Institut für Softwaretechnologie (ISTE)
Abteilung Software Engineering II
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3295

Schaffung einer Basis für die kontinuierliche Qualitätsanalyse

Mehmed Metuh

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Wagner
Betreuer:	Dipl.-Ing. Jan-Peter Ostberg Christof Mayer
begonnen am:	01. Februar 2012
beendet am:	02. August 2012
CR-Klassifikation:	D.2.5, D.2.7, D.2.8, D.2.9, D.2.11, K.6.3

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Wirtschaftlich gesehen, kann es sich heutzutage kaum ein Unternehmen mehr leisten auf hohe Softwarequalität zu verzichten. Besonders für Softwaresysteme, die langlebig sein sollen, ist die Qualität enorm wichtig und wird als entscheidender Erfolgsfaktor angesehen. Erfahrungsgemäß nimmt aber die Qualität vor allem bei diesen Softwaresystemen mit der Zeit rapide ab. Die kontinuierlichen Änderungen und Erweiterungen des Quellcodes, sich ständig ändernde Anforderungen und die Anpassungen der Software an unterschiedliche technische Umgebungen können zum schleichenden Qualitätsverfall führen. Die Qualitätsdefizite, die dabei entstehen, wie die ansteigende Komplexität, die mangelhafte Dokumentation oder der Architekturverfall, sind nur einige der Mängel, die durch die oben genannten Änderungen und Erweiterungen verursacht werden. Darunter leidet am stärksten die innere Qualität des Softwaresystems. Die Wartbarkeit, die Testbarkeit und die Modifizierbarkeit werden stark erschwert und wirken sich im schlimmsten Fall negativ auf die äußeren Qualitätsmerkmale, wie Funktionalität oder Benutzbarkeit, aus.

Viele Studien zeigen [LL10], dass in der Praxis die meisten Kosten nicht während der initialen Entwicklung, sondern während der Wartungsphase bestehender Softwaresysteme auftreten. Der Kostenanteil für die Wartung wird auf 60% bis 80% der gesamten Kosten des Softwarelebenszykluses, geschätzt [Dei10]. Mehr als die Hälfte der Aufwände fallen dabei nicht für die Fehlerbehebungen oder Systemanpassungen an, sondern für das Hinzufügen oder Ändern von Funktionen [Som01]. Hierbei spielt die Wartbarkeit eine entscheidende Rolle, um diese Änderungen und Erweiterungen existierender Softwaresysteme effizient durchführen zu können. Trotz der Anerkennung der auftretenden Qualitätsdefizite und der größeren Bedeutung der Wartbarkeit, fehlt es bei vielen Unternehmen immer noch an geeigneten Methoden, Prozessen und Werkzeugen zur Sicherstellung der Qualität und vor allem der Wartbarkeit.

Eine mögliche Lösung, um diesen Qualitätsverfall entgegen zu wirken und die Wartbarkeit zu erhalten, ist, das Softwareprodukt selbst zu analysieren und zu verbessern. Durch eine kontinuierliche Softwarequalitätsanalyse können bestimmte Qualitätseigenschaften und -parameter beobachtet, analysiert und ausgewertet werden. Diese Qualitätsanalyse sollte so früh wie möglich in den Entwicklungsprozess integriert werden, um zeitnahe die Qualitätsmängel erkennen zu können. Die Einführung des Qualitäts-Controllings in den früheren Phasen der Softwareentwicklung erleichtert die Beseitigung der Qualitätsdefizite und führt zu geringeren Kosten [DH11].

Auf Grund der Größe heutiger Softwaresysteme ist es empfehlenswert solch eine Qualitäts-Überprüfung sowohl durch manuelle Inspektionen, als auch mittels dafür geeigneten Werkzeugen durchzuführen [Dei10].

Die Werkzeuge zur Operationalisierung der kontinuierlichen Softwarequalitätsanalyse werden „Software Quality Dashboards“ oder „Cockpits“ genannt. Diese Werkzeuge sammeln und liefern Daten über den aktuellen Qualitätszustand eines Softwaresystems in Form von Tabellen, Trends und Übersichtsgrafiken. Mit Hilfe dieser Daten können dann Aussagen über die Qualität und die Weiterentwicklungsfähigkeit des Softwaresystems gemacht werden.

Es existieren verschiedene Werkzeuge, die sich mit der Qualitätsanalyse von Softwaresystemen befassen. Ein Beispiel für solch ein Tool ist das für diese Diplomarbeit verwendete Open-Source-Werkzeug „ConQAT“. Durch dieses Werkzeug werden Daten in aggregierter Form gesammelt und visualisiert, um qualitativ hochwertige Aussagen über das Produkt „Software“ machen zu können. Diese Diplomarbeit beschäftigt sich mit dem „Continuous Quality Assessment Toolkit“ (ConQAT), sowie mit verschiedenen Qualitätsmetriken und Methoden, um eine Basis für eine kontinuierliche Qualitätsanalyse beim Industriepartner „XYZ“ zu schaffen.

1.2. Aufgabenstellung

Diese Diplomarbeit hat das Ziel das Continuous Quality Assessment Toolkit (ConQAT) beim Industriepartner „XYZ“ einzuführen, sowie die Integration des Werkzeugs in den Qualitätssicherungsprozess des Unternehmens. Zudem sollen auch eventuelle Verbesserungen bei der Unterstützung der Qualitätssicherung evaluiert werden. Hierfür soll zunächst der vorhandene Qualitätssicherungsprozess analysiert und die Zufriedenheit der Mitarbeiter mit diesem Prozess ermittelt werden. Je nach Ergebnis der Analyse sollen eine oder mehrere Metriken zur Verbesserung der Qualitätsbeurteilung vorgeschlagen und eventuell eingesetzt werden. ConQAT soll als Werkzeug für die Analyse und Darstellung der aggregierten Ergebnisse in den vorhandenen Qualitätssicherungsprozess eingebunden werden. Zum Schluss sollen die Änderungen des Prozesses durch eine Mitarbeiterbefragung beurteilt werden.

1.3. Ausgangssituation und tatsächlicher Ablauf

Die Diplomarbeit „Schaffung einer Basis für die kontinuierliche Qualitätsanalyse“ sollte in einem mittelständischen Unternehmen durchgeführt werden. Das Hauptziel der Diplomarbeit war die Einführung des Softwareanalysewerkzeugs im Unternehmen.
Zu Beginn der Diplomarbeit war nur bekannt, dass das Unternehmen für die Entwicklung seiner Software hauptsächlich die Programmiersprachen Java und C einsetzt. Anfangs war es noch geplant, das Softwareanalysewerkzeug ConQAT bei den Java-Entwicklern einzusetzen, da das Werkzeug hauptsächlich für die Analyse von objektorientierten Programmiersprachen entwickelt worden ist. Aus den Gesprächen mit den Leitern der Java- und C-Abteilung hat sich herausgestellt, dass die Java Abteilung bereits ein Werkzeug namens Sonar zur

Softwarequalitätsanalyse einsetzt, im C-Bereich aber noch kein solches Werkzeug vorhanden war. Daher wurde entschieden ConQAT in dieser Abteilung einzusetzen.
Vor dem Einsatz dieses Werkzeugs sollte eine Analyse der Qualitätssicherung des Unternehmens „XYZ“ durchgeführt werden, um zu erfahren, welche Maßnahmen bisher bzgl. Softwarequalität getroffen wurden. Außerdem war es interessant zu erfahren, welche Ergebnisse Sonar liefert und ob sich die Wartbarkeit durch den Einsatz von Sonar erleichtert.
Nach der Analyse wurde ConQAT in einem kleinem Team im C-Bereich eingesetzt. Hierfür wurden mehrere ConQAT-Blöcke erstellt (siehe Kapitel 6), die zur Analyse des Quellcodes nötig waren.
Zum Schluss wurde eine zweite Analyse durchgeführt, um die positiven und die negativen Erkenntnisse nach dem Einsatz von ConQAT zu erfassen.

1.4. Aufbau der Arbeit

Nach der Einleitung in diesem Kapitel werden in Kapitel 2 die Grundlagen der Software-Qualität beschrieben, die notwendig sind, um diese Arbeit besser verstehen und thematisch einordnen zu können. In Kapitel 3 wird die Vorgehensweise bei der mündlichen und schriftlichen Befragung näher erläutert. Außerdem werden die Ergebnisse der Befragung vorgestellt. Das Kapitel 4 handelt von der kontinuierlichen Qualitätsanalyse und ihrer Umsetzung in die Praxis. Im Kapitel 5 wird das „Continuous Quality Assessment Toolkit“ (ConQAT) und dessen Analysemöglichkeiten beschrieben. In Kapitel 6 werden die Umsetzung von ConQAT und die durchgeführten Analysen im Unternehmen näher erläutert. Kapitel 7 beschreibt die zweite Mitarbeiterbefragung, welche dem Zweck diente, die Eignung des Einsatzes von ConQAT im Unternehmen sowie die Akzeptanz der Entwickler gegenüber dem Werkzeug, festzustellen. Außerdem werden Schlussfolgerungen aus den Ergebnissen der beiden Untersuchungen getroffen. Abschließend erfolgt eine Zusammenfassung der Diplomarbeit.

