzeugen kleiner. *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

5.4.12 Datenstrukturen

Datenstrukturen

Kontext: Datenstrukturen ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Falsche Datenstruktur)

Ist die verwendete Datenstruktur ihrer Verwendung angemessen? Typische Probleme sind:
- Es wird eine *falsche* Datenstruktur verwendet (z.B. *List* statt *Hash-Tabelle*).
- Die Laufzeiteigenschaften sind ungenügend (z.B. *verkettete List* statt *Heap*).
- Durch die Evolution der Software ist die Datenstruktur unübersichtlich komplex geworden.

Beeinflusste Aktivitäten
- *Code Lesen*: Ein ungeschickt gewählte Datenstruktur erschwert das Lesen des Quelltexts, weil sich der Fokus des Lesers meist mehr auf die durch die Datenstruktur bedingten Probleme als auf ihre eigentlich Aufgabe richtet. *(positiv)*
- *Änderung*: Ungeschickte gewählte Datenstrukturen führen meist zu unötiger Komplexität in der Implementierung. Durch diese Komplexität besteht bei Änderungen ein erhöhtes Maß an Fehleranfälligkeit. *(positiv)*

Bewertung (manuell)
Ob eine Datenstruktur angemessen ist oder nicht, kann nur manuell überprüft werden. Die Überprüfung sollte sich dabei an den, in der Beschreibung gegebenen, Punkte orientieren.

5.4.13 Einrückung

Die im Quelltext verwendete Einrückung.

Kontext: Einrückung ► Format ► Programmtext ► Statik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Wird Einrückung verwendet?

Beeinflusste Aktivitäten
- *Code Lesen*: Die Strukturierte Einrückung des Quelltextes erleichtert das Lesen. *(positiv)*

Bewertung (automatisch)
Beim konsequenten Einsatz eines Beautifiers, ist diese Eigenschaft automatisch erfüllt.

Konsistenz

Ist die Einrückung konsistent?

Beeinflusste Aktivitäten
- *Code Lesen*: Die Strukturierte Einrückung des Quelltextes erleichtert das Lesen. *(positiv)*

Bewertung (automatisch)
Beim konsequenten Einsatz eines Beautifiers, ist diese Eigenschaft automatisch erfüllt.

5.4.14 Entwurfs-Entscheidungen

Während der Software-Entwicklung werden in allen Entwicklungs-Phasen immer wieder Entwurfs-Entscheidungen getroffen, die in vielen Fällen große Auswirkungen haben. Oft ist im Nachhinein nicht mehr Nachvollziehbar, warum welche Entscheidung wie getroffen wurde.

Kontext: Entwurfs-Entscheidungen ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Sind Entwurfs-Entscheidungen dokumentiert?

Beeinflusste Aktivitäten

- *Zurückverfolgung*: Oft werfen konkrete Implementierung die Frage auf, warum ein Problem auf eine bestimmte Weise gelöst wurde. Sind Entwurfsentscheidungen dokumentiert, lässt sich diese Frage beantworten. Dies gilt insbesondere bei seltsam anmutenden Implementierungen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Ist die Entwurfsentscheidungen-Dokumentation vollständig?

Beeinflusste Aktivitäten

- *Zurückverfolgung*: Eine vollständige Dokumentation der Entwurfsentscheidungen erleichtert es, die aktuelle Implementierung mit diesen Entscheidungen in Bezug zu setzen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.15 Ergebnisse

Dieser Teil der Testfall-Dokumentation beschreibt welche Ergebnisse von dem Testfall erwartet werden.

Kontext: Ergebnisse ► Testfall-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Sind die zu erwartenden Ergebnisses eines Tests dokumentiert.

Beeinflusste Aktivitäten

- *Test*: Ein Test kann nur sinnvoll durchgeführt werden, wenn die zu erwartenden Ergebnisse dokumentiert sind. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.16 Erklärung

Die *Begründung* gibt an, warum ein Testfall existiert, d.h. welche Anforderung er testet.

Kontext: Erklärung ► Testfall-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Ist die Motivation für einen Testfall dokumentiert.

Beeinflusste Aktivitäten

- *Test*: Bei der Durchführung eines Testfalls ist es wichtig zu verstehen, welchen Zweck dieser hat. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.17 FOR-Schleifen

FOR-Schleifen werden benutzt um ein Anweisungsblock mehrmals auszuführen. Normalerweise ist bei FOR-Schleifen die Anzahl der Iterationen nicht von Berechnungen innerhalb des Anweisungsblock abhängig.

Kontext: FOR-Schleifen ▷ Schleifen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit

Fliesskomma-Variablen sollten nicht als Schleifen-Zähler verwendet werden, da der Abbruchbedingungs-Test vom Compiler und der Laufzeitumgebung abhängen kann.

Beeinflusste Aktivitäten

- *Änderung*: Da die Auswertung der Abbruchbedingung vom Compiler und der Laufzeitumgebung abhängig sein kann, können bei der Modifikation unerwartete Effekte auftreten. So kann es passieren, dass durch Rundungsfehler eine Iteration mehr oder weniger ausgeführt wird. *(negativ)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Überflüssigkeit △

5.4.18 Fallunterscheidungen

Unter Fallunterscheidung versteht man Verzweigungen im Programm-Kontrollfluss.

Kontext: Fallunterscheidungen ▷ Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Überflüssigkeit △

5.4.19 Fehlerbehandlung

Der Fehlerbehandlungsmechanismus der Programmiersprache.

Kontext: Fehlerbehandlung ► Programmiersprache ► Statik ► System ▷ Produkt-Artefakte ► Situation

Verbindlichkeit

Hat die Sprache einen Fehlerbehandlungsmechanismus, der Fehlerbehanldung an bestimmten Stellen zwingend vorschreibt? Ein Beispiel ist der Exception-Mechanismus von Java.

Beeinflusste Aktivitäten

- *Änderung*: Durch die erzwungene Fehlerbehandlung können sich bei Änderungen weniger leicht, neue, unbehandelte Fehler einschleichen. *(positiv)*
- *Coding*: Die Programmierung wird vereinfacht, wenn die Sprache bestimmte Fehlerbehandlungs-Mechanismen vorschreibt, da der Entwickler sich nicht diesbezüglich keine eigenen Gedanken machen muss. Außerdem trägt diese Eigenschaft dazu bei, die Fehleranfälligtkeit zu reduzieren, da eine, zumindest minimale, Fehlerbehandlung implementiert werden muss. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.20 Fliesskomma-Variablen

Fliesskomma-Zahlen stellen des komplexesten primitiven Datentyp der meisten Programmiersprachen dar.

Kontext: Fliesskomma-Variablen ▷ Variablen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit △

Die Fliesskomma-Variablen zu Grunde liegende Bit-Repräsentation soll vom Entwickler nicht explizit genutzt werden, da Ergebnis von Compiler und der Laufzeitumgebung abhängen kann.

Beeinflusste Aktivitäten

- *Änderung*: Bei Änderungen kann Funktionalität, die auf der Bit-Repräsentation von Fliesskomma-variablen beruht, zu unerwarteten Fehlern führen. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Lokalität (Gültigkeitsbereich) △

Zugänglichkeit (Verschattung) △

Überflüssigkeit (Toter Code) △

5.4.21 Format

Das Format (Layout) der Dokumentation.

Kontext: Format ► Darstellung ► Dokumentation ▷ Produkt-Artefakte ► Situation

Konsistenz

Hat die Dokumentation eine einheitliche Aufmachung?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Ein konsistentes Format erleichert das Lesen der Dokumentation. *(positiv)*

Bewertung (manuell)
Falls die Dokumentation manuell erstellt wird, muss diese Eigenschaft manuell überprüft werden. Wird die Dokumentation jedoch generiert (z.B. *JavaDoc*) so ist sie für die generierten Teile der Dokumentation automatisch erfüllt.

Zugänglichkeit

Ist das Format der Dokumentation so gestaltet, dass sie einfach zu *benutzen* ist?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Ein leicht lesbares Format erleichtert die *Benutzung* der Dokumentation. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden.

5.4.22 Format

Das *Format* eines Bezeichners ist vor allem durch Groß-/Kleinschreibung und Separierung einzelner Wörter bestimmt.

Kontext: Format ► Bezeichner ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Kongruenz

Die IRONMAN-Programmierrichtlinie definiert eine Reihe von Vorgaben für Bezeichner:

- In Namen, welche aus mehr als einem Wort bestehen, werden dieWörter zusammengeschrieben und jeweils durch Großbuchstaben pro Wortbeginn getrennt.
- Die Namen von Klassen, Strukturen, Aufzählungstypen und zu Implementationszwecken per *typedef* definierten Typen müssen mit einem Großbuchstaben beginnen.
- Die Namen sind ausschließlich aus dem Englischen abzuleiten.
- Wenn eine Header-Datei oder Code-Datei exakt eine Klasse enthält, ist der Namensstamm ihres Dateinamens identisch mit dem Namen der Klasse zu wählen.
- Wenn eine Header-Datei oder Code-Datei mehrere Klassen enthält, ist der Namensstamm ihres Dateinamens identisch (abgesehen von einem eventuellen Suffix) mit dem Namen der wesentlichsten darin enthaltenen Klasse zu wählen. Zusätzlich kann es Dateien geben, in denen z.B. Aufzählungstypen oder Stringtypen definiert werden. Diese sollten durch den Namen als solche erkennbar sein.
- Die Namen von Konstanten, Variablen und Methoden müssen mit einem Kleinbuchstaben beginnen.
- Namen, die vom Präprozessor ausgewertet werden, sind ausschließlich mit Großbuchstaben zu schreiben. Wörter innerhalb dieser Namen sind mit Unterstreichungszeichen zu trennen.
- Namen dürfen sich nicht nur durch die Verwendung von Groß- und Kleinschreibung unterscheiden.
- Die Namen von Aufzählungstypen und zu Implementationszwecken per *typedef* definierten Typen müssen mit *Type* enden.

Beeinflusste Aktivitäten

- *Code Lesen*: Die Einhaltung der Richtlinien trägt zur Lesbarkeit des Quelltexts bei. *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

5.4.23 Funktionalität

Die Dokumentation der Funktionalität einer Komponente. Hier wird die Relation zwischen Ein- und Ausgabe-Werten einer Komponente dokumentiert.

Kontext: Funktionalität ► Black-Box-Dokumentation ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Vollständigkeit

Ist die Funktionalität einer Komponente vollständig dokumentiert?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Das Lesen der Dokumentation führt nicht zum gewünschten Ergebnis, wenn die Dokumentation der Funktionalität unvollständig ist. *(positiv)*

Bewertung (semi-automatisch)
Diese Eigenschaft muss manuell überprüft werden. Bei Java-Systemen kann mit *ConQATs JavaDocAnalyzer* überprüft werden, ob Komponenten überhaupt dokumentiert sind.

5.4.24 GOTO-Anweisungen

Im Allgemeinen geht man davon aus, dass die Verwendung von GOTO-Anweisungen zu unstrukturierten Code führt *(Goto considered harmful)*.

Kontext: GOTO-Anweisungen ▷ Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Werden GOTO-Anweisungen verwendet?

Beeinflusste Aktivitäten

- *Code Lesen*: Programme mit GOTO-Anweisungen sind schwer zu lesen. *(negativ)*
- *Änderung*: Änderungen an Programmen mit GOTO-Anweisungen können leicht zu Fehlern führen. *(negativ)*

Bewertung (automatisch)
Die Verwendung von GOTO-Anweisung wird vo Code-Inspector-Regel 13.4.6 analysiert.

Überflüssigkeit △

5.4.25 Glossar

Der Glossar beschreibt die zentrale Konzepte eines Systems, wie sie in der Software und in der Dokumentation vorkommen. Konzepte können sowohl fachlicher Natur (z.B. Kontonummer) als auch technischer Natur (z.B. Stack) sein.

Kontext: Glossar ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Konsistenz

Sind die einzelnen Glossar-Einträge konsistent, d.h. widersprechen sie sich nicht?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Inkonsistenzen im Glossar können zur Verunsicherung des Lesers führen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden.

Redundanz

Enthält der Glossar redundante Einträge?

Beeinflusste Aktivitäten

- *Änderung*: Enthält der Glossar redundant Anteile, besteht die Gefahr, dass Änderungen zu Inkonsistenzen führen. *(negativ)*

Bewertung (semi-automatisch)

Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden. Es besteht jedoch die Möglichkeit, den CloneDetective auf eine ASCII-Repräsentation des Glossars anzuwenden.

Vollständigkeit

Ist der Glossar vollständig?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Ist der Glossar unvollständig, kann er nicht seinen vollen Nutzen entfalten und das Vertrauen der Benutzer in ihn sinkt. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden.

5.4.26 If-Zero-Direktive

Die Präprozessor-Direktive *if-0* erlaubt es ganze Code-Blöcke vor dem Compiler zu *verstecken*.

Kontext: If-Zero-Direktive ▷ Präprozessor-Direktiven ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit

Die *If-0-Direktive* wird oft benutzt um darin Dokumentation zu platzieren. Das ist keine angemessene Nutzung.

Beeinflusste Aktivitäten

- *Quelltext-Analyse*: Quelltext-Analysen können oft nicht effizient ausgeführt werden, wenn *IFf-0*-Blöcke nicht mehr benutzten Code oder gar Dokumentation, die nicht der Grammatik entspricht, enthalten. *(positiv)*

Bewertung (semi-automatisch)

Explizite *If-0-Direktive* können leicht gefunden werden. Einen Überprüfung des Inhalts muss manuell erfolgen.

Überflüssigkeit △

5.4.27 Implementierung

Die Implementierung einer Methode (Glass-Box-Sicht).

Kontext: Implementierung ► Methoden ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Regularität

Ist die Struktur der Implementierung regulär? Die Struktur der Implementierung kann als ein Graph aufgefasst werden, dessen Knoten die Anweisungen der Methoden bilden. Die Kanten sind Beziehungen zwischen diesen Knoten, z.B. Aufrufbeziehungen. Das Ideal an Regularität stellt ein Graph in Form einer Kette dar. Er entspricht einer Methode, die eine Anweisung nach der anderen ausführt und weder Sprünge noch Fallunterscheidungen enthält.

Beeinflusste Aktivitäten

- *Code Lesen*: Methoden mit hoher Regularität lassen sich leichter verstehen. *(positiv)*

Bewertung (manuell)

Obwohl prinzipiell die McCabe-Metrik zur Verfügung steht, die die Regularität des Kontrollflussgraphen bestimmt, muss diese Eigenschaft manuell überprüft werden. Gründe dafür sind, das außer Kontrollflussabhängigkeiten weitere Abhängigkeiten wie Datenabhängigkeiten und temporale Abhängigkeiten bestehen. Außerdem erzeugt die McCabe-Metrik Werte, die gut zur *Testbarkeit* einer Komponente, aber weniger gut zu ihrer Verständlichkeit passen (z.B. für *Switch*-Statements). Darüberhinaus funktioniert die McCabe-Metrik nicht für polymorphe Aufrufe.

Zugänglichkeit △

Methoden mit komplizierter Logik können durch *Inline-Kommentare* leichter lesbar gemacht werden.

Beeinflusste Aktivitäten

- *Code Lesen*: Gut kommentierte Methoden erleichtern das Lesen. *(positiv)*

Bewertung (semi-automatisch)

Die Metrik *Comment-Ratio (CR)* kann mit dem *CommentRatioAnalzyer* von *ConQAT* berechnet werden.