2. Grundlagen

In diesem Kapitel werden kurz die Grundlagen der Softwarequalität beschrieben - von der Definition der Begriffe „Software-Qualität“ und „Softwarewartung“ über die Möglichkeiten zur deren Messung - bis hin zu den Gründen und Zielen für das Software Refactoring.

2.1. Software Qualität

In der Literatur gibt es zahlreiche Definitionen von Qualität, jede betont unterschiedliche Aspekte und betrachtet den Begriff aus einem anderen Blickwinkel.
Einige der bekanntesten Experten definieren „Qualität“ wie folgt:

> *„ Qualität ist die Erfüllung der Anforderungen. Die beste Gewähr für Qualität sind die Vorbeugung und Vermeidung von Fehlern.“* **(P.B Crosby)**
>
> *„Qualität ist gegeben, wenn Zufriedenheit erreicht wird.“* **(W.A.Shewhart)**
>
> *„Qualität ist Funktionstüchtigkeit (Fitness for Use) für den Kunden.“* **(J.M.Juran)**

DIN 55350 - 11 95 definiert den Begriff „Qualität“ folgendermaßen:

> *„ Qualität ist die Beschaffenheit einer Einheit bezüglich ihrer Eignung, festgelegte und abgeleitete Erfordernisse zu erfüllen.“*

In der Softwareentwicklung kann unter „Einheit“ ein Programm, ein Prozess aber auch ein Fachbuch welches die Software beschreibt, verstanden werden. Mit „festgelegten Erfordernissen“ sind explizit benannte Anforderungen gemeint. Bei den „abgeleiteten Erfordernissen“ hingegen führt eine Interpretation öfters zu Missverständnissen. Einerseits sind diese Erfordernisse nicht explizit definiert, deshalb werden sie öfters nicht berücksichtigt. Andererseits variieren sie je Produkt und Projekt enorm, was ihre Umsetzung erschwert. In der Software Entwicklung werden diese Erfordernisse als nicht-funktionale Anforderungen bezeichnet. Sie beziehen sich hauptsächlich auf die innere Qualität der Softwaresysteme (siehe Punkt 2.2) [Sch07a].
Die Verbesserung der Qualität gehört zu den wichtigsten Zielen des Software Engineerings. Aufgrund mangelnder Qualität können Kosten entstehen, welche die reinen Herstellungskosten übersteigen [LL10]. Darüber hinaus kostet es viel Zeit, Kundenanforderungen zu bearbeiten und schwerwiegende Fehler zu beheben, was im schlimmsten Fall zu einem Auftrags- und Imageverlustes führen kann. Leider denken viele Unternehmen erst beim Eintreten eines dieser Fälle an Software-Qualität.
In der Norm **ISO 9126 (ISO 25000)** ist eine Definition für Software-Qualität festgehalten:

> *„ Software-Qualität ist die Gesamtheit von Funktionen und Merkmalen eines Software-Produkts, dass die Fähigkeit besitzt, angegebene oder implizierte Bedürfnisse zu befriedigen.“*

Die Qualität ist also abhängig von der Gesamtheit aller Merkmale und Funktionen eines Produktes. Diese Merkmale und Funktionen wiederum, werden durch die Eigenschaften dieses Produktes bestimmt.Die Merkmale des Softwareproduktes können entsprechend vielfältig sein. Daher wird im folgenden Punkt 2.2, der Begriff „Qualitätsmerkmal“, sowie dessen Unterteilung näher erläutert.
Es gibt kaum ein anderes Produkt bei dem sich die Qualität so schwer beurteilen und sichern lässt wie bei Software [Dei09].
Aufgrund der Vielschichtigkeit und der Komplexität des Begriffes „Software-Qualität“ sind solche Definitionen, wie die aus dem **ISO 9126-Standard**, in der Praxis sehr schwer umzusetzen. Daher versucht man in praktischen Anwendungen, die Merkmale und die Submerkmale der Software-Qualität, durch Qualitätsmodelle zu konkretisieren und zu operationalisieren [Wal11].
Das Software Engineering befasst sich sowohl mit der Produktqualität als auch mit der Prozessqualität (Projektqualität) [LL10].
Die Prozessqualität ist eng verbunden mit der Termin- und Kosteneinhaltung, sowie mit der Minimierung des Aufwandes während der Herstellung eines Softwareproduktes. Eine gute Prozessqualität schafft im Allgemein bessere Voraussetzungen, um ein hochqualitatives Softwareprodukt zu entwickeln. Diese Diplomarbeit beschäftigt sich allerdings mit der Produktqualität, deswegen soll hier auf eine detaillierte Beschreibung der Prozessqualität nicht näher eingegangen werden.
Bei der Produktqualität handelt es sich sowohl um die Erfüllung der expliziten und implizierten Nutzeranforderung, als auch um die Verbesserung der inneren Eigenschaften eines Softwareproduktes, wie Lesbarkeit, Testbarkeit und Weiterentwicklungsmöglichkeit.
Ludewig und Lichter teilen die Produktqualität weiter in Wartungs- und Gebrauchsqualität ein. Auf diese Weise betrachten sie einerseits die Produktqualität aus Sicht eines Endverbrauchers, der das Softwareprodukt benutzen soll, andererseits aber auch aus der Sicht einer Person, welche diese Software ändert, weiterentwickelt und korrigiert.

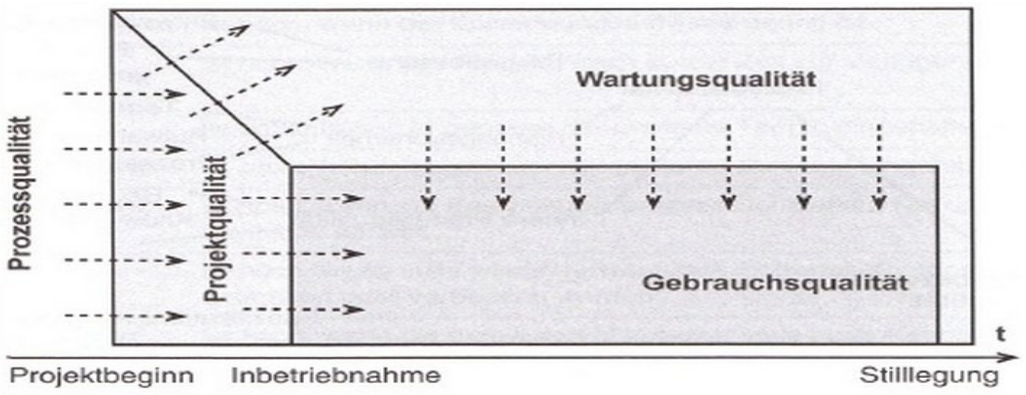


Abbildung 2.1.: Bedeutung verschiedener Qualitätsaspekte der Zeit (Ludewig, 2010 S.67)

Die Abbildung 2.1 zeigt die Bedeutung und die Einflüsse der oben beschriebenen Qualitätsaspekte auf die Gebrauchsqualität im Verlauf der Entwicklung.
Der Benutzer legt schließlich nur Wert auf die Gebrauchsqualität, trotzdem kann ihm die Wartbarkeit nicht gleich gültig sein. Es könnte sein, dass er selber die Wartung übernimmt. In diesem Fall braucht er einen leicht lesbaren und erweiterbaren Code. Sogar wenn der Kunde nichts mit der Wartung zu tun haben sollte, ist die Wartbarkeit eine sehr wichtige Voraussetzung, um schneller auf Abnehmeranforderungen einzugehen, neuen Versionen der Software wirklich zu verbessern und Fehler zu reduziert. Viele Wissenschaftler wie McCall [SSB10] Boehm, Lipow und Ludewig versuchen die Software-Qualiät als Qualitätsbaum darzustellen. Die Abbildung 2.2 zeigt wie Ludewig den Qualitätsbegriff in Anlehnung an Boehm, Brown und Lipow (1976) in einem Qualitätsbaum unterteilt [LL10].

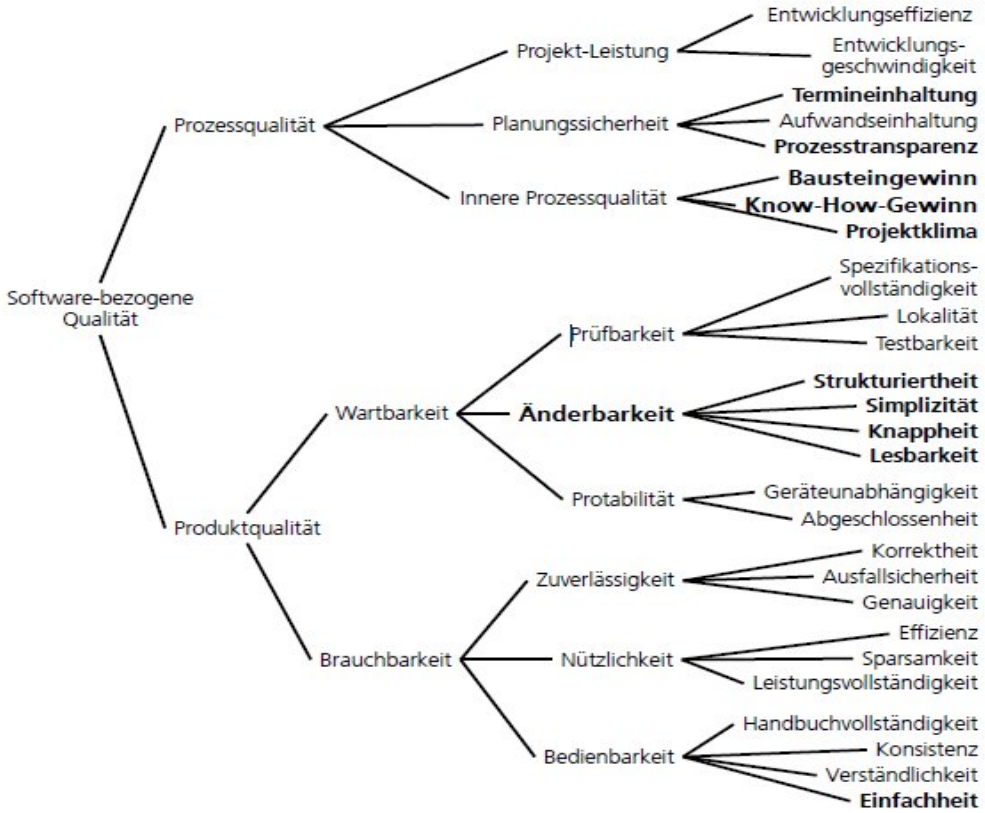


Abbildung 2.2.: Qualitätsbaum nach Ludewig (Ludewig, et al., 2010 S. 68)

Da sich diese Diplomarbeit um die Produktqualität und speziell um die Innere Qualität von Softwareprodukten dreht, werden die nächsten Kapitel die untere Hälfte des Baumes beschreiben. Dabei wird der Wartbarkeit und ihrer Verbesserung besondere Aufmerksamkeit geschenkt.

2.2. Innere und Äußere Software-Qualität

Damit die Software-Produktqualität besser betrachtet und beurteilt werden kann, wird sie in die **ISO Norm 9126** in interne und externe Qualität unterteilt [Wal11]. Softwaresysteme haben also sowohl externe als auch interne Qualitätsmerkmale. Was ist aber unter einem Qualitätsmerkmal zu verstehen?
Kurt Schneider definiert den Begriff Qualitätsmerkmal als [Sch07a]

> *„einzelne Eigenschaft einer Einheit, anhand derer ihre Qualität beschrieben und beurteilt wird."*

Die Abbildung 2.3 stellt die Software-Qualitätsmerkmale nach **ISO 9126** dar.
Die externen Qualitätsmerkmale sind besonders wichtig für den Endbenutzer eines Software-Produktes, da er sie direkt wahrnimmt. Sie äußern sich in Qualitätseigenschaften wie Benutzbarkeit, Korrektheit, Zuverlässigkeit, Funktionalität und Effizienz.

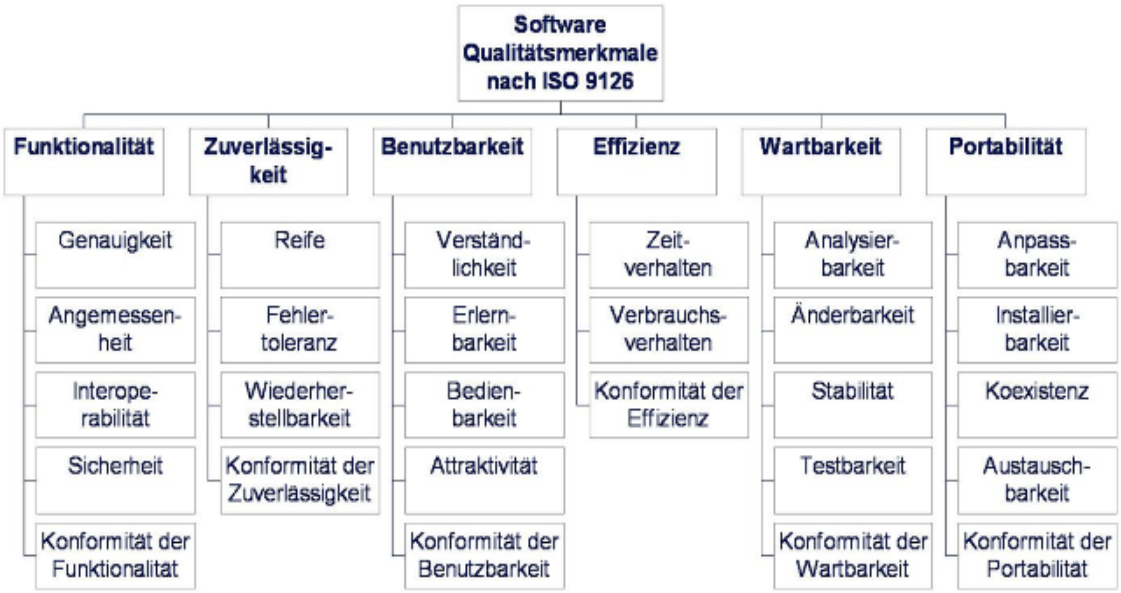


Abbildung 2.3.: Software Qualitätsmerkmale nach ISO/IEC 9126 (enzyklopaedie-der-wirtschaftsinformatik.de)

Im folgenden werden einigen der wichtigsten externen Qualitätsmerkmale kurz beschrieben [Lig02]:

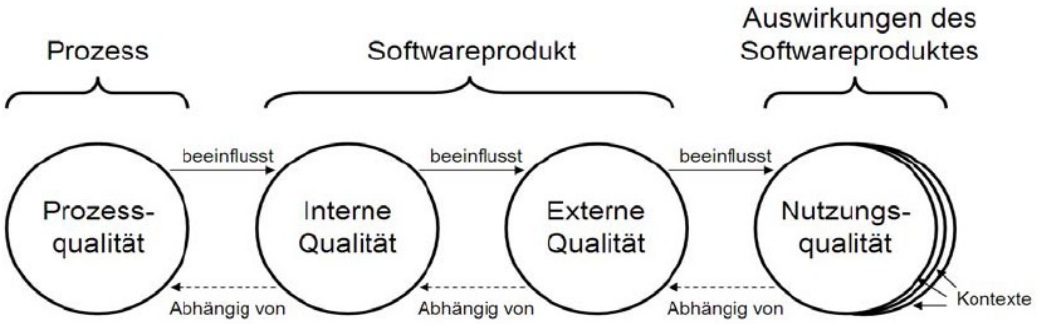
- **Benutzbarkeit:** Die Einfachheit, mit der ein Anwender die Software erlernen und bedienen kann.
- **Korrektheit:** Die Fehlerfreiheit eines Softwaresystems, d.h. ihre Konsistenz zur Spezifikation.

- **Zuverlässigkeit:** Die Ausfallfreiheit eines Systems unter vorgegebenen Arbeitsbedingungen.
- **Funktionalität:** Der Grad der Erfüllung erforderlichen Funktionen.
- **Effizienz:** Der Verbrauch von Softwareressourcen, wie z. B. die Antwortzeiten in Sekunden oder der maximale Speicherverbrauch.

Der Endkunde ist ausschließlich an diesen Qualitätsaspekten des Softwaresystems interessiert. Er will ein funktionierendes und leicht bedienbares Softwareprodukt. Die Entwickler hingegen achten auf die internen Merkmale und Charakteristiken des Softwaresystems. Eine von den wichtigsten Fragen hierbei ist: „Wie leicht ist der Code zu warten?“ Damit ist der Aufwand zum Ändern, Verstehen und Erweitern des Codes gemeint. Die ISO Norm 9126 (siehe Bild 2.3) gibt die Wartbarkeit und die Portabilität als innere Qualitätsmerkmale vor.

- **Wartbarkeit:** Der Aufwand, mit dem vorgegebene Änderungen durchzuführen sind.
- **Portabilität:** Die Anstrengung, mit der ein System von einer Umgebung in eine andere zu übertragen ist.

Wenn der gesamte Lebenszyklus eines Softwaresystems betrachtet wird, sieht man, dass die interne Qualität die externen Qualitätseigenschaften des Softwareproduktes beeinflusst. Eine schlechte interne Qualität hat zu Folge, dass darunter die Gebrauchsqualität des Softwaresystems leidet. Viele Wissenschaftler sind der Meinung, dass die innere Qualität des Softwareproduktes genau so wichtig, wie die äußere Qualität ist [Bal08].
Die Abbildung 2.4 zeigt die Beziehung zwischen der internen Qualität und der externen Qualität nach **ISO/IEC 9126-1**.