5.4.28 Implementierung

Die Implementierung einer Klasse (Glass-Box-Sicht).

Kontext: Implementierung ► Klassen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Regularität

Ist die Struktur der Implementierung regulär? Die Struktur der Implementierung kann als ein Graph aufgefasst werden, dessen Knoten die Methoden der Klasse bilden. Die Kanten sind Beziehungen zwischen diesen Knoten, z.B. Aufrufbeziehungen. Das Ideal an Regularität stellt ein Graph in Form einer Kette dar. Er entspricht einer Klasse, die einen Einstiegspunkt (eine Methode) hat und dann eine Methode nach der anderen ausführt und weder Sprünge noch Fallunterscheidungen enthält. Es gibt aber auch andere reguläre Strukturen, so kann eine Klasse z.B. einen einfach Kontainer für eine Menge von Methoden darstellen, dir gar keine Abhängigkeiten untereinander haben.

Beeinflusste Aktivitäten

- *Code Lesen*: Klassen mit hoher Regularität lassen sich leichter verstehen. *(positiv)*

Bewertung (manuell)

Die Bewertung diese Eigenschaft kann nur manuell durchgeführt werden.

5.4.29 Include-Direktive

Die *Include*-Direktive dient bei C++ zur Inkusions externer Quelltext-Dateien.

Kontext: Include-Direktive ▷ Präprozessor-Direktiven ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Inklusionsstrukturen)

Unangemessen gewählte Inklussionstrukturen.

Beeinflusste Aktivitäten

- *Code Lesen*: Ungeschickt verwendete Inklusions-Direktive erschweren das Lesen des Codes, da sie die tatsächlichen Abhängigkeiten verschleiern. *(positiv)*
- *Übersetzung*: Unangemessen verwendete Inklusionsdirektive können zu erhöhten Übersetzungszeiten führen. Die Diplomarbeit *Optimierung von C++-Inklusionstrukturen* beschreibt die Zusammenhänge im Detail. *(positiv)*

Bewertung (semi-automatisch)

Die Diplomarbeit *Optimierung von C++-Inklusionsstrukturen* beschreibt die Auswirkung von ungeschickt gewählten C++-Inlusionsstrukturen im Detail.

Eindeutigkeit (Inklusionsstrukturen)

Inklussionstrukturen müssen eindeutig sein, d.h. sie dürfen nicht von der Reihenfolge der Inklusiondirektiven abhängen.

Beeinflusste Aktivitäten

- *Übersetzung*: In bestimmten Fällen können Inklusion-Direktiven so angeordnet werden, dass die Übersetzung des Systems nur für eine ganz bestimmte Konstellation funktioniert. Die Diplomarbeit *Optimierung von C++-Inklusionstrukturen* beschreibt die Zusammenhänge im Detail. *(positiv)*

Bewertung (semi-automatisch)

Die Diplomarbeit *Optimierung von C++-Inklusionsstrukturen* beschreibt die Auswirkung von ungeschickt gewählten C++-Inlusionsstrukturen im Detail.

Regularität

Inklude-Direktive sollten keine absoluten Dateinamen enthalten.

Beeinflusste Aktivitäten

- *Änderung*: Include-Direktive mit absoluten Dateinamen können bei Änderung zur Nicht-Übersetzbarkeit des Programms führen. *(positiv)*

Bewertung (automatisch)

Diese Eigenschaft kann mit CodeInspector-Regel 12.3.5 überprüft werden.

Überflüssigkeit (Inklusionsstrukturen) △

Überflüssige Include-Direktiven führen zur überflüssigen Inlusion von Quelltextdateien.

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Unangemessen verwendete Inklusionsdirektive können zu erhöhten Übersetzungszeiten führen. Die Diplomarbeit *Optimierung von C++-Inklusionstrukturen* beschreibt die Zusammenhänge im Detail. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)
Die Diplomarbeit *Optimierung von C++-Inklusionsstrukturen* beschreibt die Auswirkung von ungeschickt gewählten C++-Inlusionsstrukturen im Detail.

5.4.30 Index

Bei einem Index handelt es sich um ein Verzeichnis aller wichtigen Schlüsselwörter. Diesesind normalerweise alphabetisch sortiert und sind mit einem Verweis auf ihre einzelnenVorkommen versehen.

Kontext: Index ► Navigation ► Darstellung ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Verfügt die Dokumentation über einen Index?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Ein Index erleichtert das Auffinden von Information in der Dokumentation. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Ist der Index vollständig?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Ist der Index unvollständig sinkt sein Wert; vor allem weil die Entwickler das »Vertrauen« in den Index verlieren. *(positiv)*

Bewertung (manuell)
Obwohl diese Eigenschaft durch Stichproben prinzipiell manuell untersucht werden musst, besteht die Möglichkeit der Semi-Automatisierung indem Vergleiche zu anderen Werkzeugen gebildet werden.

5.4.31 Inhalt

Unter dem *Inhalt* eines Bezeichners versteht man seine lexikalische Bedeutung, die unabhängig von seinem Format ist.

Kontext: Inhalt ► Bezeichner ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit (Homonyme)

Bezeichner mit mehrern Bedeutungen werden *Homonyme* genannt.

Beeinflusste Aktivitäten

- *Code Lesen*: Beim Lesen des Quelltexts wird der Leser durch *Homonyme* gestört, da diese für unterschiedliche Konzepte stehen können. *(negativ)*

Bewertung (semi-automatisch)
Das *Identifier Dictionary* hilft der Analyse der Bezeichner und trägt dazu bei homonyme Bezeichner zu identifizieren.

Korrektheit

Ist der Bezeichner *korrekt* und *präzise* gewählt?

Beeinflusste Aktivitäten

- *Code Lesen*: Eine genaue Beschreibung der Auswirkung von suboptimal gewählten Bezeichnern enthält *Concise and Consistent Naming* von Florian Deißenböck und Markus Pizka. *(positiv)*
- *Konzept-Lokalisierung*: Die Lokalisierung von Konzepten im Quelltext wird erschwert bis unmöglich gemacht, wenn die Repräsentation der Konzepte im Quelltext nicht durch die entsprechenden Bezeichner gekennzeichnet ist. *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss manuell überprüft werden. Genauer Informationen dazu enthält *Concise and Consistent Naming* von Florian Deißenböck und Markus Pizka. An der TU gibt es derzeit Bemühungen, die Analyse von Bezeichner zu semi-automatisieren.

Redundanz (Synonyme)

Stehen meherer Bezeichner für das selbe Konzept so werden diese als *Synonyme* bezeichnet.

Beeinflusste Aktivitäten

- *Konzept-Lokalisierung*: Synonyme Bezeichner erschweren die Konzept-Lokalisierung, da der Suchraum unnötig aufgebläht wird. *(negativ)*

Bewertung (semi-automatisch)
Das *Identifier Dictionary* hilft der Analyse der Bezeichner und trägt dazu bei synonyme Bezeichner zu identifizieren.

5.4.32 Klammerung

Dieser Fakt beschreibt die Klammerung von Quelltext-Blöcken.

Kontext: Klammerung ► Format ► Programmtext ► Statik ► System ▷ Produkt-Artefakte ► Situation

Vollständigkeit

Sind Böcke vollständig geklammert.

Beeinflusste Aktivitäten

- *Code Lesen*: Eine vollständige Klammerung erlaubt es, zusammeghörige Blöcke schneller zu erfassen und erleichtert damit das Lesen. *(positiv)*

- *Änderung*: Sind zusammengehörige Blöcke vollständig geklammert, kann es bei Modifikation weniger leicht zu ungewollten Verhalten kommen (z.B. *Dangling-Else-Problem*). *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft kann mit Quelltext-Analyse-Werkzeugen überprüft werden.

5.4.33 Klassen

Dieser Fakt beschreibt die Klassen eines objekt-orientierten Systems.

Kontext: Klassen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Lokalität

Ist die Klasse *so lokal wie möglich definiert*, d.h. ist ihr Gültigkeitsbereich und ihre Sichtbarkeit minimal?

Beeinflusste Aktivitäten
- *Code Lesen*: Das Lesen des Codes vereinfacht sich, wenn Klassen so *lokal* wie möglich definiert sind, da der Leser mit Klassen, die ihn im aktuellen Context nicht betreffen, gar nicht erst in Berührung kommt. *(positiv)*
- *Abhängigkeitsanalyse*: Die Abhängigkeitsanalyse wird erleichtert, wenn Klassen so *lokal* wie möglich definiert sind, da unnötige Abhängigkeiten damit ausgeschlossen werden. *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Redundanz

Implementiert eine Klasse die selbe Funktionalität wie eine andere?

Beeinflusste Aktivitäten
- *Änderung*: Bei Änderungen an redundanten Komponenten besteht immer die Gefahr von *Update-Anomalien*. *(negativ)*

Bewertung (semi-automatisch)
Zur Bewertung gibt es einen semi-automatischen und einen manuellen Ansatz:
- Der *CloneDetective* kann genutzt werden um redundante Quelltext-Stellen zu identifzieren. Damit können auch redundante Klassen erkannt werden.
- Bei der manuell Bewertung bietet es sich an, eine erste Bewertung anhand der Klassennamen vorzunehmen.

Überflüssigkeit (Toter Code) △

Ist eine Klasse überflüssig?

Beeinflusste Aktivitäten
- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)
Bei C++ muss diese Überprüfung semi-automatisch durchgeführt werden, indem verdächtige Klassen entfernt werden und der Code dann übersetzt wird. Zu beachten ist, dass viele Programme Klassen enthalten, die nicht genutzt werden, aber für Nutzung durch Client-Software vorgesehen sind. Bei Java können ungenutzte Klassen mit *ConQATs JDependAdapterAnalyzer* entdeckt werden: Klassen, von denen keine andere Klasse abhängig ist, müssen als verdächtig gelten. Auch hier ist die potentielle externe Nutzung zu beachten. Außerdem müssen die Analyseergebnisse mit Vorsicht genossen werden, wenn das Programm *Reflection* benutzt.

5.4.34 Komma-Operator

Der Komma-Operator Bewirkt die sequenzielle Ausführung von zwei Ausdrücken.

Kontext: Komma-Operator ▷ Operatoren ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Überladung) △

Existenz

Der Komma-Operator sollte außerhalb des For-Schleifen-Kopfes nicht benutzt werden.

Beeinflusste Aktivitäten

- *Code Lesen*: Der Komma-Operator stört den Lesefluss unnötig, da das Muster »eine Anweisung pro Zeile« dadurch gebrochen wird. *(negativ)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Überflüssigkeit △

5.4.35 Komponenten-Dokumentation

Die Dokumentation einer einzelnen Komponente.

Kontext: Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Redundanz

Gibt es redundante Beschreibungen einer Komponente?

Beeinflusste Aktivitäten

- *Pflege*: Bei Änderungen an redundanten Dokumentationsteilen besteht immer die Gefahr von *Update-Anomalien*. *(positiv)*

Bewertung (semi-automatisch)
Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden. Es besteht jedoch die Möglichkeit, den CloneDetective auf eine ASCII-Repräsentation der Dokumentation anzuwenden.

5.4.36 Kopf

Der Kopf einer FOR-Schleife definiert die Zählvariabel, die Abbruchbedingung und die Inkrementierung.

Kontext: Kopf ► FOR-Schleifen ▷ Schleifen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Zugänglichkeit

Der Kopf einer FOR-Schleife sollte so einfach wie möglich gestaltet sein. Insbesondere sollte er keinen Code außer der Inkrementierung und der Abbruchbedingung enthalten.

Beeinflusste Aktivitäten

- *Code Lesen*: Ein klar verständlicher Schleifen-Kopf erleichtert das Lesen des Codes. *(positiv)*
- *Änderung*: Komplizierte Strukturen innerhalb des Schleifenkopfes, können bei Änderungen zu Fehlverhalten führen. *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

5.4.37 Label

Label dienen als Sprungziele.

Kontext: Label ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Überflüssigkeit △

Ist ein Label überflüssig?

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)
Diese Eigenschaft muss manuell überprüft werden, wird aber durch einfache Suchfunktionalität der Entwicklungsumgebung unterstützt.

5.4.38 Laufzeit

Die Laufzeit-Dokumentation einer Komponente beschreibt sowohl zu erwartenden Ausführungszeiten als auch die zu erwartende Speichernutzung.

Kontext: Laufzeit ► Black-Box-Dokumentation ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Gibt es eine Laufzeitdokumentation?

Beeinflusste Aktivitäten

- *Profiling*: Das Profiling wird erleichtert, wenn die Komponenten-Dokumentation Angaben über die zu erwartende Laufzeiteigenschaften macht. *(positiv)*
- *Änderung*: Für den Entwickler ist es sehr hilfreich, wenn Komponenten Dokumentation über Laufzeit-relevante Aspekte enthalten, da er diese bei der Änderung gesondert berücksichtigen kann. *(positiv)*

Bewertung (semi-automatisch)
Diese Eigenschaft muss manuell überprüft werden. Bei Java-Systemen kann mit *ConQATs JavaDocAnalyzer* überprüft werden, ob Komponenten überhaupt dokumentiert sind.

5.4.39 Laufzeit-Überprüfung

Von der Sprache (oder ihrer Laufzeitumgebung) durchgeführte Überprüfung, wie die Überschreitung von Array-Grenzen.

Kontext: Laufzeit-Überprüfung ► Programmiersprache ► Statik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Bietet die Sprache Laufzeit-Überprüfungen?

Beeinflusste Aktivitäten

- *Änderung*: Durch Änderungen am Code können sich neue Situation (mit neuen Fehlern) ergeben, die durch Laufzeitüberprüfungen erkannt werden. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.40 Leerzeichen

Verwendung von Leerzeichen.

Kontext: Leerzeichen ► Format ► Programmtext ► Statik ► System ▷ Produkt-Artefakte ► Situation

Konsistenz

Werden Leerzeichen zur Formatierung konsistent benutzt?

Beeinflusste Aktivitäten

- *Code Lesen*: Eine konsistente Verwendung von Leerzeichen erleichtert das Lesen des Codes. *(positiv)*

Bewertung (automatisch)
Beim konsequenten Einsatz eines Beautifiers, ist diese Eigenschaft automatisch erfüllt.

5.4.41 Leerzeilen

Benutzung von Leerzeilen zur Strukturierung.

Kontext: Leerzeilen ► Format ► Programmtext ► Statik ► System ▷ Produkt-Artefakte ► Situation

Konsistenz

Werden Leerzeilen zur Formatierung konsistent genutzt?

Beeinflusste Aktivitäten

- *Code Lesen*: Leerzeilen können durch ihren strukturierenden Character das Lesen erleichtern. *(positiv)*

Bewertung (automatisch)
Beim konsequenten Einsatz eines Beautifiers, ist diese Eigenschaft automatisch erfüllt.

5.4.42 Literale

Unter *Literalen* versteht man unveränderliche Werte im Programm.

Kontext: Literale ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Implizität (Magic Numbers)

Literale, die eine bestimmte, implizite Bedeutung haben, werden oft als *Magic Numbers* bezeichnet. Ein Beispiel sind bestimmte Rückgabewerte einer Funktion, die einen Fehler signalisieren. Solche Literal sollten durch Konstanten explizit gemacht werden.