Quelle: International Standard (2001): ISO/IEC 9126-1: Software Engineering-Product Quality-Part 1

Abbildung 2.4.: Beziehungen zwischen interne Qualität und externe Qualität

Welchen Nutzen hat ein System mit hoher externer Qualität, wenn die kleinsten Änderungen und Erweiterungen sehr zeitaufwändig und mit hohen Kosten verbunden sind?
Wenn die innere Qualität nicht stimmt, d.h das Programm sehr schwer lesbar, erweiterbar oder allgemein wartbar ist, erschwert sich die Fehlersuche und die Ausbaufähigkeit der

Software. Dadurch verschlechtert sich aber die Korrektheit und die Zuverlässigkeit. Es ist sehr schwer auf Kundenwünsche zu reagieren, wenn die Software unflexibel ist. Als Folge davon sinkt die Benutzerfreundlichkeit.
Eine hohe interne Qualität ist also Voraussetzung für hohe externe Qualität [McC05]. Die relevanten Qualitätsattribute für diese Arbeit sind die inneren Qualitätsmerkmale, daher werden sie als Nächstes genauer betrachtet. Im Mittelpunkt stehen dabei die Wartbarkeit und ihre Submerkmale: Analysierbarkeit, Änderbarkeit, Stabilität und Testbarkeit.

2.3. Wartbarkeit

Die Abbildung 2.1 zeigt, dass die Gebrauchsqualität von der Wartungsqualität beeinflusst wird. Wirtschaftlich betrachtet, ist die Wartbarkeit, besonders von Softwaresystemen mit langem Lebenszyklus ein äußerst wichtiges Kriterium, das leider öfters ignoriert wird [BSB08].
Damit die Wartbarkeit besser verstanden werden kann, muss zuerst der Begriff „Software-Wartung“ geklärt werden.
Die Norm **IEEE 610** definiert die SoftwareWartung wie folgt:

> „*Software-Wartung ist die Veränderung eines Softwareproduktes nach dessen Auslieferung, um Fehler zu beheben, Performance oder andere Attribute zu verbessern oder Anpassungen an die veränderte Umgebung vorzunehmen.*“

Diese Definition der Softwarewartung ist aber nicht ganz widerspruchsfrei. Einerseits gibt es Probleme, die noch während der Entwicklung behoben werden müssen. Der einzige Unterschied zu der obigen Definition ist, dass das Softwaresystem noch nicht bei den Anwendern eingesetzt wird. Andererseits wird die Software auch nach ihrer Auslieferung planmäßig erweitert und verbessert. Der letzte Fall wird von Ludewig und Opferkuch als Entwicklung und nicht als Wartung, betrachtet und wahrgenommen. Daher definieren sie die Wartung als [LL10]:

> „[...] *jede Arbeit an einem bestehenden Software-System, die nicht von Beginn der Entwicklung an geplant war oder hätte geplant werden können und die unmittelbare Auswirkung auf den Benutzer der Software hat.*“

Weiterhin lässt sich die Software-Wartung in 4 Kategorien klassifizieren [BSB08]:

- **Korrektive Wartung:** Dies ist die Behebung von Software-Fehlern, die nach der Auslieferung des Softwaresystems vorgetreten und von den Anwendern erkannt worden sind.
- **Präventive Wartung:** Hiermit ist die Behebung von Software-Fehler gemeint, die entdeckt worden sind, bevor sie als effektive Fehler im Feld ihrer Wirkung aufgetreten sind.
- **Adaptive Wartung:** Dies ist die Anpassung der Software an eine sich ändernde Systemumgebung, so dass sie neue oder geänderte Anforderungen erfüllt.

- **Perfektionierende Wartung:** Sie ist allgemein mit der Verbesserung des Softwaresystems verbunden, wie zum Beispiel Performanceverbesserung oder die Verbesserung der Benutzbarkeit.

Es gibt eine Menge von Studien, die den Aufwand der Software-Wartung analysieren und einen Endruck von ihrem Gewicht im gesamten Software-Lebenszyklus verschaffen. Shaw und Gannon (1979) geben an, dass 67% des Gesamtaufwandes mit der Wartung verbunden sind. Abran und Nguyenkim (1991) ermitteln 55% Wartungsaufwand [LL10]. Demzufolge kann die Wartbarkeit als eines der entscheidenden Qualitätsmerkmale eines Softwaresystems angesehen werden.
Die Norm **IEEE 610** definiert die Wartbarkeit wie folgt:

> *„Wartbarkeit ist die Einfachheit mit der ein Softwaresystem oder eine Komponente modifiziert werden kann, um Fehler zu beheben, Performance oder andere Attribute zu verbessern oder Anpassungen an die veränderte Umgebung vorzunehmen.“*

Die Wartbarkeit bestimmt mit welchem Aufwand, und mit welcher Qualität die Änderungen in einem Softwaresystem verbunden sind. Ein System soll in diesem Sinne bestimmte Eigenschaften aufweisen, damit es leicht modifiziert und verbessert werden kann [BSB08]. Die **ISO/IEC Norm 9126** unterteilt die Wartbarkeit weiter in Submerkmale, die eine verfeinerte Betrachtung dieses Begriffes ermöglichen und definiert sie wie folgt [91201]:

- **Analysierbarkeit:**

  > *„Aufwand, der benötigt wird, um Ursachen von Versagen oder Mängel zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen.“*

- **Änderbarkeit/Modifizierbarkeit:**

  > *„Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassungen an Umgebungsänderungen.“*

- **Stabilität:**

  > *„Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen.“*

- **Testbarkeit:**

  > „[...] *der zur Prüfung der geänderten Software benötigte Aufwand.“*

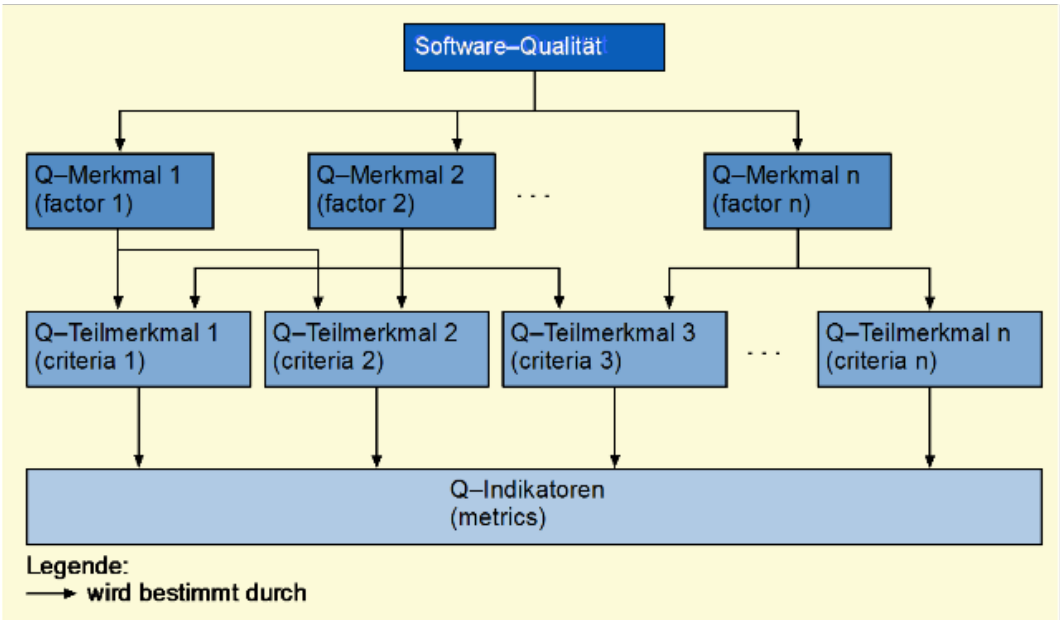
In der Fachliteratur wird öfters auch zwischen der Weiterentwicklungsfähigkeit und der Wartbarkeit unterschieden. Da die Wiederverwendbarkeit einer der wichtigsten Voraussetzungen für kürzere Entwicklungszeiten und einen geringeren Weiterentwicklungsaufwand ist, betrachtet man sie auch als ein wichtiges inneres Qualitätsmerkmal [Brc11] [RB09].
Die Wiederverwendbarkeit kann nach [Sam97] definiert werden als:

> „[...] *der Aufwand, der benötigt wird, um ein Software-System auf Grundlagen von einer existierenden Implementierung weiterzuentwickeln.“*

Dieser Aufteilung und Standardisierung der inneren Qualitätsaspekte kann auf die Code-Ebene eines Softwaresystems abgebildet werden. Somit lassen sich die Begriffe der Wartbarkeit und der Weiterentwicklungsfähigkeit abmessen und quantifizieren.

2.4. Qualitätsmodelle

Das Ergebnis dieser Verfeinerung und der Aufteilung der Software-Qualitätsziele in Qualitätsmerkmale und Submerkmale wird als Qualitätsmodell bezeichnet. Die Qualitätsziele können je nach Projekt und Produkt unterschiedlich definiert und interpretiert werden. Beispiele für Qualitätsziele sind die „Wartbarkeit“ oder die „Weiterentwicklungsfähigkeit“ von Softwaresystemen. Die aus diesen oder anderen Qualitätszielen resultierenden Qualitätsmerkmale können in weitere Teilmerkmale aufgefächert werden, um das Qualitätsmodell weiter zu charakterisieren [Gol11]. In diesem Sinne können die Qualitätsmodelle als eine abstrakte Beschreibung der Qualität von Softwaresystemen angesehen werden. Dadurch versucht man die Software-Qualität in einer strukturierten Art und Weise zu vereinfachen, um sie leichter messbar und bewertbar zu machen [WDFJ08].
Die bekanntesten Vertreter solcher Qualitätsmodelle sind von Böhm et al. [BBK+78] und MCCall & Walter vorgeschlagen worden [MW77].
Die Abbildung 2.3 stellt das Software-Produktqualitätsmodell der Norm ISO/ IEC 9126 dar. Das Qualitätsmodell unterscheidet sechs Qualitätsmerkmale, die weiter oben bereits beschrieben worden sind. Auf Basis dieses Qualitätsmodells können weitere projektspezifische Modelle erstellt werden. Einer der bekanntesten Ansätze, um Qualitätsmodelle zu erstellen, ist der von Cavano und McCall vorgeschlagene FCM-Qualitätsmodellansatz (Factor, Critera, Metrics) [CM78]. Cavano und McCall betrachten die Softwarequalität als eine Menge von Qualitätsfaktoren (Factors), die sich wiederum aus Qualitätskriterien (Criteria) zusammensetzen. Diese zusammengefassten Kriterien können dann durch eine oder mehrere Metriken (Metrics) gemessen werden.



Quelle: enzyklopaedie-der-wirtschaftsinformatik.de

Abbildung 2.5.: Factor-Critera-Metrics-Qualitätsmodell

2.5. Messen von Softwarequalität

Die Erstellung eines Qualitätsmodells hat unter anderem den Sinn, die Ermittlung von präzisen Qualitätsmetriken zu unterstützen. Mit den ermittelten Metriken kann dann eine objektive, quantitative und qualitative Aussage über bestimmte Eigenschaften eines Softwaresystems gemacht werden. In dieser Diplomarbeit liegt der Fokus auf den inneren Qualitätsmerkmalen eines Softwaresystems. Daher soll der Begriff „Softwarequalitätsmessung“ als die Suche nach Qualitätsdefiziten, die potentielle Fehlerquellen darstellen können, verstanden und interpretiert werden. Auf dem Begriff des Qualitätsdefizites wird in Kapitel 2.8 eingegangen.

2.5.1. Sinn und Zweck der Software Messung

Ein wesentlicher Zweck der Software Messung basiert auf dem Wunsch, qualitative und quantitative Aussagen über das Produkt Software machen zu können. Man unterscheidet dabei zwischen der Vermessung der Softwareeigenschaften und die Kontrolle der Entwicklungsprozesse [Lig02]. Mit den Eigenschaften der Software sind die inneren und die äußeren Qualitätsmerkmale, wie die Wartbarkeit oder die Benutzbarkeit gemeint. Unter Entwicklungsprozessen sollen die definierten Prozesse zur Erstellung eines Softwareproduktes verstanden werden. Darüber hinaus ist das Verstehen der Software ein wichtiges Ziel der Software Messung. Vor allem will man sich einen Überblick über die verschiedenen Bauelemente einer Software verschaffen und die Beziehungskomplexität zwischen diesen Bauelemente ermitteln. Mit einer Software Messung bzw. Analyse versucht man durch Zahlen und Symbole die Software zu beurteilen, um ihre Qualität und Komplexität kontrollieren und steuern zu können. Die Messergebnisse, die daraus resultieren, können zum Vergleich unterschiedlicher Softwareprodukte oder unterschiedlicher Module innerhalb eines Software-Produktes verwendet werden.
Ein weiteres wichtiges Ziel ist die Kontrolle der Einhaltung von definierten Unternehmensstandards, wie Code- und Dokumentationsregeln. Wird das Ganze projektübergreifend betrachtet, können durch eine Messung auch Vorhersagen über den Aufwand und die Kosten eines Projektes getroffen werden.
Besonders wichtig ist die Messung aber, wenn es um Softwaresysteme geht, die über mehrere Jahre entwickelt und eingesetzt werden. Solche Systeme bestehen gewöhnlich aus mehreren Millionen Zeilen Code und werden als „Langlebige Softwaresysteme“ bezeichnet [Dei09]. Wirtschaftlich betrachtet, spielt die Wartbarkeit und die Weiterentwicklungsfähigkeit solcher Systeme eine sehr wichtige Rolle für die Unternehmen. Deswegen versucht man bei diesen Systemen, frühzeitig Schwachstellen im Code zu identifiziert, damit der Software-Qualitätsverfall vermieden werden kann.

2.5.2. Metriken

Das Wort Metrik kommt aus dem Griechischen und bedeutet Kunst des Messens [Lig02]. In der Mathematik wird der Begriff „Metrik“ als Distanzfunktion verwendet. Im Software

Engineering hingegen wird er als „Maß“ verstanden, mit dessen Hilfe man quantifizierte Analysen über Entwicklungsprozesse oder Produkte macht [LL10].
Die Norm IEEE 1061 definiert Softwaremetrik wie folgt:

> *„Eine Softwarequalitätsmetrik ist eine Funktion, die eine Softwareeinheit in einem Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad eines Qualitätsziels für die Softwareeinheit.“*

Mit dem Einsatz von Softwaremetriken können Aussagen über die Komplexität eines Softwaresystems getroffen werden. Diese Aussagen können später zum Vergleich mit den Werten anderen Programme verwendet werden. Zudem können die Metriken benutzt werden, um Erfahrungen aus früheren Projekte zu quantifizieren, Kosten-, Termin- und Qualitätsprognosen zu treffen oder Entscheidungen, wie die Frage nach der Notwendigkeit von Refactoring oder Reengineering Maßnahmen, zu unterstützen. Allgemein kann man sagen, dass Softwaremetriken eine objektive Einschätzung der Qualität der Software erlauben. Man erkennt triviale oder komplexe Bereiche und schafft eine Planungsgrundlage für die Verbesserung der Prozesse und der Produkte.
In der Praxis werden öfters außer Zahlenwerte auch Symbole und Zeichen in die Messwertdarstellungen einbezogen, um mehr von den wesentlichen Aspekten eines Softwaresystems erfassen zu können [Sch07a].
Im Ganzen können die Software Metriken als ein Bestandteil der Qualitätskontrolle von Softwareprodukten und -projekten angesehen werden.
Die Metriken müssen aber auch gewisse Kriterien erfüllen. Sie müssen differenziert, vergleichbar, reproduzierbar, verfügbar, relevant, rentabel und plausibel sein. Vergleichbar heißt in diesem Fall, dass sich die Bewertungen vergleichen lassen. Mit differenzierbar ist gemeint, dass die Metriken unterschiedliche Werte liefern sollen, z. B. wenn man zwei unterschiedlich schwer lesbare Programme miteinander vergleicht. Plausibel heißt, dass eine Metrik ein schlechtes Programm nicht als gut bewerten darf [LL10].
Software Metriken sind sehr hilfreich, aber sie haben auch gewisse Schwachstellen. Erstens gibt es sehr viele und sehr unterschiedliche Metriken. Zweitens es ist abhängig von dem Projekt und dem Produkt, wie die Metriken eingesetzt und interpretiert werden.Drittens ist es umstritten, ob man eine Softwareeinheit nur auf eine Zahl abbilden sollte, um Aussagen über ihre Qualität zu machen. Auch wenn diese Zahl im Bereich, der von den Autoren definierten Grenzwerten liegt, ist es trotzdem meistens unklar, wie sie interpretiert werden soll und welche Maßnahmen sich daraus ableiten lassen.Außerdem kosten die Metriken Geld und Zeit, daher müssen die sorgfältig ermittelt und mit Vorsicht eingesetzt werden [Gol11].

2.5.3. Klassische Softwaremetriken

Von allen Softwareartefakten eignet sich der Programmcode am besten für eine Messung [SSB10]. Da es sich in dieser Diplomarbeit ausschließlich um die Produktqualität eines Softwaresystems dreht, werden hauptsächlich Codemetriken betrachtet. Diese Metriken sind hilfreich, um die Qualität (Komplexität und die Wartbarkeit) eines Quellcodes beurteilen zu können.