Beeinflusste Aktivitäten

- *Code Lesen*: Das Lesen des Programms wird durch *Magic Numbers* erschwert. *(negativ)*

Bewertung (semi-automatisch)
Da die Bedeutung der Literale eben nicht explizit ist, kann diese Eigenschaft nur manuell überprüft werden. Allerdings können sowohl der *CloneDetective* als auch der *RedundantLiteralAnalyzer* von *ConQAT* bei der Suche helfen.

Redundanz

Literale die mehrfach verwendet werden, sollte durch Konstanten ersetzt werden.

Beeinflusste Aktivitäten

- *Änderung*: Bei Änderungen an redundanten Komponenten besteht immer die Gefahr von *Update-Anomalien*. *(positiv)*

Bewertung (semi-automatisch)
Redudante Literale können mit *ConQATs RedundantLiteralAnalyzer* gefunden werden.

Überflüssigkeit △

Ist ein Literal überflüssig?

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.43 Methoden

Dieser Fakt beschreibt Methoden und Funktionen.

Kontext: Methoden ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Lokalität

Ist die Methode *so lokal wie möglich definiert*, d.h. ist ihr Gültigkeitsbereich und ihre Sichtbarkeit minimal?

Beeinflusste Aktivitäten
- *Code Lesen*: Das Lesen des Codes vereinfacht sich, wenn Methoden so *lokal* wie möglich definiert sind, da der Leser mit Methoden, die ihn im aktuellen Context nicht betreffen, gar nicht erst in Berührung kommt. *(positiv)*
- *Abhängigkeitsanalyse*: Die Abhängigkeitsanalyse wird erleichtert, wenn Methoden so *lokal* wie möglich definiert sind, da unnötige Abhängigkeiten damit ausgeschlossen werden. *(positiv)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Redundanz

Implementiert eine Methode die selbe Funktionalität wie eine andere?

Beeinflusste Aktivitäten
- *Änderung*: Bei Änderungen an redundanten Komponenten besteht immer die Gefahr von *Update-Anomalien*. *(positiv)*

Bewertung (semi-automatisch)
Zur Bewertung gibt es einen semi-automatischen und einen manuellen Ansatz:
- Der *CloneDetective* kann genutzt werden um redundante Quelltext-Stellen zu identifzieren. Damit können auch redundante Methoden erkannt werden.
- Bei der manuell Bewertung bietet es sich an, alle Methoden nach ihren Parametern und ihren Rückgabewert zu sortieren und dann eine erste Bewertung anhand der Methodennamen vorzunehmen.

Umfang

Der Umfang einer Methode.

Beeinflusste Aktivitäten
- *Code Lesen*: Überlange Methoden sind schwer zu lesen, da der Überblick verloren geht. *(negativ)*

Bewertung (automatisch)
Der Umfang einer Methode kann in LOC gemessen werden.

Zugänglichkeit (Überschreibung)

Überrscheibt eine Methode eine Methode, die in einer der Superklassen definiert wurde? Methoden-Überschreibung kann im Sinne der Vererbung sehr sinnvoll genutzt werden, solange es sich um tatsächliche Spezialisierung handelt. Aktuelle Programmiersprachen verhindern jedoch nicht, dass bei der Methoden-Überschreibung die ursprüngliche Semantik vollständig umdefiniert wird. Dies führt zu schwer verstehbaren Strukturen. Um dies zu vermeiden besteht z.B. die Möglichkeit Methoden-Überschreibung nur für abstrakte Methoden zuzulassen.

Beeinflusste Aktivitäten

- *Code Lesen*: Überschriebene Methode erschweren das Lesen des Codes, da in vielen Fällen nicht sofort ersichtlich ist, dass eine Methode überschrieben wurde. *(positiv)*
- *Abhängigkeitsanalyse*: Überschriebene Methode erschweren die Abhängigkeitsanalyse da sie Mensch und Werkzeug vor komplizierte zu analysierende Strukturen stellen. *(positiv)*

Bewertung (automatisch)

Überschriebene Methoden können mit entsprechenden Werkzeugen automatisch identifiziert werden. Der Dokumentations-Generator für Java (*JavaDoc*) zeichnet überschriebene Methode entsprechend aus.

Überflüssigkeit (Toter Code) △

Ist eine Method überflüssig?

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)

Bei C++ muss diese Überprüfung semi-automatisch durchgeführt werden, indem verdächtige Methoden entfernt werden und der Code dann übersetzt wird. Zu beachten ist, dass viele Programme Methoden enthalten, die nicht genutzt werden, aber für Nutzung durch Client-Software vorgesehen sind. Bei Java werden private ungenutzte Methoden von *ConQAT* und Eclipse erkannt.

5.4.44 Nachbereich

Unter dem Nachbereich einer Schnittstelle versteht man, alle von der Benutzung der Schnittstelle potentiell betroffenen Komponenten.

Kontext: Nachbereich ► Schnittstelle ► Methoden ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Umfang

Wie groß ist der Nachbereich einer Methode?

Beeinflusste Aktivitäten

- *Abhängigkeitsanalyse*: Je Größer der Nachbereich einer Funktion ist, desto schwieriger gestaltet sich die Abhängigkeitsanalyse. *(negativ)*

Bewertung (semi-automatisch)

Die Größe des Nachbereichs einer Methode ist definiert als:

1. Rückgabewert. Dieser Wert ist automatisch bestimmbar.
2. Anzahl veränderter Objekte (transitiv). Dieser Wert kann in Abhängigkeit von der Komplexität der Methode automatisch, semi-automatisch oder u.U. sogar nur manuell bestimmt werden.

5.4.45 Nebenläufigkeit

Ausführung mehrerer paralleler Prozesse.

Kontext: Nebenläufigkeit ► Dynamik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Gibt es nebenläufige Programmanteile?

Beeinflusste Aktivitäten

- *Code Lesen*: Nebenläufige Programmanteile sind schwer zu lesen. *(negativ)*
- *Debugging*: Nebenläufige Programmanteile sind schwierig zu *Debuggen*, da mehrere Threads verfolgt werden müssen. Erschwert wird diese weiterhin, da manche Debugger dies nicht unterstützten. *(negativ)*

Bewertung (semi-automatisch)

Diese Eigenschaft muss manuell überprüft werden, jedoch besteht die Möglichkeit nebenläufige Programmanteile durch Suche nach entsprechenden Schlüsselwörter zu unterstützen. So weisen in Java z.B. die Schlüsselwörter *Thread* und *Runnable* auf nebenläufige Programmanteile hin.

5.4.46 Nebenläufigkeit

Dieser Teil der *Black-Box-Dokumentation* beschreibt nebenläufige Aspekte einer Komponente.

Kontext: Nebenläufigkeit ► Black-Box-Dokumentation ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Ist Nebenläufigkeit dokumentiert?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Das Lesen wird erleichtert, wenn die Dokumentation explizit auf nebenläufige Anteile hinweist. *(positiv)*

Bewertung (semi-automatisch)

Diese Eigenschaft muss manuell überprüft werden. Bei Java-Systemen kann mit *ConQATs JavaDocAnalyzer* überprüft werden, ob Komponenten überhaupt dokumentiert sind.

5.4.47 Operatoren

Dieser Fakt bescheibt die im Produkt verwendeten Operatoren.

Kontext: Operatoren ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Überladung)

Wird Operator-Überladung angemessen verwendet? Operator-Überladung kann zu sehr schwierig lesbaren Code führen, wenn Operatoren in unangemessener Weise überladen werden. So wäre z.B. das Überladen des +-Operators mit der Funktion *delete* fatal.

Beeinflusste Aktivitäten

- *Code Lesen*: Unangemessene überladenene Operatoren erschwerden das Lesen des Quelltext, da sich ihre Bedeutung nicht intuitiv erschliesst. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Überflüssigkeit △

Ist ein Operator überflüssig?

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (manuell)
Die Bewertung dieser Eigenschaft muss manuell durchgeführt werden.

5.4.48 Priorisierung

Dieser Teil der Testfall-Dokumentation beschreibt welche Priorität der Testfall hat.

Kontext: Priorisierung ► Testfall-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Ist die Priorität des Testfalls dokumentiert?

Beeinflusste Aktivitäten

- *Test*: Eine Priorisierung der Testfälle erlaubt eine effiziente Planung der Testreihenfolge. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.49 Präprozessor-Direktiven

Präprozessor-Direktive werden als Teil des Quelltexts betrachtet, dienen aber der Steuerung des Präprozessors und sind *nicht* Teil der Programmiersprache.

Kontext: Präprozessor-Direktiven ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Überflüssigkeit △

Ist eine Präprozessor-Direktive überflüssig? In vielen System existieren überflüssige Präprozessor-Direktiven in Form von *IFDEFs*. Meist ist dies auf Systemkonfigurationen zurückzuführen, die in der akutellen Version des Systems gar nicht mehr vorhanden sind.

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell bewertet werden.

5.4.50 Querverweise

Querverweise innerhalb der Dokumentation.

Kontext: Querverweise ► Navigation ► Darstellung ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Verfügt die Dokumentation über Querverweise?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Da Computer-Programme im Allgemeinen keine lineare Struktur haben, lassen sie sich auch schlecht als lineares, Buch-ähnliches Dokument beschreiben. Querverweise innerhalb der Dokumentation sind ein Weg, mit diesem Problem umzugehen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Alle Themen auf die sich ein Abschnitt oder ein Dokument bezieht unddie nicht in diesem selbst erklärt werden sollten durch Querverweise gekennzeichnetsein. Hierfür sollten Richtlinien existieren, die festlegen auf welche Themen verwiesenwerden muss (z.B. bei der Beschreibung einer Klasse muss auf alle Klassen, vondenen geerbt wird, alle implementierten Schnittstellen, alle bekannten Klassen zudenen eine Beziehung besteht oder die von dieser Klasse erben und alle Methodenverwiesen werden.)

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Unvollständige Querverweisstrukturen erschweren die Informationssuche und können das Vertrauen der Entwickler in die Dokumentation negativ beeinflussen. *(positiv)*

Bewertung (semi-automatisch)
Obwohl diese Eigenschaft durch Stichproben prinzipiell manuell untersucht werden musst, besteht die Möglichkeit der Semi-Automatisierung indem Vergleiche zu anderen Werkzeugen gebildet werden.

Zugänglichkeit (Hyperlinks)

Sind die Querverweise in From von Hyperlinks realisiert?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Hyperlinks erlauben eine effiziente und bequeme Navigation in der Dokumentation. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

5.4.51 Reflektion

Die Verwendung von Reflektion.

Kontext: Reflektion ► Dynamik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Wird Reflektion im Programm genutzt?

Beeinflusste Aktivitäten

- *Code Lesen*: Der Einsatz von Reflektion macht das Programm meist schlechter lesbar, da dass beim Lesen verwendete kognitive Modell um eine Ebene erweitert wird. *(negativ)*
- *Quelltext-Analyse*: Die Quelltext-Analyse von Programmen, die Reflektion verwenden, gestaltet sich schwierig, da die wenigsten Analysewerkzeuge mit Reflektion umgehen können. *(negativ)*
- *Änderung*: Änderungen von Programmen, die Reflektion verwenden, gestalten sich schwierig, da übliche Fehlerüberprüfungsmechanismen des Compilers »ausgehebelt« sind. Außerdem lassen sich Refactoring-Werkzeuge bei Reflektion nur beschränkt einsetzen. *(negativ)*

Bewertung (semi-automatisch)

Diese Eigenschaft muss manuell überprüft werden, jedoch besteht die Möglichkeit Programmteile, die Reflektion verwenden, durch Suche nach entsprechenden Schlüsselwörter zu unterstützen. So weisen in Java z.B. die Schlüsselwörter *forName* und *newInstance* auf Programmteile, die Reflektion verwenden, hin.

5.4.52 Rekursion

Rekursive Aufrufstrukturen.

Kontext: Rekursion ► Dynamik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Gibt es rekusive Aufrufstrukturen?

Beeinflusste Aktivitäten

- *Code Lesen*: Die meisten Entwickler empfinden rekursive Aufrufstrukturen als schwer verständlich. Dies ist vor allem auf das komplizierte und meist ungewohnte *kognitive Modell* zurückzuführen. *(negativ)*
- *Debugging*: Die Fehlersuche gestaltet sich für rekursive Aufrufstrukturen meist schwierig, da die Rekursion manuell *mitverfolgt* und der Aufrufstack überwacht werden muss. *(negativ)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden indem Kontrollfluss-Analysewerkzeuge eine Zyklenüberprüfung durchführen.

5.4.53 Rumpf

Der Rumpf einer FOR-Schleife enthält die auszuführenden Anweisungen.

Kontext: Rumpf ► FOR-Schleifen ▷ Schleifen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Regularität

Innerhalb des Rumpfes eine FOR-Schleife sollte keine Modifikation der Zählvariablen durchgeführt werden.

Beeinflusste Aktivitäten
- *Code Lesen*: Modifikation der Zählvariable innerhalb einer FOR-Schleifen erschweren das Lesen, da die Iterationsanzahl dadurch verändert werden kann. *(positiv)*

Bewertung (automatisch)
Ob die Zählvariable der Schleife innerhalb des Rumpfes verändert wird, kann mit der Code-Inspector-Regel 7.2.3 überprüft werden.

5.4.54 Schleifen

Schleifen sind Kontrollstrukturen, die eine wiederholte Ausführung einer oder mehrerer Anweisungen erlauben.

Kontext: Schleifen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Überflüssigkeit △

Bei Schleifen gibt es zwei Fälle von *Überflüssigkeit*:
- Eine Schleife wird nie durchlaufen. Sie ist also komplett überflüssig.
- Eine Schleife wird nur einmal durchlaufen. In diesem Falle, kann die Schleife entfernt werden, ihr Rumpf jedoch bleibt bestehen.

Beeinflusste Aktivitäten
- *Code Lesen*: Überflüssgige Komponenten verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)
Die Überflüssigkeit von Schleifen muss ihm Rahmen des Reviews bewertet werden. Zur Semi-Automatisierung dieser Bewertung können Coverage-Werkzeuge benutzt werden, die Code, der nicht ausgeführt wird, entsprechend annotieren. Diese Analyse kann aber immer nur für eine bestimmte Menge an Eingabewerten durchgeführt werden.

5.4.55 Schnittstelle

Die Schnittstelle (Aussensicht) einer Methode. Dies ist der Black-Blox-Anteil einer Methode.

Kontext: Schnittstelle ► Methoden ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Implizität (Nebenwirkungen)

Hat diese Methode unerwartete Nebenwirkungen?

Beeinflusste Aktivitäten

- *Code Lesen*: Unerwartete Nebenwirkung erschweren das Lesen des Codes. *(negativ)*
- *Änderung*: Bei Änderungen können unbekannte Nebenwirkung zu unerwartetem Fehlverhalten führen. *(negativ)*

Bewertung (potentiell automatisch)

Diese Bewertung muss manuell durchgeführt werden, jedoch könnte ein Werkzeug helfen, potentielle Nebenwirkung aufzudecken.

Konsistenz

Verwendet die Methode Parameter und Rückgabewert in zu den anderen Methoden konsistenter Weise? Gegenbeispiel sind zwei Methoden, die beide eine Modifikation einer Liste durchführen wobei einer der beiden Methoden die übergebene Liste verändert während die andere eine neue List zurück gibt.