2.5.3.1. Lines of Code

Die bekannteste und die am weitesten verbreitete Softwaremetrik ist die Lines of Code (LOC). Diese Metrik dient als Angabe der Größe einer Funktion, einer Datei oder eines Softwaresystems. Gewöhnlich werden die Codezeilen eines Softwareprogramms gezählt, um traditionelle Aussagen, wie die Produktivität pro Monat oder pro Tag treffen zu können [Thaoo]. Man nimmt außerdem an, dass die Übersichtlichkeit, mit steigender Länge des Codes, schlechter wird. In der Praxis ist aber die Interpretation dieser Metrik nicht ganz so einfach. Zwei oder mehrere unterschiedliche Codezeilen können sich in ihrer Einfachheit ziemlich stark unterscheiden. Zudem tauchen auch folgende Fragen auf: Sollen nur Zeilen mit ausführbaren Code (Statements) betrachtet werden? Werden Datendefinitionen, Leerzeilen oder Kommentare mitgezählt? Dabei spielen unterschiedliche Programmiersprachen auch eine sehr große Rolle, da die erforderliche Quellcodezahl von der jeweiligen Sprache abhängig ist. Hierbei muss man beachten, dass sich schlechte und bedeutungslose Kommentare als nutzlos und kontraproduktiv erweisen können. Bei der Betrachtung dieses einfachen Maßes sieht man schon, wie schwer es ist überhaupt Metriken zu definieren und zu interpretieren. Offenbar kann LOC nur beschränkt eingesetzt werden, um qualitative Aussagen über den Quellcode zu treffen. Sinnvoller ist es, wenn man diese Metrik in Kombination mit anderen Metriken anwendet, damit gewisse Softwareeigenschaften präziser beurteilt werden können. In den späteren Kapiteln dieser Arbeit, wird diese Metrik in einer Kombination mit anderen Metriken betrachtet und beschrieben.

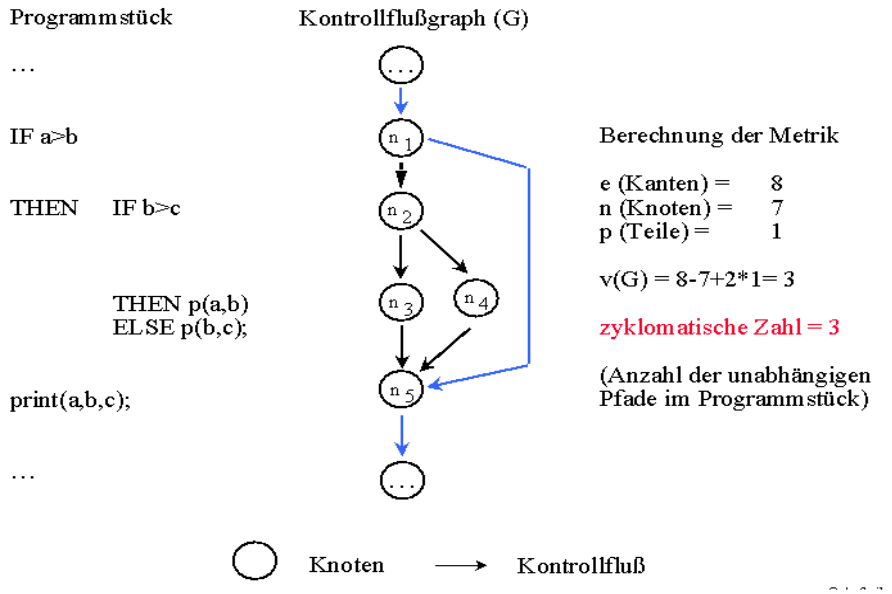
2.5.3.2. Zyklomatische Komplexität

Die zyklomatische Komplexität, auch als McCabe-Zahl bekannt, ist eine der am weitesten verbreiteten Metriken in Analyse- und Testwerkzeugen. Diese Metrik wurde 1976 eingeführt und ist unabhängig von der Programmiersprache. Die Grundidee dabei ist komplexe Module zu ermitteln, welche die Testbarkeit und die Wartbarkeit eines bestimmten Quellcodestückes schwer beeinträchtigen [Gol11].
McCabe betrachtet den Programmcode als einen gerichteten Graphen mit Kanten und Knoten. Er nutzt den Programmablaufgraph als ein Hilfsmittel, um die Komplexität eines Programms zu ermitteln, in dem er die Verzweigungen in dem Programmablauf berücksichtigt. Die zyklomatische Zahl **C(G)** eines Programmablaufgraphen bzw. Kontrollflussgraphen **G** wird definiert durch:

$$C(G) = e - n + 2c,$$

Dabei ist **e** die Anzahl der Kanten von **G**, **n** die Anzahl der Knoten von **G** und **c** die Anzahl verwandter Teilgraphen. Die verwandten Teilgraphen entsprechen den aufgerufenen Funktionen oder Prozeduren eines Programms [SSB10].

Es gibt auch eine andere Möglichkeit für die Ermittlung der zyklomatischen Zahl. Wird eine strukturierte Sprache benutzt, kann diese Zahl direkt aus dem Programmcode ablesen werden. Hierfür müssen die relevanten Anweisungen im Programmcode gezählt werden [Sch07a]:



Quelle: www.computer.freepage.de

Abbildung 2.6.: Beispiel für Berechnung der McCabe-Zahl

Zyklomatische Komplexität [auf Codebasis] =

Anzahl der Verzweigungen		(if)
+	Anzahl der Schleifen	(for, while, repeat usw.)
+	je: (Anzahl der Zweige - 1)	(case-, switch- Verzweigungen)
+	1	

Quelle: Definition für zyklomatische Komplexität (Schneider, et al., 2007 S. 64)

Am Schluss wird immer der Wert 1 addiert, dadurch bekommt sogar ein sehr einfaches Programm die zyklomatische Zahl 1 und nicht null. Diese Metrik bringt aber auch Schwächen mit sich. Wenn man die Möglichkeit mit dem Kontrollflussgraphen nimmt, sieht man dass hier nur Kanten und Knoten gezählt werden. Die Kommentare und die Wahl der Bezeichner werden beispielsweise nicht berücksichtigt. Offenbar berechnet diese Möglichkeit den „Verzweigungsgrad“ des Programmcodes. McCabe vertritt folgende Meinung: je mehr Ablaufmöglichkeiten es in einem gerichteten Graph gibt, desto schwieriger ist es dieses Programm zu testen. Außerdem soll die höhere Komplexität zu mehr Fehlern führen, da der Code schwer zu verstehen ist. Mit der Zahl, die durch diese Metrik ermittelt wird, kann wenig angefangen werden. McCabe vergleicht sie mit den Maßen anderer Programme und kommt zu der Schlussfolgerung, dass „eine zyklomatische Komplexität bis **10 niedrig** sei, bis **20** dann **mittel**, darüber **hoch** und über **50 undurchschaubar**“ [Sch07a].

2.5.3.3. Halstead Software Science

Die Halstead-Metrik ist ein anderes analytisches Verfahren, mit dem man die Komplexität und die Größe des Codes beurteilen kann. Der Ansatz wurde 1977 entwickelt und wird von seinem Erfinder Maurice Howard Halstead mit dem Begriff „Software Science“ bezeichnet [SSB10]. Als Grundlage für die Messung mit der Metrik dienen die Anzahl der unterschiedlichen Operanden und Operatoren und die Gesamtzahl der Operatoren und Operanden [Lig02].
Unter Operatoren sind Schlüsselwörter und Symbole zu verstehen (**+, (, if, while, for**). Zu den Operanden hingegen gehören Konstanten, Variablen, Zahlen usw. Die Halstead-Metrik nutzt hauptsächlich vier Kenngrößen:

- **n** = Anzahl verschiedener Operatoren
- **m** = Anzahl verschiedener Operanden
- **N** = Gesamtzahl aller Operatoren
- **M** = Gesamtzahl aller Operanden

Mit **L = M + N** bezeichnet Halstead die abstrakte Länge eines Programms, **l = m+n** entspricht die Vokabulargröße. Mit Hilfe dieser Größen kann dann der Schwierigkeitsgrad **D**, um ein Programm zu lesen, berechnet werden. Die Formel für der Schwierigkeitsgrad ist:

$$D = \frac{n}{2} * \frac{M}{m}$$

Weiterhin können das Programmvolumen:

$$V = L * \log_2(l),$$

und die Anzahl der ausgelieferten Bugs **B=V/3000** berechnet werden [Sch07a].
B schätzt die Anzahl der Implementierungsfehler und ist eine wichtige Metrik für den dynamischen Test von Software. Beim Testen sollte man so viele Fehler finden, wie die Metrik B angibt [Gol11]. Diese Metrik weist auch Schwächen auf, da nur der lineare Aufbau des Quellcodes in Betracht gezogen wird, nicht aber die Verzweigungen oder die Verschachtelungen im Programm. Außerdem ist die Vergleichbarkeit von Systemen, die in unterschiedlichen Programmiersprachen geschrieben worden sind, nicht gegeben.
Wie man sieht, bringt jede Metrik auch gewisse Mängel mit sich. Deswegen ist es wichtig solche Metriken auszuwählen und zu definieren, die am besten zu den jeweiligen Projekt oder Softwaresystem passen. Man muss also das messen, was wichtig ist, um seine Verbesserungsziele zu erreichen.
Außer den hier betrachteten klassischen Metriken gibt es auch objektorientierte Metriken, mit welchen man die Vererbungstiefe oder die Kohäsion untersuchen kann. Da aber in dieser Arbeit die Programmiersprache **C** analysiert wird, werden diese Metriken nicht in Betracht gezogen.

2.6. Goal Question Metric

Einer der bekanntesten Vorgehensweise, um Software Qualitätsmetriken zielgerichtet zu ermitteln ist, der von Basili [BCR94] vorgeschlagenen Goal Question Metric-Ansatz (GQM). Es handelt sich dabei um einen Top-down-Ansatz, bei dem ausgehend von den eigenen Zielen die benötigten Metriken abgeleitet werden.

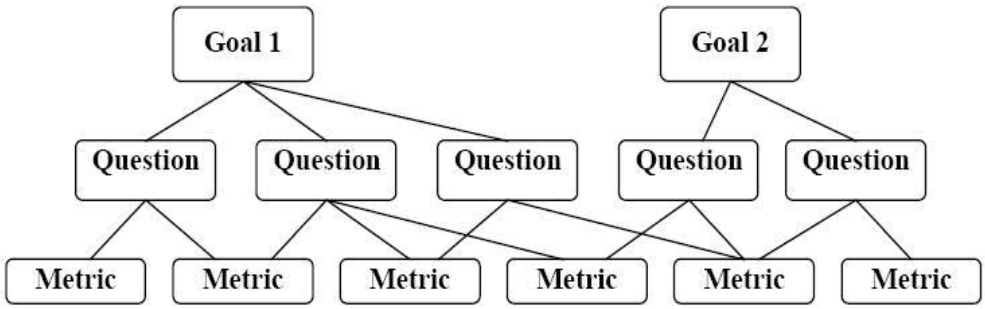


Abbildung 2.7.: Goal Question Metric

Häufig versucht man Metriken, die man zufällig findet oder kennt, anzuwenden. Die Ergebnisse, die dabei ermittelt werden, sind öfters sehr schwer zu interpretieren. Mit GQM hingegen überlegt man sich im Vorfeld, wie man die möglichen Ergebnisse interpretieren und auswerten kann [Sch07a]. Bei diesem Verfahren versucht man schrittweise die eigenen Ziele über Fragen bis hin zu Metriken zu verfeinern. Dabei sind drei Fragen zu beantworten [Lig02]:

1. Welche Ziele sollen erreicht werden (**Goal**)?
2. Welche Fragen sind zu beantworten, um die vordefinierten Ziele zu erreichen (**Question**)?
3. Durch welche Maße können die benötigten Eigenschaften erfasst werden (**Metric**)?

Die Abbildung 2.7 stellt den Ansatz von Basili [BCR94] dar.
Wie hier zu sehen ist, werden zuerst die Ziele gesetzt. Als nächstes werden die Fragen zu diesen Zielen gestellt. Durch die Fragen kann man erkennen, wann die Ziele erreicht sind oder wie sie zu erreichen sind. In dem letzten Schritt werden Metriken ausgewählt, die uns helfen sollen, Antworten auf die gestellten Fragen zu finden.
Die Beliebtheit dieses Ansatzes kommt daher, dass von vorne herein viele irrelevante Metriken ausgeschlossen werden. Da man sich an gewissen vordefinierten Ziele festhält, wird sichergestellt, dass die erhobenen Metriken tatsächlich benötigt werden [LL10].

2.7. Statische Codeanalyse

Eine der Möglichkeiten, um die innere Qualität der Software zu messen, ist die statische Codeanalyse. Im Gegensatz zu der dynamischen Code-Analyse verzichtet man hier auf die

Ausführung des Programms . Die statische Analyse ist ein Mittel, um Schwachstellen im Code zu ermitteln und die Einhaltung gewisser Standards zu überprüfen. Grundsätzlich kann die statische Analyse auch manuell ausgeführt werden. Die manuelle Überprüfung weist aber gewisse Nachteile auf. Einerseits ist so eine Analyse extrem aufwändig, wenn beispielsweise ein System mit mehreren Millionen Zeilen Code überprüft werden soll. Andererseits ist die Qualität der aggregierten Ergebnisse schlechter als von automatisch erzeugten Ergebnissen. Deswegen wird empfohlen, solch eine Analyse, von Werkzeugen unterstützt oder gar nicht einzusetzen [Lig02]. Die automatisierte Überprüfung soll den Prüfer außerdem so wenig Zeit wie Möglich kosten, damit er sich auf das Wesentliche konzentrieren kann.
In dieser Diplomarbeit wird das Continuous Quality Assessment Toolkit (ConQAT) eingesetzt. Das Werkzeug wird in den späteren Kapitel detaillierter erklärt und beschreiben.

2.8. Qualitätsdefizite in den Softwaresystemen

Martin Fowler und Kent Beck beschreiben in ihrem Buch „Refactoring“ eine Reihe von Qualitätsdefiziten, wie z.B. Code-Duplikate, lange Parameterlisten oder Toter Code in der Programmierung. Diese Schwachstellen im Programmcode werden von den beiden Autoren des Buches als „Code Smells“ bezeichnet. Dabei handelt es sich nicht um Programmfehler, sondern um Stellen im Code, die die Wartbarkeit enorm einschränken. Beide Autoren nutzen den Begriff „Code Smell“, um die Qualitätsdefizite im Sourcecode aufzuzeigen, die durch Refactoring-Maßnahmen behoben werden sollten. Martin Fowler definiert den Begriff „Refactoring“ wie folgt [FBBO02]:

> *„ Refactoring ist ein Prozess, der die interne Struktur eines Softwaresystems verbessert und dabei das externe Verhalten der Implementierung unverändert lässt.“*

Die Code Smells, die von Fowler und Beck beschrieben worden sind, beziehen sich eher auf die objektorientierte Programmierung [FBBO02]. Einige von den “Code Smells“ sind aber auch für andere Programmier-Paradigmen (z. B. für nicht objektorientierte Sprachen) relevant. Als nächstes werden verschiedene „Code Smells“, die sich auch für nicht objektorientierte Sprachen eignen, beschrieben.

2.8.1. Code Duplikate

Als „Code Duplikate“ werden identische Codefragmente bezeichnet, die sich an unterschiedlichen Stellen unnötig wiederholen. Darunter leidet am meistens die Wartbarkeit der Software Produkte. Durch die Umwandlung der duplizierten Stelle in einer Funktion könnte dieses Qualitätsdefizit behoben werden. In Punkt 5.1.2.2 sind die Gründe und die Auswirkungen der Code Duplikate auf die Wartbarkeit eines Softwaresystems ausführlich beschrieben.

2.8.2. Lange Funktion

Als eine „lange Funktion“ bezeichnet man eine Funktion, deren Inhalt weit über ihren Namen hinausgeht. Es wird seit langem angenommen, dass die langen und komplexen Funktionen schwer zu verstehen und zu warten sind. Zudem sind sie fehlerträchtiger und mühsamer zu erweitern als die kürzeren Funktionen.

2.8.3. Große Module

Große Module (Klassen/Dateien) besitzen meistens viele Verantwortlichkeiten und erschweren dadurch die Lesbarkeit und die Verständlichkeit des Quellcodes. Module mit vielen Attributen und mehreren Methode sind ein Beispiel dafür.

2.8.4. Lange Parameterliste

Funktionen mit langen Parameterlisten erschweren unnötig die Lesbarkeit und die Verständlichkeit des Sourcecodes. Zudem wird auch die Anzahl der benötigten Testfälle enorm erhöht.

2.8.5. Toter Code

Als „toter Code“ werden in der Programmierung Codestellen bezeichnet, die nie ausgeführt werden. Diese Codefragmente erhöhen erheblich die Komplexität, die Lesbarkeit und dadurch die Wartbarkeit der Software. Zudem wird durch die Ausführung des toten Codes unnötig CPU-Zeit vergeudet.

2.8.6. Kommentare

Kommentare sind eine wichtige und mächtige Dokumentationsart für den Quellcode. Öfter werden aber die Kommentare genutzt, um schlecht geschriebene Codestellen zu verdecken. Dabei ist das Ziel eines Kommentars nicht den Code zu beschreiben, sondern die Entwurfsgedanken der Entwickler. Die Kommentare im Programm sind für die Wartung eines Softwaresystem von einer sehr großen Bedeutung.

2.8.7. Architektur Smells

Neben den von Beck und Fowler beschriebenen Code Smells existieren auch „Architektur Smells“ [LR04]. Ein typisches Beispiel dafür sind die unnötigen Abhängigkeiten, die im Extremfall zu Abhängigkeitszyklen führen konnten. Die Symmetrie und die Aufteilung der Subsysteme zählen auch unter anderem zu den „Architektur Smells“. Diese Schwachstellen in der Architektur erschweren enorm die Weiterentwicklungsfähigkeit und die Wartbarkeit der Softwaresysteme. Deswegen ist es empfehlenswert, durch dafür geeigneten Werkzeuge und manuelle Reviews, die problematischen Stellen in der Architektur so schnell wie möglich zu finden und zu verbessern.

3. Analyse

In diesem Kapitel wird das genaue Vorgehen bei der Analyse des Qualitätssicherungsprozesses der „XYZ“ AG beschrieben. Zuerst werden die Vorbereitungsschritte erläutert, dann die eingesetzten Methoden und Techniken der Analyse und zum Schluss die Auswertung der Ergebnisse.