Beeinflusste Aktivitäten

- *Code Lesen*: Sind die Schnittstellen der Methoden ähnlich gestaltet, wir das Lesen erleichtert, da die Entwickler sich auf bekannte Muster verlassen können. *(positiv)*
- *Coding*: Sind die Schnittstellen von Methoden konsistent gestaltet, so vereinfacht sich der Entwurf neuer Methoden, da die Entwicklern sich an den bereits bestehenden orientieren können. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden.

Regularität (Varargs)

Die Verwendung einer Parameterliste variabler Länge ist nur in wenigen Fällen sinnvoll und muss eindeutig motiviert sein.

Beeinflusste Aktivitäten

- *Code Lesen*: Reguläre Methodenschnittstellen erleichtern das Lesen. Eine Methode mit variabler Parameteranzahl kann schwer zu lesen sein, wenn nicht ersichtlich ist, wie diese Parameter verarbeitet werden. *(positiv)*

Bewertung (potentiell automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

5.4.56 Schnittstelle

Die Schnittstelle einer Klasse (Black-Box-Sicht).

Kontext: Schnittstelle ► Klassen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Umfang

Welchen Umfang hat die Schnittstelle einer Methode?

Beeinflusste Aktivitäten

- *Code Lesen*: Der Umfang der Schnittstelle hat einen entscheidenden Einfluss auf ihre *Verständlichkeit*. So sind z.B. Methoden mit einer großen Anzahl an Parametern schwerer zu verstehen. *(positiv)*

Bewertung (semi-automatisch)
Der Umfang der Schnittstelle setzt sich zusammen aus den Umfängen von Vorbereich und Nachbereich.

5.4.57 Shift-Operator

Shift-Operatoren führen Bit-Schiebeoperationen durch.

Kontext: Shift-Operator ▷ Operatoren ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Programmier-Tricks) △

Shift-Operationen sollten nur für geeignete Aufgaben verwendete werden. Eine nicht-angemessene Verwendung ist die Verwendung des Shift-Operators um Multiplikationen oder Divisionen durchzuführen.

Beeinflusste Aktivitäten

- *Code Lesen*: Unangemessene überladenene Operatoren erschwerden das Lesen des Quelltext, da sich ihre Bedeutung nicht intuitiv erschliesst. *(positiv)*
- *Änderung*: Wird der Shift-Operator missbraucht um z.B. Multiplikation durchzuführen, besteht bei einer Änderung des Quelltexts eine erhöhte Fehleranfälligkeit. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Überflüssigkeit △

5.4.58 Sicherheit

Die Sicherheits-Dokumentation beschreibt Sicherheits-relevante Eigenschaften einer Komponente.

Kontext: Sicherheit ► Black-Box-Dokumentation ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Sind Sicherheitseigenschaften einer Komponente dokumentiert?

Beeinflusste Aktivitäten

- *Änderung*: Für den Entwickler ist es sehr hilfreich, wenn Komponenten Dokumentation über evtl. Sicherheits-relevante Aspekte enthalten, da er diese bei der Änderung gesondert berücksichtigen kann. *(positiv)*

Bewertung (semi-automatisch)
Diese Eigenschaft muss manuell überprüft werden. Bei Java-Systemen kann mit *ConQATs JavaDocAnalyzer* überprüft werden, ob Komponenten überhaupt dokumentiert sind.

5.4.59 Size-of-Operator

In C++ wird mit dem sizeof-Operator für einen Wertetyp die entsprechende Größe in Bytes abgerufen.

Kontext: Size-of-Operator ▷ Operatoren ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Angemessenheit (Überladung) △

Korrektheit

> Die Anwendung des *sizeof* -Operators auf Ausdrücke wie *a*++ führt zu einem Fehler, da der Operator nur auf dem Typ des Ausdrucks arbeitet und die Inkrementierung nicht durchführt.
>
> **Beeinflusste Aktivitäten**
> - *Änderung*: Wird der *sizeof* -Operator inkorrekt verwendet, können bei Modifikation Fehler zu Tage treten, die zuvor nicht erkannt wurden. *(positiv)*
>
> **Bewertung (potentiell automatisch)**
> Die Bewertung muss manuell erfolgen, jedoch kann die Suche nach entsprechenden Ausdrücken durch geeignete Werkzeuge unterstützt werden.

Überflüssigkeit △

5.4.60 Sprache

Dieser Fakt beschreibt die in der Dokumentation verwendete Sprache. Hierbei ist nicht die Unterscheidung zwischen Englisch und Deutsch, sondern die Unterscheidung zwischen technisch und umgangssprachlich gemeint.

Kontext: Sprache ► Darstellung ► Dokumentation ▷ Produkt-Artefakte ► Situation

Angemessenheit

> Ist die Sprache einer technische Dokumentation angemessen?
>
> **Beeinflusste Aktivitäten**
> - *Dokumentation Lesen*: Eine, einer technischen Dokumentation angemessene Sprache, erleichtert das Lesen der Dokumentation. *(positiv)*
>
> **Bewertung (manuell)**
> Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden.

5.4.61 Struktur

Die Struktur des Systems wird meist als seine *Architektur* bezeichnet.

Kontext: Struktur ► Statik ► System ▷ Produkt-Artefakte ► Situation

Kongruenz (Architektur-Verletzungen)

Stimmt die reale Architektur mit dem Architekturentwurf überein?

Beeinflusste Aktivitäten

- *Code Lesen*: Da der Programmleser beim Lesen des Quelltexts in vielen Fällen durch sein architektonisches Wissen über das System vorgeprägt ist, stören Architekturverletzungen das Lesen. Es kann zu einer kognitiven Dissonanz zwischen der realen Implementierung und der Architekturbeschreibung kommen. *(positiv)*
- *Abhängigkeitsanalyse*: Die Abhängigkeitsanalyse stützt sich meist auf mehrere Quelle wie Code, Dokumentation und Architekturentwurf. Verletzungen der Architektur können daher die Analyse komplizieren. *(positiv)*

Bewertung (semi-automatisch)

Die Aufdeckung von Architekturverletzungen wird durch Werkzeuge wie dem *Sotograph* unterstützt. Zentrale Aktivität hierbei ist es, Abhängigkeiten (z.B. Aufrufbeziehungen) zwischen Komponenten zu finden, die laut Architekturentwurf keine Abhängigkeit haben dürften. So kann die Architektur z.B. vorschreiben, dass Komponenten aus der GUI-Schicht nich direkt auf die Datenbank-Schicht zugreifen darf. Für Java-Systemen bietet auch *ConQAT* eine einfache Bewertung von Abhängigkeiten.

5.4.62 Struktur

Die Struktur der Dokumentation.

Kontext: Struktur ► Dokumentation ▷ Produkt-Artefakte ► Situation

Kongruenz

Ist die Struktur der Dokumentation kongruent zur Struktur des Systems?

Beeinflusste Aktivitäten

- *Zurückverfolgung*: Ist die Struktur der Dokumentation *(positiv)*
- *Dokumentation Lesen*: Das Lesen der Dokumentation wird vereinfacht, wenn sich die Struktur der Dokumentation an der Struktur des System orientiert. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden. Wird die Dokumentation jedoch generiert (z.B. *JavaDoc*) so ist sie für die generierten Teile der Dokumentation automatisch erfüllt.

Vollständigkeit

Ist die Struktur der Dokumentation vollständig?

Beeinflusste Aktivitäten

- *Zurückverfolgung (positiv)*
- *Dokumentation Lesen (positiv)*

Bewertung (semi-automatisch)

Diese Eigenschaft muss durch Stichprobenexperimente manuell bestimmt werden. Wird die Dokumentation jedoch generiert (z.B. *JavaDoc*) so ist sie für die generierten Teile der Dokumentation automatisch erfüllt.

5.4.63 Strukturierung

Die Strukturierung des Quelltexts. Dazu gehört insbesondere, dass jede Anweisung in einer separaten Zeile steht.

Kontext: Strukturierung ► Format ► Programmtext ► Statik ► System ▷ Produkt-Artefakte ► Situation

Konsistenz

Ist die Strukturierung des Quelltextes einheitlich?

Beeinflusste Aktivitäten

- *Code Lesen*: Eine einheitliche Quelltextstruktur erleichtert das Lesen. *(positiv)*

Bewertung (manuell)
Beim konsequenten Einsatz eines Beautifiers, ist diese Eigenschaft automatisch erfüllt.

5.4.64 Switch-Anweisung

Switch-Anweisungen aktivieren die Ausführung einer oder mehrerer Anweisungen, wenn ein festgelegter Ausdrucks einem bestimmten Wert entspricht. Switch-Anweisungen erlauben somit *multiple branching* und erlauben es, mehrzweigige Fallunterscheidungen kompakt zu notieren.

Kontext: Switch-Anweisung ▷ Fallunterscheidungen ▷ Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Vollständigkeit (Default-Zweig)

Eine Switch-Anweisung muss einen Default-Zweig haben.

Beeinflusste Aktivitäten

- *Änderung*: Switch-Anweisungen ohne Default-Zweig können im Änderungsfalle unerwartete Ergebnisse produzieren. Selbst wenn kein *sinnvoller* Default-Zweig möglich ist, sollte daher mindestens eine Logging-Funktionalität implementiert sind, die darauf hinweist, dass ein Wert von der Switch-Anweisung nicht verarbeitet wurde. Falls das Nicht-behandeln von unbekannten Werten beabsichtigt ist, sollte der Default-Zweig einen entsprechenden Kommentar enthalten. *(positiv)*

Bewertung (automatisch)
Die Existenz des Default-Zweigs wird mit Code-Inspector-Regel 7.3.3 überprüft.

Überflüssigkeit △

5.4.65 System

Dieser Fakt beschreibt eine Gesamtsicht auf das System.

Kontext: System ▷ Produkt-Artefakte ► Situation

Redundanz (Klone)

Hat das System redundante Anteile?

Beeinflusste Aktivitäten

- *Übersetzung*: Redundante Programanteile erhöhen die Übersetzungszeit. *(negativ)*

- *Änderung*: Bei Änderungen an redundanten Programmanteile besteht Fehlegefahr, wenn nicht alle *geklonten* Stellen entsprechend geändert werden. *(negativ)*

Bewertung (semi-automatisch)
Zur Bewertung dieser Eigenschaft steht der *CloneDetective* zur Verfügung.

Umfang

Welchen Umfang hat das gesamte System?

Beeinflusste Aktivitäten
- *Code Lesen*: Je größer das System ist, desto aufwändiger gestaltet sich das Lesen. *(positiv)*
- *Übersetzung*: Je größer das System ist, desto länger dauert eine vollständige Übersetzung. *(positiv)*
- *Konzept-Lokalisierung*: Der Aufwand, ein bestimmtes Konzept im System zu lokalisieren, erhöht sich mit der Größe des Systems. *(positiv)*

Bewertung (automatisch)
Es gibt unterschiedliche Metriken, den System-Umfang zu messen:
- Lines of Code (LOC) *(ConQAT)*
- # Dateien *(ConQAT)*
- # Klassen *(ConQAT)*
- # Methoden *(ConQAT)*
- # Anweisungen *(ConQAT, auf Bytecode-Ebene)*
- # Methoden *(ConQAT)*

5.4.66 Ternäre Ausdrücke

Ternäre Ausdrücke werden meist als verkürzte Schreibweise für If-Then-Else-Anweisungen verwendet. Im Allgemeinen haben sie das Format: ((condition) ? »true«: »false«) Von den meisten Entwickler werden Ternäre Ausdrücke, im Gegensatz zu If-Then-Else-Anweisungen, als schwer lesbar beurteilt.

Kontext: Ternäre Ausdrücke ▷ Ausdrücke ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit △

Existenz

Werden ternäre Ausdrücke benutzt?

Beeinflusste Aktivitäten
- *Code Lesen*: Ternäre Ausdrücke erschweren das Lesen, da sie von vielen Entwickler als unnötig komplex aufgefasst werden. *(negativ)*

Bewertung (potentiell automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Überflüssigkeit △

5.4.67 Testfälle

Die im System vorhandenen Testfälle. Testfälle werden nicht als Teil des Endprodukts betrachtet, da sie im Normalfall nicht mit ausgeliefert werden.

Kontext: Testfälle ▷ Produkt-Artefakte ► Situation

Automatisierungsgrad

Wie stark ist ein Testfall automatisiert?

Beeinflusste Aktivitäten

- *Test*: Je stärker die einzelnen Testfälle automatisiert sind, desto weniger manueller Testaufwand fällt an. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Umfang

Der Umfang eines Testfalls bestimmt sein Ausführungszeit.

Beeinflusste Aktivitäten

- *Test*: Die Laufzeit der einzelnen Testfälle bestimmt den für das Testen benötigten Zeitaufwand. *(negativ)*

Bewertung (automatisch)
Die Ausführungszeit kann vom Test-Framework gemessen werden.

5.4.68 Type-Casts

Type-Casts sind Anweisungen, die der expliziten Konvertierung von Typen dienen.

Kontext: Type-Casts ▷ Anweisungen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Implizität (Informationsverlust)

Type-Casts sollten explizit gemacht werden, wenn die Möglichkeit des Informationsverlustes besteht, z.B. bei Casts von Typen unterschiedlicher Genauigkeit.

Beeinflusste Aktivitäten

- *Änderung*: Type-Casts mit möglichen Genauigkeitsverlust, können bei Änderungen zu unvorhersehbaren Problemen führen. Um nicht übersehen zu werden, sollten sie explizit gemacht werden. *(negativ)*

Bewertung (semi-automatisch)
CodeInspector Regel 5.2.8 hilft bei der Suche nach Type-Casts, mit denen ein Informationsverlust einhergeht.

Überflüssigkeit △

Gibt es überflüssige Type-Casts.

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssige Type-Casts verwirren den Leser. *(positiv)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden. Für Java bietet die Eclipse-Entwicklungsumbegung eine entsprechende Unterstützung.

5.4.69 Variablen

Dieser Fakt beschreibt die im Programm verwendeten Variablen.

Kontext: Variablen ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit

Jede Variable, die deklariert wird, muß einen Wert erhalten, bevor sie verwendet wird (26.5.4), da die automatische Vorbelegung von Compiler und Laufzeitumgebung abhängt.

Beeinflusste Aktivitäten

- *Änderung*: Durch die Abhängigkeiten von einer bestimmten Konstellation, kann es bei Änderungen zu unerwarteten Fehlern kommen, wenn die Variablen nicht explizit initialisiert wurden. *(positiv)*

Bewertung (potentiell automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Lokalität (Gültigkeitsbereich)

Ist die Variable *so lokal wie möglich definiert*, d.h. ist ihr Gültigkeitsbereich und ihre Sichtbarkeit minimal?

Beeinflusste Aktivitäten

- *Code Lesen*: Das Lesen des Codes vereinfacht sich, wenn Variablen so *lokal* wie möglich definiert sind, da der Leser mit Variablen, die ihn im aktuellen Context nicht betreffen, gar nicht erst in Berührung kommt. *(positiv)*
- *Abhängigkeitsanalyse*: Die Abhängigkeitsanalyse wird erleichtert, wenn Variablen so *lokal* wie möglich definiert sind, da unnötige Abhängigkeiten damit ausgeschlossen werden. *(positiv)*

Bewertung (potentiell automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Zugänglichkeit (Verschattung)

Sind Variablen verschattet? Variablen-Verschattung erfüllt in den seltensten Fällen einen tatsächlichen Zweck und ist meist nur verwirrrend.

Beeinflusste Aktivitäten

- *Code Lesen*: Verschattete Variablen erschweren das Lesen, daher sollte auf sie soweit wie möglich verzichtet werden. *(positiv)*

Bewertung (potentiell automatisch)

Beim Einsatz entsprechender Werkzeuge kann automatisch erkannt werden, ob Variablen verschattet sind.

Überflüssigkeit (Toter Code) △

Überflüssige Variablen können sowohl Variablen sein, die deklariert jedoch nie genutzt werden, als auch Variablen die zwar deklariert und genutzt werden, jedoch keine Bedeutung haben. Ein Beispiel für Letzteres sind Variablen, die durch die Evolution der Software ihre Bedeutung verloren haben, aber nie entfernt wurden.

Beeinflusste Aktivitäten

- *Code Lesen*: Überflüssige Variablen verwirren den Leser. *(negativ)*
- *Übersetzung*: Überflüssige Komponenten werden meist vom Compiler nicht als solche erkannt und erhöhen die Übersetzungszeiten unnötig. *(negativ)*
- *Konzept-Lokalisierung*: Da überflüssige Komponenten verwirrend wirken, behindern sie die Lokalisierung von Konzepten. *(negativ)*

Bewertung (semi-automatisch)

Variablen, die deklariert, aber nie genutzt werden, können mit der CodeInspector-Regel 05.01.2016 gefunden werden. Variablen ohne Bedeutung müssen ihm Rahmen eines Reviews aufgedeckt werden. CodeInspector-Regel 3.2.4 findet Variablen, auf die schreibend, aber nie lesend zugegriffen wird.

5.4.70 Variablen-Deklaration

Die Deklaration von Variablen.

Kontext: Variablen-Deklaration ► Programmiersprache ► Statik ► System ▷ Produkt-Artefakte ► Situation

Implizität

Müssen Variablen vor ihrer Benutzung explizit deklariert werden? Gegenbeispiele sind VisualBasic und Perl.

Beeinflusste Aktivitäten

- *Code Lesen*: Das Lesen wird erschwert, wenn der Programmtext Variablen enthält, die zuvor nicht explizit deklariert wurden. *(negativ)*
- *Pflege*: Änderungen an Programmtext mit nicht-deklarierten Variablen können zu unerwartetem Fehlverhalten führen. *(negativ)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden. C++ und Java erfordern explizite Variablendeklaration.

5.4.71 Verbreitung

Der Verbreitungsgrad der Programmiersprache. Werkzeugunterstützung und Support sind davon abhängig.

Kontext: Verbreitung ► Programmiersprache ► Statik ► System ▷ Produkt-Artefakte ► Situation

Umfang (Verbreitungsgrad)

Wie hoch ist der Verbreitungsgrad der Programmiersprache?

Beeinflusste Aktivitäten

- *Code Lesen*: Für weit verbreite Programmiersprachen stehen meist bessere Tools zur Verfügung. *(positiv)*
- *Quelltext-Analyse*: Für weit verbreite Programmiersprachen stehen meist bessere Tools zur Verfügung. *(positiv)*
- *Profiling*: Für weit verbreite Programmiersprachen stehen meist bessere Tools zur Verfügung. *(positiv)*
- *Debugging*: Für weit verbreite Programmiersprachen stehen meist bessere Tools zur Verfügung. *(positiv)*
- *Coding*: Für weit verbreite Programmiersprachen stehen meist bessere Tools zur Verfügung. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden. Einen kleinen Hinweis kann google.com geben:

- Suche nach *java -island -indonesia*: 293,000,000 Seiten
- Suche nach *visual basic*: 86,400,000 Seiten
- Suche nach *c++*: 73,900,000 Seiten
- Suche nach *algol*: 1,580,000 Seiten

5.4.72 Vergleiche

Vergleiche sind eine bestimmte Art von *Ausdrücken*, die mehrere Werte mit Hilfe von Vergleichs-Operatoren vergleichen.

Kontext: Vergleiche ▷ Ausdrücke ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit △

Fliesskommavariablen sollten nicht auf Gleichheit getestet werden, da diese Vergleich von der von Compiler bzw. System verwendeten Genauigkeit abhängen. Anstelle dessen sollte überprüft werden, ob ihre Differenz in einem bestimmten Intervall liegt. Enumerations-Typen sollten nur mit Enumerations-Typen verglichen werden.

Beeinflusste Aktivitäten

- *Code Lesen*: Nicht-eindeutige Ausdrücke sind schwer zu lesen und können zu Fehlinterpretationen führen. *(positiv)*
- *Änderung*: Vergleiche, der Eindeutigkeit nicht sichergestellt ist, können bei Modifikationen am Quelltext zu unerwartetem Fehlverhalten führen. *(positiv)*

Bewertung (automatisch)

Vergleiche von Fliesskommavariablen können mit CodeInspector-Regel 5.4.1 gefunden werden. Vergleiche von Enumerations-Typen können mit CodeInspector-Regel 5.2.9 gefunden werden.

Implizität

Eine Überprüfung, ob ein Wert o ist, muss explizit gemacht werden (*if (a==o)* statt *if (a)*).

Beeinflusste Aktivitäten

- *Code Lesen*: Je expliziter der Vergleich formuliert ist, desto leicher lässt er sich lesen. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Überflüssigkeit △

5.4.73 Verhaltensbeschreibung

Die *Verhaltensbeschreibung* beschreibt die Unterkomponenten einer Komponenten und ihre Interaktion, sie enthält keine explizit Beschreibung der Aussensicht.

Kontext: Verhaltensbeschreibung ► Glass-Box-Dokumentation ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Ist die Implementierung einer Komponenten dokumentiert?

Beeinflusste Aktivitäten

- *Code Lesen*: Ein klare Beschreibung der Subkomponenten einer Komponente und ihrer Zusammenhänge erleichtert das Lesen. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.74 Vermischung

Vermischung unterschiedlicher Sprachen wie im Beispiel JSP, das Java, HTML und evtl. JavaScript miteinander vermengt.

Kontext: Vermischung ► Programmiersprache ► Statik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Wird eine Vermengung unterschiedlicher Programmiersprachen verwendet?

Beeinflusste Aktivitäten

- *Code Lesen*: Das Lesen von Programme wird kann durch die Mischung mehrerer Sprachen erheblich erschwert. *(negativ)*
- *Debugging*: Programme, die aus einer Mischung mehrerer Sprachen bestehen sind äußerst kompliziert zu debuggen, da das Debugging auf unterschiedlichen *Ebenen* statt findet und Debugging-Werkzeuge meist in solch heterogenen Umgebungen nicht funktionieren. *(negativ)*
- *Abhängigkeitsanalyse*: Die Abhängigkeitsanalyse gestaltet sich bei Programmen, die aus mehreren Sprachen bestehen kompliziert, da sie von nahezu keinem Werkzeug unterstützt wird. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.75 Verschränkte Rekursion

Eine rekursive Aufrufstruktur, die sich über mehrere Komponenten erstreckt.

Kontext: Verschränkte Rekursion ▷ Rekursion ► Dynamik ► System ▷ Produkt-Artefakte ► Situation

Existenz △

Gibt es verschränkt rekursive Aufrufstrukturen?

Beeinflusste Aktivitäten

- *Code Lesen*: Verschränkte Rekursion ist aufgrund ihrerer Komplexität und Verteiltheit meist extrem schwer zu lesen. *(negativ)*
- *Debugging*: Fehlersuche in verschränkt rekursiven Aufrufstrukturen ist äußerst aufwändig. *(negativ)*
- *Änderung*: Aufgrund ihrer Komplexität sind verschränkt rekursive Aufrufstrukturen schwer zu ändern. *(negativ)*

Bewertung (semi-automatisch)
Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden indem Kontrollfluss-Analysewerkzeuge eine Zyklenüberprüfung durchführen.

5.4.76 Verteilung

Die Verteilung des Systems auf mehrere Knoten.

Kontext: Verteilung ► Dynamik ► System ▷ Produkt-Artefakte ► Situation

Existenz

Hat das System verteilte Anteile?

Beeinflusste Aktivitäten

- *Code Lesen*: Verteile Quelltextanteile sind schwer zu lesen, da die Zusammenhänge immer manuell hergestellt werden müssen. *(negativ)*
- *Profiling*: Ein Profiling verteilter Anwendung kann sich extrem aufwändig gestalten, da die Zusammenhänge sehr komplex sind. *(negativ)*
- *Debugging*: Verteilte Anwendung sind schwer zu debuggen. Wichtig ist, dass der Debugger die Untersuchung verteilter Anwendungen unterstützt. *(negativ)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

5.4.77 Volltextsuche

Eine Volltextsuche ermöglicht das vollständige Durchsuchen aller Dokumente nach bestimmtenWörtern oder Kombinationen dieser Wörter. Als Resultat werden alle Dokumentegeliefert in denen diese Wörter auftauchen. Die Ergebnisse sollten nach ihrerRelevanz sortiert sein.

Kontext: Volltextsuche ► Navigation ► Darstellung ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Bietet die Dokumentation eine Volltext-Suche-Funktion?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Eine Volltextsuchfunktion erleichtert den Zugang zur Dokumentation. *(positiv)*

Bewertung (manuell)
Diese Eigenschaft muss manuell überprüft werden.

Vollständigkeit

Durchsucht die Volltextsuche-Funktion tatsächlich die komplette Dokumentation?

Beeinflusste Aktivitäten

- *Dokumentation Lesen*: Nur wenn die Volltextsuche vollständige Ergebnisse liefert, erlangt sie das Vertrauen der Benutzer und kann somit helfen die Produktivtät zu steigern. *(positiv)*

Bewertung (semi-automatisch)
Obwohl diese Eigenschaft durch Stichproben prinzipiell manuell untersucht werden musst, besteht die Möglichkeit der Semi-Automatisierung indem Vergleiche zu anderen Werkzeugen gebildet werden.

5.4.78 Vorbereich

Der Vorbereich einer Schnittstelle ist die Menge der Komponente, auf die von der Funktion lesend zugegriffen wird.

Kontext: Vorbereich ► Schnittstelle ► Methoden ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Implizität △

Ist der Vorbereich der Methoden-Schnittstelle explizit angegeben oder macht die Methoden *versteckten* Gebrauch von weiteren Eingabedaten.

Keine beeinflusste Aktivitäten

Bewertung (manuell)
Diese Eigenschaft muss derzeit manuell überprüft werden.

Umfang

Wie groß ist der Vorbereich einer Methode?

Beeinflusste Aktivitäten

- *Code Lesen*: Die Größe des Vorbereichs einer Schnittstelle hat einen starken Einfluss auf die *Verstehbarkeit* der Schnittstelle. Selbst wenn die Schnittstelle eine einfache logische Relation beschreibt, kann sie durch die schiere Größe des Vorbereichs kompliziert zu verstehen sein. *(negativ)*
- *Abhängigkeitsanalyse*: Je Größer der Vorbereich der Schnittstelle ist, desto mehr Abhängigkeiten hat die Kompoenente. Deren Analyse erschwert sich. *(negativ)*

Bewertung (semi-automatisch)
Die Größe des Vorbereichs einer Methode ist definiert als:

1. # Parameter. Dieser Wert ist automatisch bestimmbar.

2. # verwendeter Objekte (transitiv). Dieser Wert kann in Abhängigkeit von der Komplexität der Methode automatisch, semi-automatisch oder u.U. sogar nur manuell bestimmt werden.

5.4.79 Vorraussetzungen

Dieser Teil der Testfall-Dokumentation beschreibt welche Vorraussetzungen erfüllt sein müssen, um den Testfall auszuführen.

Kontext: Vorraussetzungen ► Testfall-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

Sind die Vorraussetzungen, um einen Testfall durchzuführen, dokumentiert?

Beeinflusste Aktivitäten

- *Test*: Für die Asführung eines Test-Falls ist es wichtig, dass beschrieben ist, welche Vorraussetzungen erfüllt sein müssen, um den Testfall auszuführen. *(positiv)*

Bewertung (manuell)

Diese Eigenschaft muss manuell überprüft werden. Wird jedoch bei der Testfall-Dokumentation eine entsprechende Vorlage oder ein geeignetes Werkzeug verwendet, sollte dieser Teil der Dokumentation gezwungenermaßen vorhanden sein.

5.4.80 Vorzeichenlose Ausdrücke

Vorzeichenlose Ausdrücke sind Ausdrücke deren Typ vorzeichenlos ist.

Kontext: Vorzeichenlose Ausdrücke ▷ Ausdrücke ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit △

Der unäre Minus-Operator soll nicht auf vorzeichenlose Ausdrücke angewandt werden, da das Ergebnis von Compiler und Laufzeitumgebung abhängt.

Beeinflusste Aktivitäten

- *Code Lesen*: Nicht-eindeutige Ausdrücke sind schwer zu lesen und können zu Fehlinterpretationen führen. *(positiv)*
- *Änderung*: Veränderung am Code können zur Folge haben, dass *instabile* Ausdrücke unvorhergesehene Fehler erzeugen. *(positiv)*

Bewertung (potentiell automatisch)

Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Überflüssigkeit △

5.4.81 Wartungs-Dokumentation

Die Wartungs-Dokumentation einer Komponenten gibt Hinweise bzgl. ihrer zukünftigen Wartung. Die Wartungs-Dokumentation ist besonders wertvoll, da der ursprüngliche Entwickler bei der Erstellung meist schon eine gute Vorstellung von den Problemen hat, die bei zukünftigen Änderungen auftreten können. Wird dies nicht dokumentiert, geht dieses Wissen verloren und muss später aufwändig erarbeitet werden.

Als Teil der Wartungs-Dokumentation sind die Abhängigkeiten einer Komponenten von anderen Komponente beschrieben. Idealerweise werden dabei beide Richtung dokumentiert:

1. Wer nutzt diese Komponente?
2. Wen nutzt dieses Komponente?

In vielen Fällen kann Dokumentation dieser Art automatisch generiert werden. So bietet JavaDoc zumindest die Möglichkeit Frage 1 auf Klassenebene zu beantworten.

Kontext: Wartungs-Dokumentation ► Glass-Box-Dokumentation ► Komponenten-Dokumentation ► Inhalt ► Dokumentation ▷ Produkt-Artefakte ► Situation

Existenz

> Verfügt die Komponenten über spezifische Instruktionen für ihre Wartung?
>
> **Beeinflusste Aktivitäten**
> - *Änderung*: Die Modifikation wird stark erleichtert, wenn der ursprüngliche Autor einer Komponenten bereits dokumentiert hat, auf was bei der Änderung zu achten ist. *(positiv)*
>
> **Bewertung (manuell)**
> Diese Eigenschaft muss manuell überprüft werden.

5.4.82 Zeiger-Arithmetik

Uner *Zeiger-Arithmetik* versteht man arithmetische Operationen die nicht auf Werten, sondern auf Zeiger ausgeführt werden.

Kontext: Zeiger-Arithmetik ▷ Ausdrücke ▷ Komponenten ► Statik ► System ▷ Produkt-Artefakte ► Situation

Eindeutigkeit △

Existenz

> Wird Zeiger-Arithmetik verwendet?
>
> **Beeinflusste Aktivitäten**
> - *Code Lesen*: Zeiger-Arithmetik ist schwer zu verstehen, speziell wenn Zeiger auf Zeiger verwendet werden. *(negativ)*
> - *Änderung*: Die Veränderung von Code mit Zeiger-Arithmetik kann leicht zu unerwarteten Fehler führen. *(negativ)*
>
> **Bewertung (potentiell automatisch)**
> Diese Eigenschaft muss derzeit ihm Rahmen des Reviews überprüft werden, könnte aber mit entsprechenden Werkzeugen auch automatisch überprüft werden.

Überflüssigkeit △

6 Matrix

Ähnlich der Abbildung 2.2 geben die folgenden Seiten einen Überblick über das Modell indem sie tabellarisch die Einflussbeziehungen zwischen attributierten Fakten und Aktivitäten aufschlüsseln.

Aus layouttechnischen Gründen sind diese Seiten im Querformat angeordnet.

		Abhängigkeitsanalyse	Änderung	Change-Management	Code Lesen	Coding	Debugging	Dokumentation Lesen	Erstellung	Konfigurations-Management	Konzept-Lokalisierung	Pflege	Prävention	Profiling	Prozessoptimierung	Quelltext-Analyse	Risikoanalyse	Test	Übersetzung	Wissensmanagement	Zurückverfolgung
Algorithmen	Angemessenheit		+																		
	Umfang		–		–																
	Zugänglichkeit		+																		
Anforderungen	Korrektheit																				+
	Vollständigkeit																				+
Annahmen	Existenz																+				
	Vollständigkeit																+				
Anweisungen	Überflüssigkeit				–						–								–		
Ausdrücke	Eindeutigkeit		+		+																
	Überflüssigkeit				–						–								–		
Ausführung	Existenz																	+			
Beautifier	Existenz					+															
Bezeichner	Überflüssigkeit				–						–								–		
Bezug	Existenz																				+
Bitoperationen	Eindeutigkeit		+																		
	Überflüssigkeit				–						–								–		
Boolsche Ausdrücke	Eindeutigkeit		+		+																
	Vollständigkeit		+		+																
	Zugänglichkeit		+		+																
	Überflüssigkeit				–						–								–		
Case-Zweig	Vollständigkeit		+		+																
Change-Management	Existenz			+																	
	Verbindlichkeit			+																	
Change-Management	Existenz			+																	
	Kongruenz			+																	
	Vollständigkeit			+											+						
Code-Assistenz	Existenz					+															
Code-Vervollständigung	Existenz					+															
Datenstrukturen	Angemessenheit		+		+																
Debugger	Existenz						+														
Dokumentations-Konvention	Existenz								+												
Einrückung	Existenz				+																
	Konsistenz				+																
Entwurfs-Entscheidungen	Existenz																				+
	Vollständigkeit																				+
Ergebnisse	Existenz																	+			
Erklärung	Existenz																	+			
FOR-Schleifen	Eindeutigkeit		–																		

		Abhängigkeitsanalyse	Änderung	Change-Management	Code Lesen	Coding	Debugging	Dokumentation Lesen	Erstellung	Konfigurations-Management	Konzept-Lokalisierung	Pflege	Prävention	Profiling	Prozessoptimierung	Quelltext-Analyse	Risikoanalyse	Test	Übersetzung	Wissensmanagement	Zurückverfolgung
	Überflüssigkeit				–						–								–		
Fallunterscheidungen	Überflüssigkeit				–						–								–		
Fehlerbehandlung	Verbindlichkeit		+			+															
Fliesskomma-Variablen	Eindeutigkeit		–																		
	Lokalität	+			+																
	Zugänglichkeit				+																
	Überflüssigkeit				–						–								–		
Fluktuation	Umfang																			–	
Format	Konsistenz							+													
	Zugänglichkeit							+													
Format	Kongruenz				+																
Funktionalität	Vollständigkeit							+													
GOTO-Anweisungen	Existenz		–		–																
	Überflüssigkeit				–						–								–		
Generatoren	Existenz								+			+									
Glossar	Konsistenz							+													
	Redundanz		–																		
	Vollständigkeit							+													
If-Zero-Direktive	Angemessenheit															+					
	Überflüssigkeit				–						–								–		
Implementierung	Regularität				+																
	Zugänglichkeit				+																
Implementierung	Regularität				+																
Include-Direktive	Angemessenheit				+														+		
	Eindeutigkeit																		+		
	Regularität		+																		
	Überflüssigkeit				–						–								–		
Index	Existenz							+													
	Vollständigkeit							+													
Inhalt	Eindeutigkeit				–																
	Korrektheit				+						+										
	Redundanz										–										
Inkrementeller Compiler	Existenz					+															
Klammerung	Vollständigkeit		+		+																
Klassen	Lokalität	+			+																
	Redundanz		–																		
	Überflüssigkeit				–						–								–		
Komma-Operator	Angemessenheit				+																

		Abhängigkeitsanalyse	Änderung	Change-Management	Code Lesen	Coding	Debugging	Dokumentation Lesen	Erstellung	Konfigurations-Management	Konzept-Lokalisierung	Pflege	Prävention	Profiling	Prozessoptimierung	Quelltext-Analyse	Risikoanalyse	Test	Übersetzung	Wissensmanagement	Zurückverfolgung
	Existenz				–																
	Überflüssigkeit				–						–								–		
Komponenten-Browser	Existenz	+																			
Komponenten-Dokumentation	Redundanz											+									
Konfigurations-Management	Existenz									+											
Konsolidierung	Existenz												+								
Kopf	Zugänglichkeit		+		+																
Label	Überflüssigkeit				–						–								–		
Laufzeit	Existenz		+											+							
Laufzeit-Überprüfung	Existenz		+																		
Leerzeichen	Konsistenz				+																
Leerzeilen	Konsistenz				+																
Literale	Implizität				–																
	Redundanz		+																		
	Überflüssigkeit				–						–								–		
Methoden	Lokalität	+			+																
	Redundanz		+																		
	Umfang				–																
	Zugänglichkeit	+			+																
	Überflüssigkeit				–						–								–		
Metrik-Werkzeuge	Existenz															+					
Nachbereich	Umfang	–																			
Namenskonvention	Existenz					+															
Nebenläufigkeit	Existenz				–		–														
Nebenläufigkeit	Existenz							+													
Operatoren	Angemessenheit				+																
	Überflüssigkeit				–						–								–		
Plege-Werkzeuge	Existenz											+									
Priorisierung	Existenz																	+			
Profiler	Existenz													+							
Prozessmodell	Existenz														+						
Präprozessor-Direktiven	Überflüssigkeit				–						–								–		
Qualitätssicherung	Existenz												+		+	+		+			
Quelltext-Konvention	Existenz					+										+					
Querverweise	Existenz							+													
	Vollständigkeit							+													
	Zugänglichkeit							+													
Refactoring	Automatisierungsgrad		+																		

		Abhängigkeitsanalyse	Änderung	Change-Management	Code Lesen	Coding	Debugging	Dokumentation Lesen	Erstellung	Konfigurations-Management	Konzept-Lokalisierung	Pflege	Prävention	Profiling	Prozessoptimierung	Quelltext-Analyse	Risikoanalyse	Test	Übersetzung	Wissensmanagement	Zurückverfolgung
Reflektion	Existenz		–		–											–					
Rekursion	Existenz				–		–														
Richtlinien-Checker	Existenz															+					
Risiko-Analyse	Existenz																+				
Rumpf	Regularität				+																
Schleifen	Überflüssigkeit				–						–								–		
Schnittstelle	Implizität		–		–																
	Konsistenz				+	+															
	Regularität				+																
Schnittstelle	Umfang				+																
Shift-Operator	Angemessenheit		+		+																
	Überflüssigkeit				–						–								–		
Sicherheit	Existenz		+																		
Size-of-Operator	Angemessenheit				+																
	Korrektheit		+																		
	Überflüssigkeit				–						–								–		
Soziale Kompetenz	Umfang	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	+
Spezialisten	Existenz	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	+
	Redundanz	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	+
Sprache	Angemessenheit							+													
Struktur	Kongruenz	+			+																
Struktur	Kongruenz							+													+
	Vollständigkeit							+													+
Struktur-Navigator	Existenz	+																			
	Vollständigkeit	+																			
Strukturierung	Konsistenz				+																
Switch-Anweisung	Vollständigkeit		+																		
	Überflüssigkeit				–						–								–		
Syntaxhervorhebung	Existenz				+	+															
System	Redundanz		–																–		
	Umfang				+						+								+		
Technische Kompetenz	Umfang	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	+
Ternäre Ausdrücke	Eindeutigkeit		+		+																
	Existenz				–																
	Überflüssigkeit				–						–								–		
Testfälle	Automatisierungsgrad																	+			
	Umfang																	–			
Type-Casts	Implizität		–																		

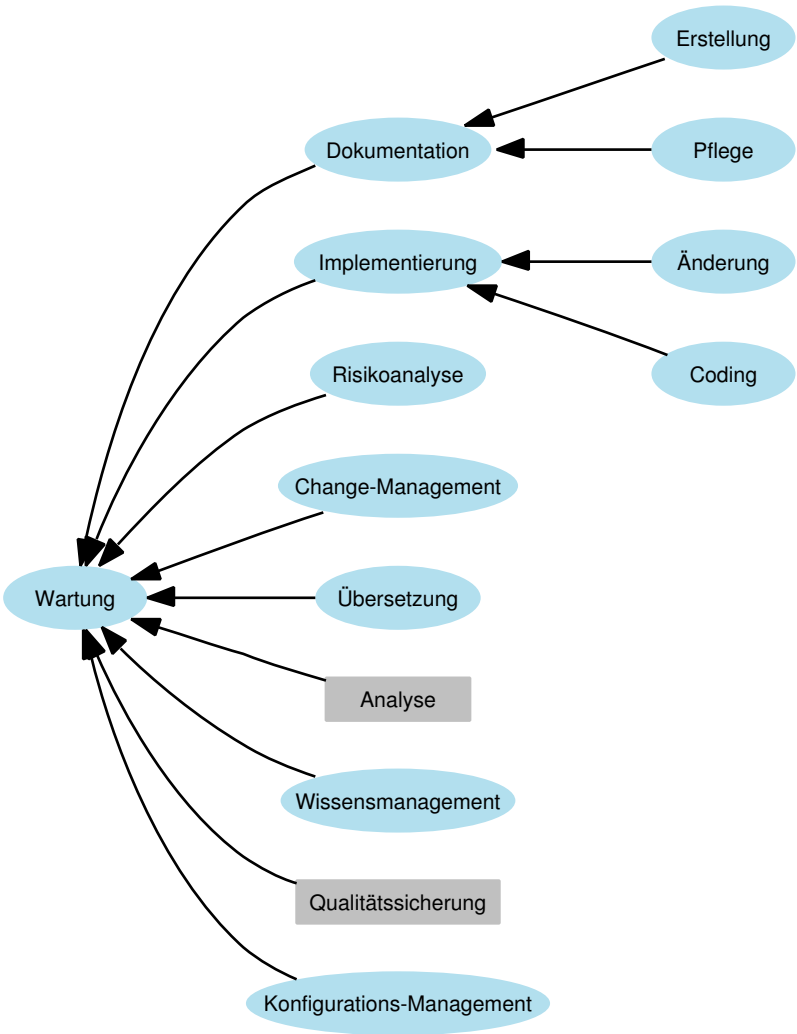
		Abhängigkeitsanalyse	Änderung	Change-Management	Code Lesen	Coding	Debugging	Dokumentation Lesen	Erstellung	Konfigurations-Management	Konzept-Lokalisierung	Pflege	Prävention	Profiling	Prozessoptimierung	Quelltext-Analyse	Risikoanalyse	Test	Übersetzung	Wissensmanagement	Zurückverfolgung
	Überflüssigkeit				+						–								–		
Variablen	Eindeutigkeit		+																		
	Lokalität	+			+																
	Zugänglichkeit				+																
	Überflüssigkeit				–						–								–		
Variablen-Deklaration	Implizität				–							–									
Verbreitung	Umfang				+	+	+							+		+					
Vergleiche	Eindeutigkeit		+		+																
	Implizität				–																
	Überflüssigkeit				–						–								–		
Verhaltensbeschreibung	Existenz				+																
Vermischung	Existenz	–			–		–														
Verschränkte Rekursion	Existenz		–		–		–														
Verteilung	Existenz				–		–							–							
Visualisierungen	Existenz	+																			
	Vollständigkeit	+																			
Volltextsuche	Existenz							+													
	Vollständigkeit							+													
Vorbereich	Implizität																				
	Umfang	–			–																
Vorraussetzungen	Existenz																	+			
Vorzeichenlose Ausdrücke	Eindeutigkeit		+		+																
	Überflüssigkeit				–						–								–		
Wartungs-Budget	Umfang	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Wartungs-Dokumentation	Existenz		+																		
Wissensmanagement	Existenz																			+	
Zeiger-Arithmetik	Eindeutigkeit		+		+																
	Existenz		–		–																
	Überflüssigkeit				–						–								–		

7 Aktivitäten-Graphen

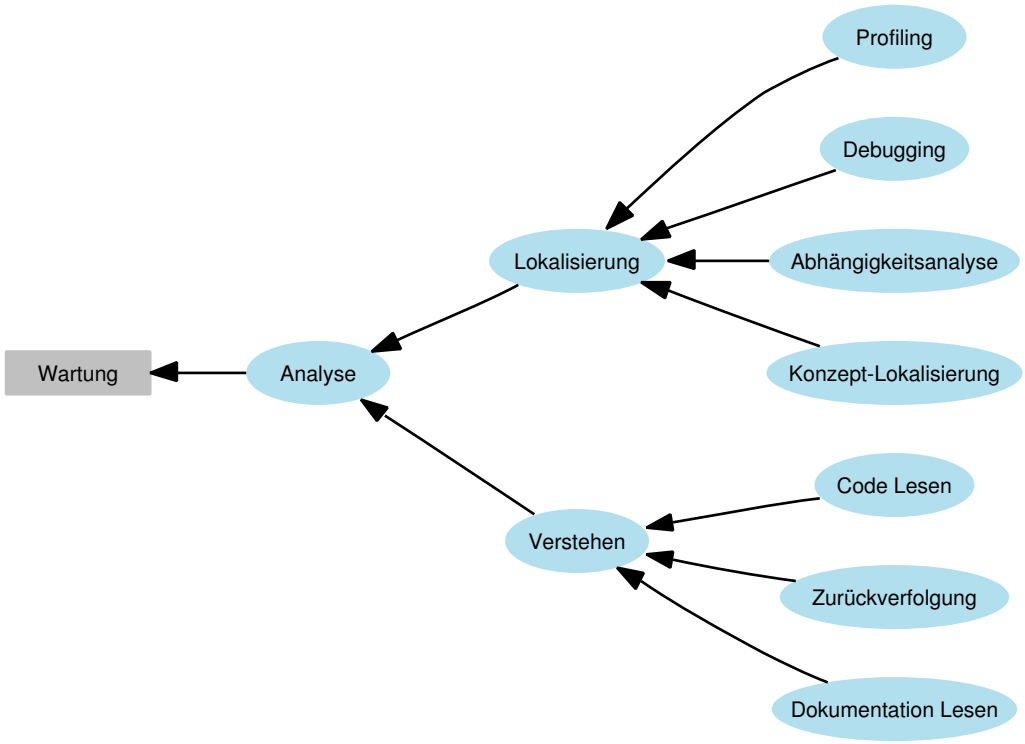
Die folgenden Seiten zeigen eine vollständige, graphische Darstellung des Aktivitätenbaums. Aus layouttechnischen Gründen ist der Baum in mehrere Teilbäumen zerlegt und auf mehrere Seiten aufgeteilt. Die Aufteilung wird von einem Algorithmus gewählt, dessen Ziel die Optimierung des verwendeten Platzes ist.

In der Darstellung symbolisiert ein rechteckiger, grauer Knoten eine Fortsetzung des Baums in einem weiteren Teilbaum. Dieser Teilbaum ist entsprechend mit dem Namen des Fortsetzungsknoten unterschrieben. Die Suche nach einem bestimmten Teilbaum kann somit vereinfacht werden, in dem nur die Bildunterschriften nach dem Knoten durchsucht werden. Aus Gründen der Übersichtlichkeit ist bei den Teilbäumen der übergeordnete Knoten noch einmal dargestellt.

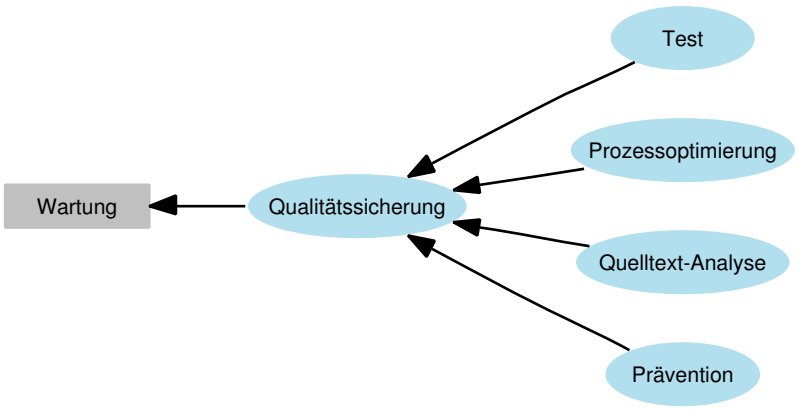
Die Reihenfolge der Teilbäume ist im Dokument so gewählt, dass jeder Pfad von der Wurzel zu einem Blatt im Dokument von vorne nach hinten verfolgt werden kann.



Wartung



Analyse



Qualitätssicherung

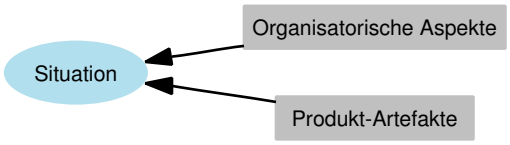
8 Situations-Graphen

Die folgenden Seiten zeigen eine vollständige, graphische Darstellung des Situationsbaums. Aus layouttechnischen Gründen ist der Baum in mehrere Teilbäumen zerlegt und auf mehrere Seiten aufgeteilt. Die Aufteilung wird von einem Algorithmus gewählt, dessen Ziel die Optimierung des verwendeten Platzes ist.

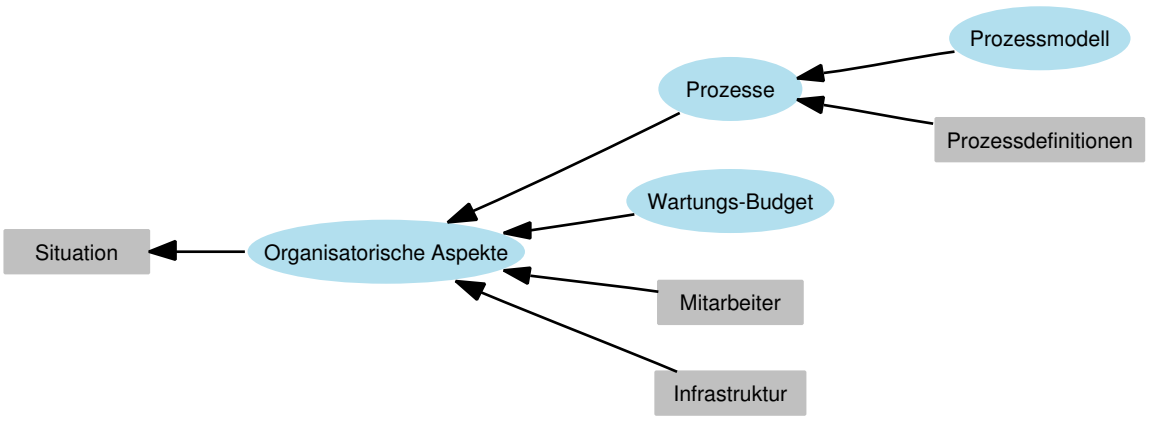
In der Darstellung symbolisiert ein rechteckiger, grauer Knoten eine Fortsetzung des Baums in einem weiteren Teilbaum. Dieser Teilbaum ist entsprechend mit dem Namen des Fortsetzungsknoten unterschrieben. Die Suche nach einem bestimmten Teilbaum kann somit vereinfacht werden, in dem nur die Bildunterschriften nach dem Knoten durchsucht werden. Aus Gründen der Übersichtlichkeit ist bei den Teilbäumen der übergeordnete Knoten noch einmal dargestellt.

Die Reihenfolge der Teilbäume ist im Dokument so gewählt, dass jeder Pfad von der Wurzel zu einem Blatt im Dokument von vorne nach hinten verfolgt werden kann.

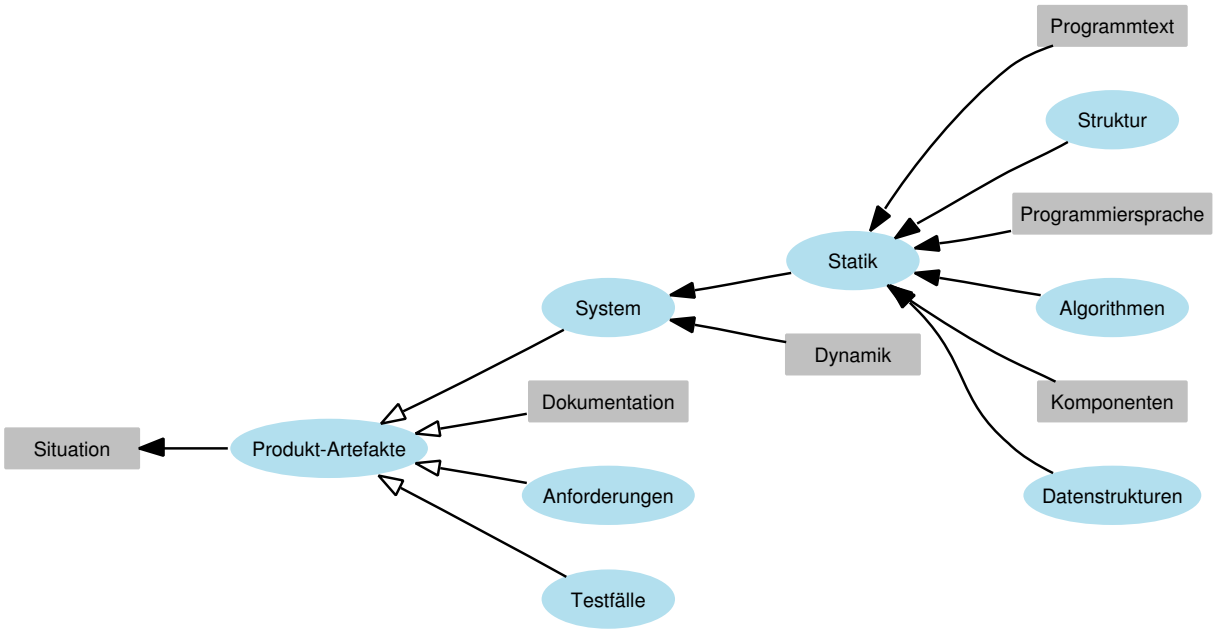
Innerhalb des Situationsbaum symbolisiert ein Pfeil mit gefüllter Spitze eine *Aggregationsbeziehung* und ein Pfeil mit ungefüllter Spitze eine *Spezialisierungsbeziehung*.



Situation

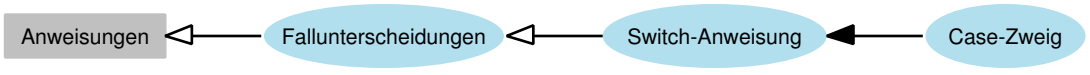


Organisatorische Aspekte

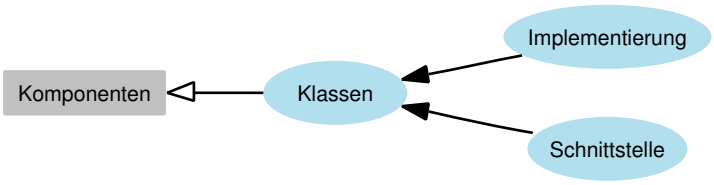


Produkt-Artefakte

Komponenten

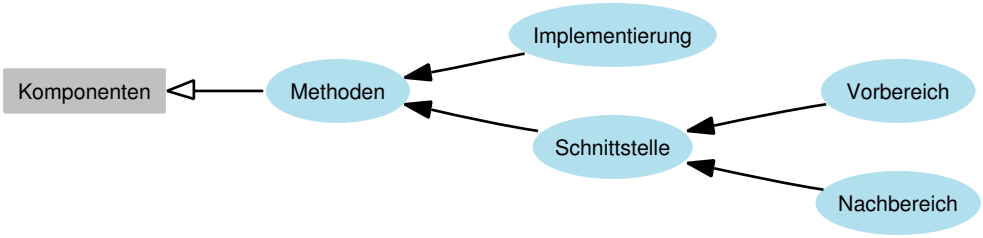


Fallunterscheidungen

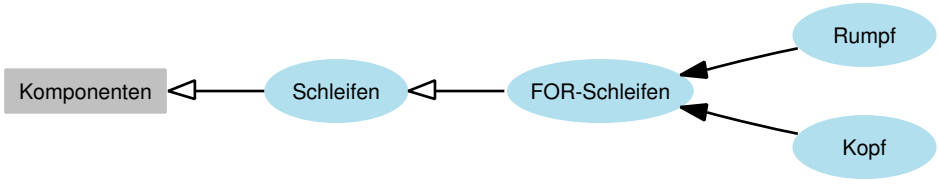


Klassen

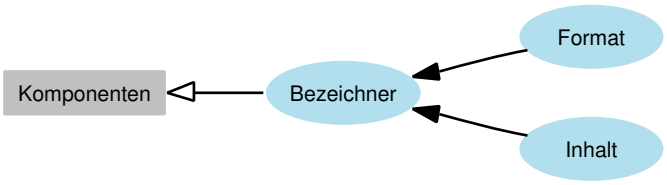
Präprozessor-Direktiven



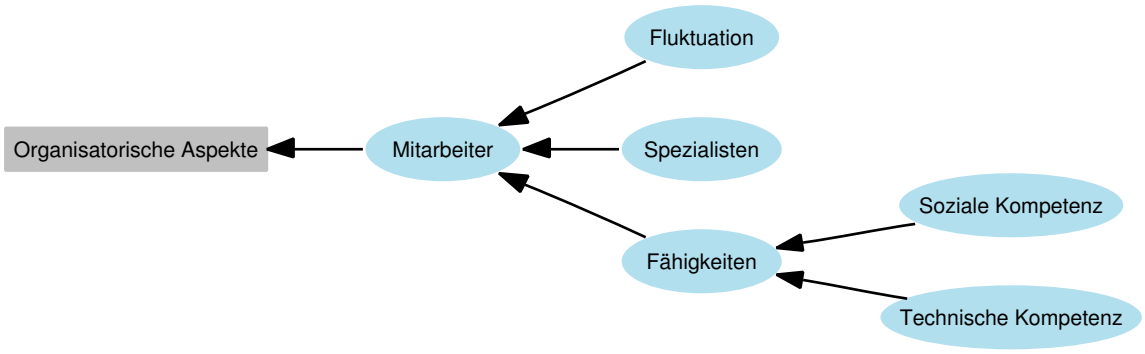
Methoden



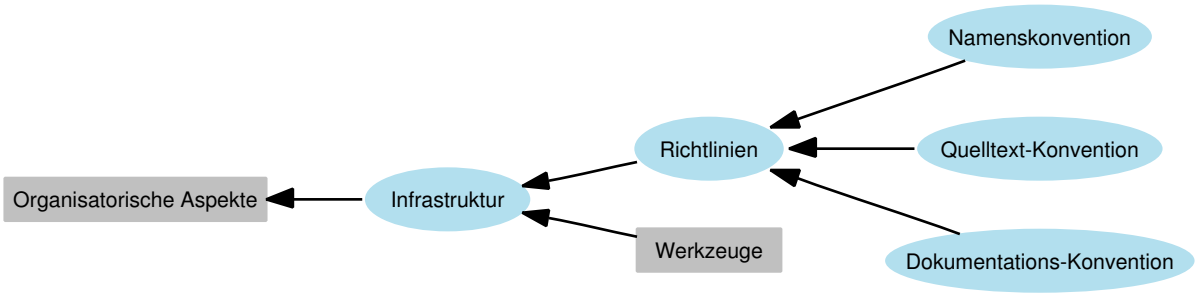
Schleifen



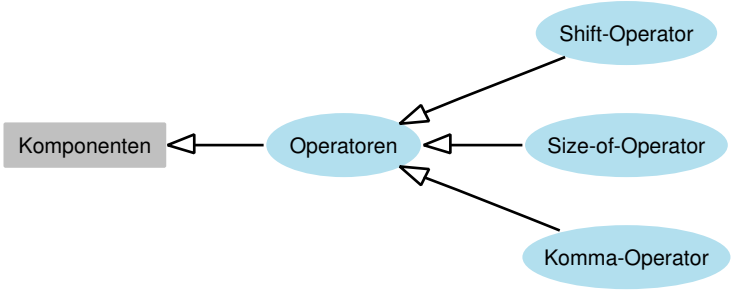
Bezeichner

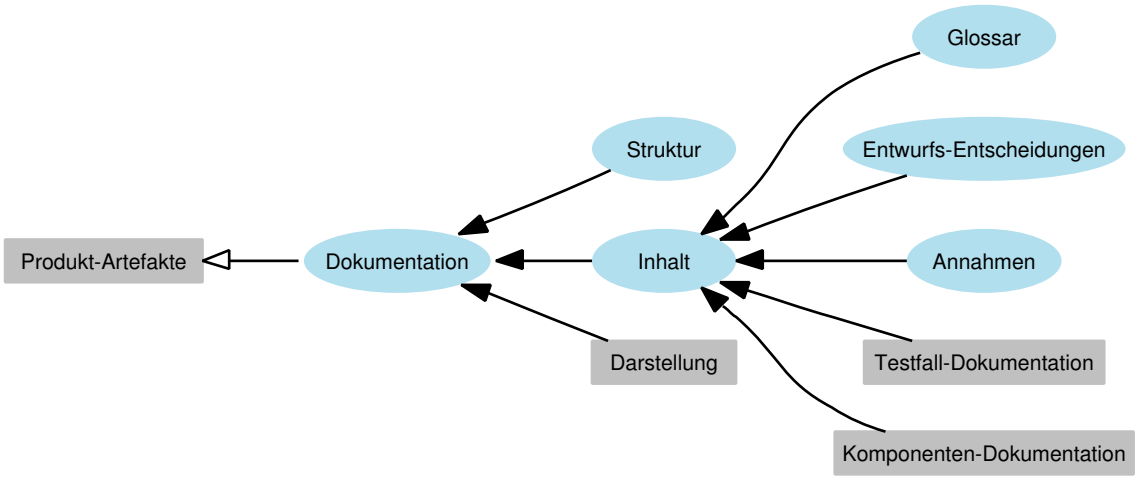


Mitarbeiter

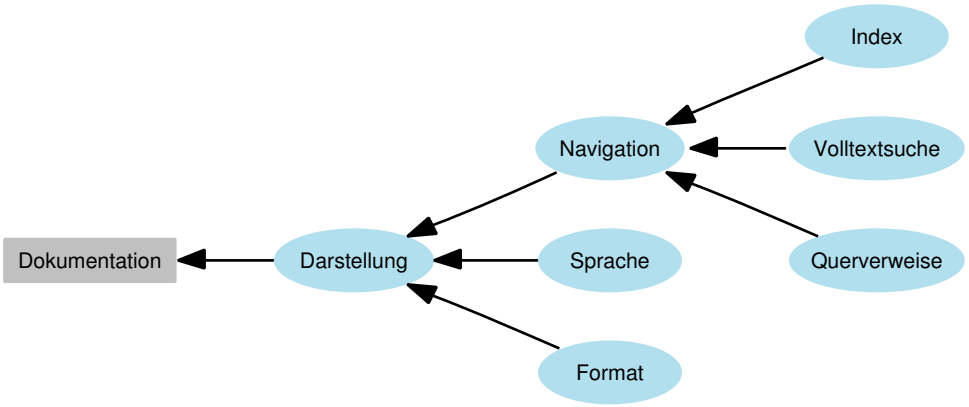


Infrastruktur

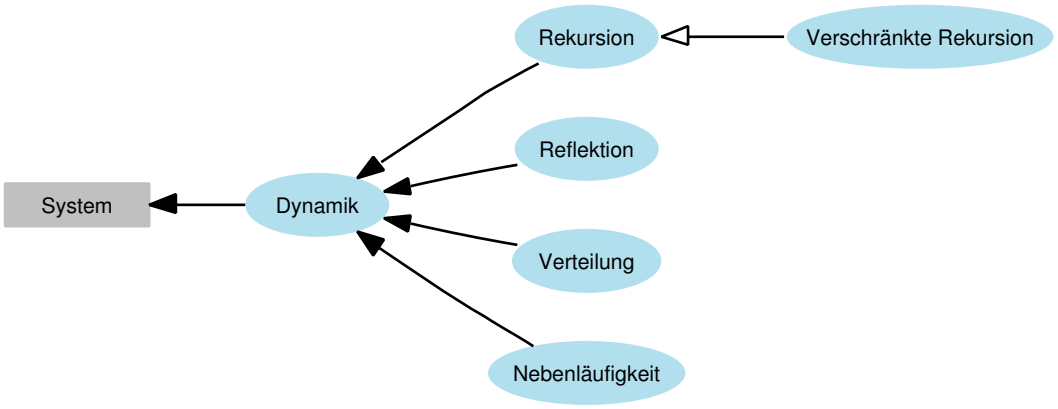




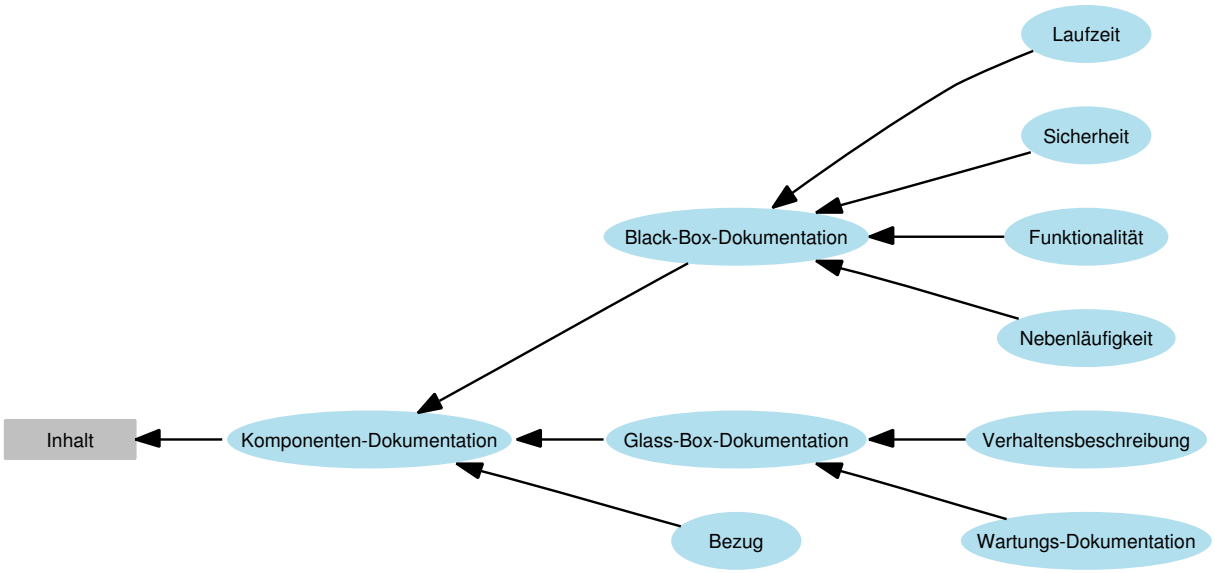
Dokumentation



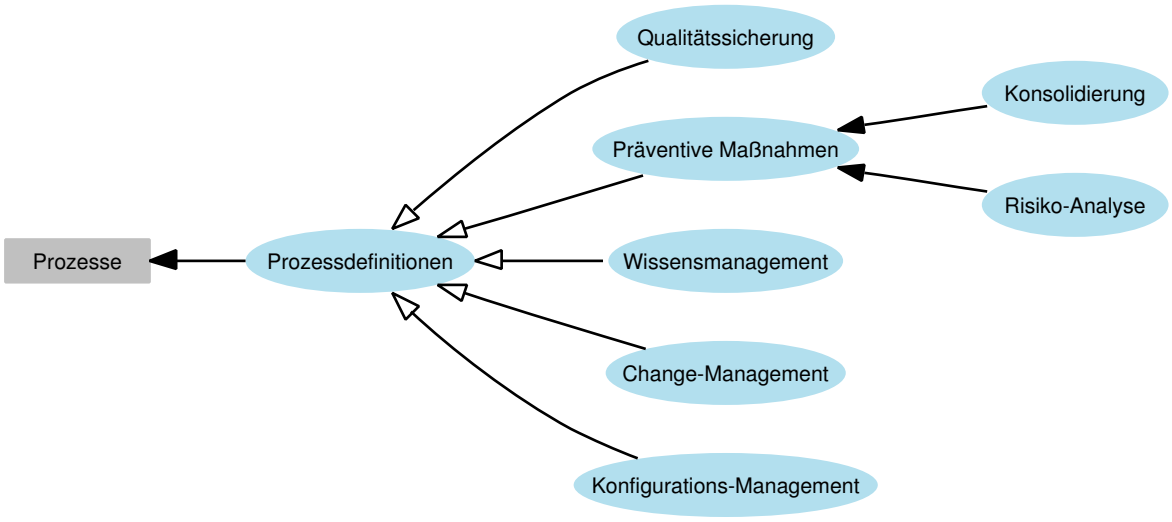
Darstellung



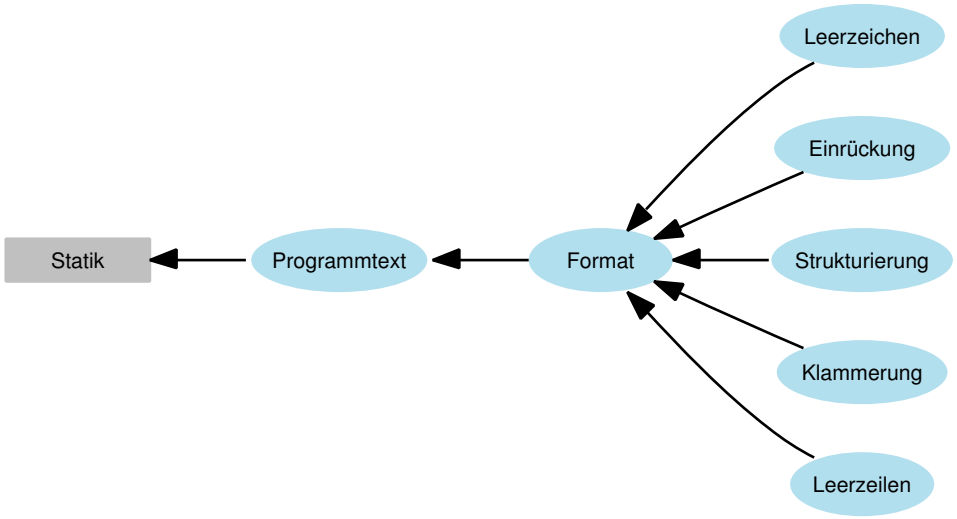
Dynamik



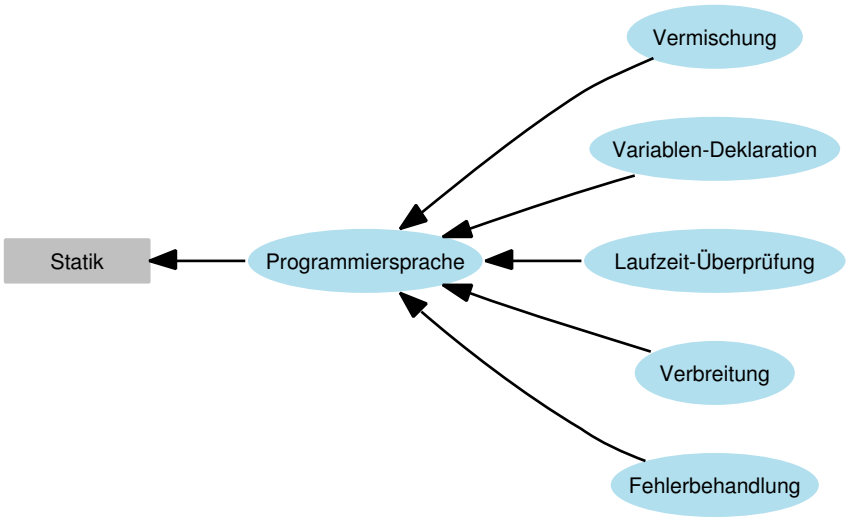
Komponenten-Dokumentation



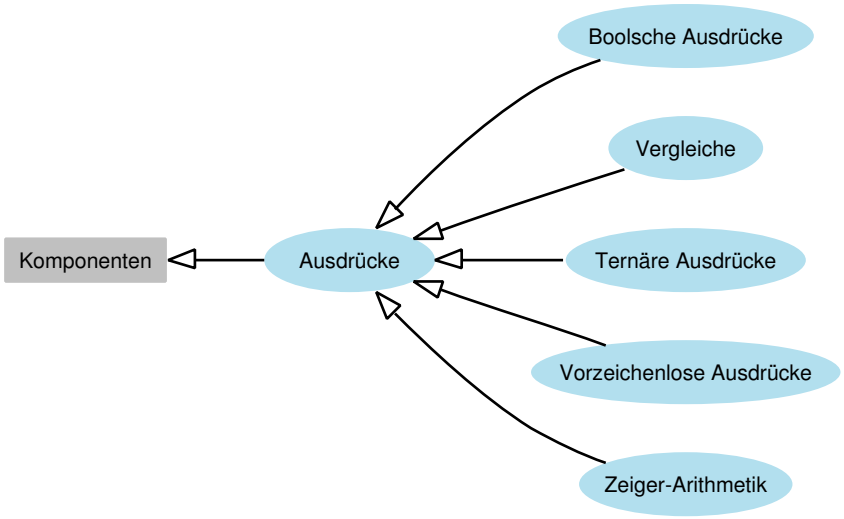
Prozessdefinitionen



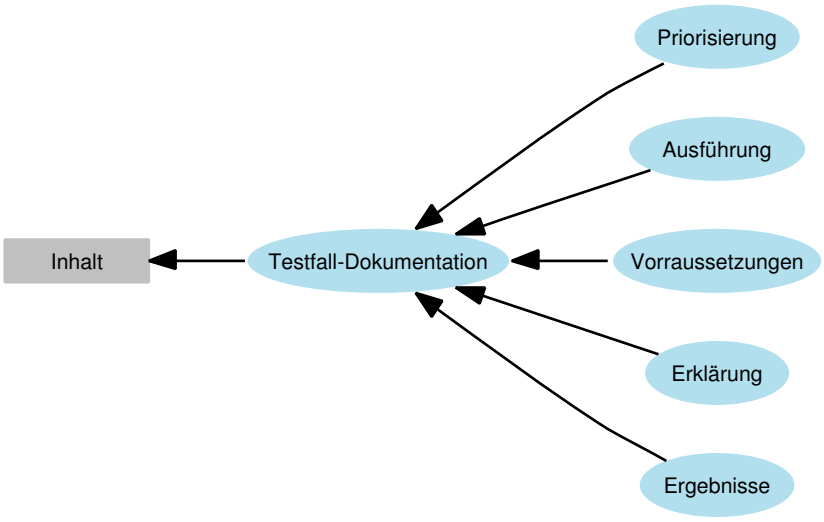
Programmtext



Programmiersprache



Ausdrücke



Testfall-Dokumentation


Generatoren
Dokumentation
Plege-Werkzeuge
Visualisierungen
Komponenten-Browser
Reverse Engineering
Struktur-Navigator
Change-Management
Infrastruktur
Werkzeuge
Richtlinien-Checker
Qualitätssicherung
Metrik-Werkzeuge
Laufzeit
Debugger
Profiler
Refactoring
Code-Vervollständigung
Entwicklungsumgebung
Code-Assistenz
Beautifier
Syntaxhervorhebung
Inkrementeller Compiler

Werkzeuge

Literaturverzeichnis

[1] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying maintainability. In *WoSQ 2006*. ACM Press, 2006. to appear.

[2] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *IWPC '00*, page 241. IEEE CS, 2000.

[3] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. In *IWPC 2005*, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Katalin Erdös and Harry M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *Proceedings of the 6th International Workshop on Program Comprehension*. IEEE CS Press, 1998.

[5] Carl S. Hartzman and Charles F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering*. IBM Press, 1993.

[6] Paul W. Oman and Curtis R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5), 1990.

[7] Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *IWPC '02*, page 271. IEEE CS, 2002.

[8] John S. Reel. Critical success factors in software projects. *IEEE Software*, 16(3):18–23, 1999.

[9] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 39–48. IEEE Computer Society Press, 1994.

[10] G. M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold Co., 1971.

Index