3.1. Vorgehen bei der Analyse

Um den aktuellen Zustand des Qualitätssicherungsprozesses zu ermitteln, wurde die Methode der Befragung angewendet. Nach wie vor wird dieser Ansatz als einer, der Besten und meist eingesetzten wissenschaftlichen Methoden zum Informationsgewinn angesehen. Dabei wird es je nach Standardisierungsgrad und Kommunikationsform zwischen schriftlicher und mündlicher Befragung unterschieden. Die erste Aufgabe dieser Diplomarbeit bestand darin, eine schriftliche Befragung der „XYZ“-Mitarbeiter mithilfe von Fragebögen durchzuführen. Die Entscheidung, für die Durchführung der Analyse mittels Fragebögen, wurde nicht von mir getroffen, sondern stand als Teil der Diplomarbeit von Anfang an fest.
Um detaillierte Informationen über die Qualitätssicherung des Unternehmens „XYZ“ zu bekommen, entschied ich mich später noch eine mündliche Befragung in Form von Interviews abzuhalten.
Viele der Fragen, die den Qualitätssicherungsprozess betrafen, wurden von meinem Betreuer bei der „XYZ“ AG oder vom Leiter der C-Abteilung beantwortet. Dafür musste ich keine explizite Vorbereitungen, wie bei Erstellung der Fragebögen und Durchführung der Interviews, treffen. Die Antworten haben sich durch die wöchentlichen Treffen mit meinem Betreuer und dem C-Abteilungsleiter, ergeben. Zum Ende der Analyse wurden die Ergebnisse der Fragebögen ausgewertet und in Form von Grafiken und Charts dargestellt. Zudem wurde auch ein kleiner Bericht erstellt und dem Betreuer, sowie den Leitern der C- und Java-Abteilungen zur Verfügung gestellt.
Die Ergebnisse der Interviews wurden hingegen, nur in Form eines kleinen Berichtes erfasst, und in einem der wöchentlichen Statusmeetings mit meinem Betreuer bei „XYZ“ AG kommuniziert. Nach dem Abschluss der schriftlichen und mündlichen Befragungen wurden die Ergebnisse der Auswertung zusammengefasst und in der Ausarbeitung dokumentiert.

3.2. Vorbereitung der Analyse

In der Vorbereitungsphase wurde eine Literaturrecherche zu den Themengebieten „Software Messung“, „Werkzeugunterstützung für kontinuierliche Qualitätsanalyse“ sowie „Qualitätsdefizite langlebiger Softwaresysteme“ durchgeführt. Zusätzlich wurde auch nach Literatur und wissenschaftlicher Arbeiten zum Thema „empirische Untersuchungen“ gesucht. Die Literaturrecherche sollte der Aneignung, des für die Durchführung der Diplomarbeit und der Erstellung der Fragebögen nötigen Wissens, dienen. Zeitgleich wurde auch die Möglichkeit untersucht die schriftliche Befragung, online mittels dafür geeignete Software durchzuführen. Aufgrund der Komplexität und der Kosten, die durch solch eine Software entstehen, habe ich mich entschieden, den Fragebogen in Papierform zu verfassen. Zu gleichen Zeit wurden die ersten Fragen der Befragung notiert, die aus Literaturrecherche und Rücksprache mit dem Universitätsbetreuer entstanden sind.

3.3. Fragebogen

Die erste Phase der Befragung wurde anhand eines Fragebogens durchgeführt. Hierfür wurden alle gesammelten Fragen mit meinen zwei Betreuern besprochen, verfeinert und als Fragebogen formuliert. Der Fragebogen enthielt sowohl geschlossenen als auch offenen Fragen.
Bei den geschlossen Fragen konnte zwischen mehreren verschiedenen Antworttypen unterschieden werden. Es traten Fragen auf, bei denen es nur eine mögliche Antwort gab. Ein Beispiel für solch einen Typ von Fragen sind Fragen, die „Ja“ oder „Nein“ als Antwortmöglichkeit haben.
„Multiple-choice“ Fragen waren ein weiter Typ von Fragen, die in dem Fragebogen auftraten. Bei diesem Fragetyp hatten die Befragten die Möglichkeit mehrere Antworten anzugeben.

Abbildung 3.1.: Beispiel für Multiple-choice Frage

Um den Befragten die Möglichkeit zu geben, selber seine Antworten zu formulieren und nicht nur vordefinierte Antwortmöglichkeiten auszuwählen, wurden auch halb-offene und offene Fragen ausgearbeitet.
Speziell bei den halb-offenen Fragen hatte der Befragte außer vordefinierten, auch eigene Antwortmöglichkeiten. Durch die Antwortmöglichkeiten „Sonstiges“ oder „Andere“ konnte er selber seine Meinung ausdrücken, falls keine der Antworten zutreffend war.

Abbildung 3.2.: Beispiel für halb-offene Frage

3.3.1. Aufbau des Fragebogens

Wie bereits erwähnt, wurde der Fragebogen mit dem Ziel erstellt, den Qualitätssicherungsprozess der „XYZ“ AG zu analysieren, sowie die Zufriedenheit der Mitarbeiter mit diesem Prozess zu ermitteln. Daher enthielt der Fragebogen folgenden Hauptfragen:

- Mit welchen Methoden des Software-Engineering werden bei „XYZ“ AG die Software-Qualitätssicherungsprozesse verfolgt?
- Denken Sie, dass die eingesetzten Methoden des Software-Engineering bei „XYZ“ AG zu Verbesserung der Software-Qualität beitragen?
- Wo sehen Sie Verbesserungsmöglichkeiten oder den größten Handlungsbedarf in dem Software-Qualitätssicherungsprozess der „XYZ“ AG?

Diese Fragen entstanden nach der Literaturrecherche und mit Rücksprache mit meinem Universitätsbetreuer. Allgemein gliederte sich der Fragebogen wie folgt:

- Personenbezogene Fragen
- Allgemein Fragen über Metriken
- Fragen über das Softwareanalyse-Werkzeug Sonar
- Fragen über den Qualitätssicherungsprozess bei „XYZ“ AG

Die Einstiegsfragen des Fragebogens dienten vorwiegend dazu, sich einen Überblick über die IT-Erfahrung der befragten Person, sowie über die eingesetzte Programmiersprache und die Größe der durchgeführten Projekte zu verschaffen.

Abbildung 3.3.: Beispiel für eine personenbezogene Frage

Die metrikbezogene Fragen wurden gestellt, um die Kenntnisse der Mitarbeiter über Softwarequalitätsmetriken zu erfassen. Nach wenigen Tagen habe ich erfahren, dass die Java-Abteilung Sonar als Werkzeug zum Softwarequalitätsanalyse einsetzt. Daher wurden auch Fragen über den Einsatz und die Ausgaben von Sonar gestellt. Die letzten Fragen des Fragebogens dienten dazu, zu erfahren, wo der größte Handlungsbedarf bezüglich der Softwarequalitätssicherung bei „XYZ“ AG bestand.
Der Fragebogen selbst befindet sich im Anhang A.

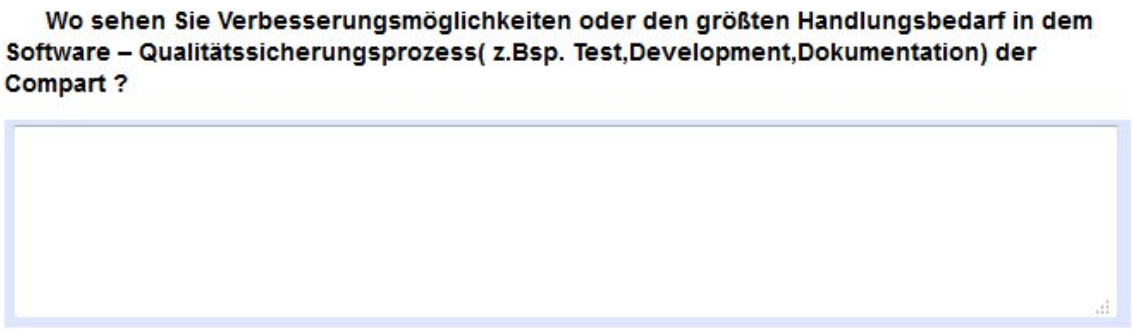


Abbildung 3.4.: Beispiel für eine Frage über den Qualitätssicherungsprozess

3.3.2. Schwierigkeiten während der Befragung

Vor und während der Befragung traten gewisse Schwierigkeiten auf. Da die erste Version des Fragebogens zu umfangreich war, wurden viele von den Fragen ausgemustert, wodurch die Anzahl der Fragen auf 23 reduziert wurde.
Ein anderes Problem war die Einhaltung des Abgabetermins. Viele Fragebögen kamen eine Woche später oder gar nicht zurück. Insgesamt wurden 35 Fragebögen verteilt, allerdings kamen nur 27 davon zurück. Leider waren nicht alle Fragebögen vollständig ausgefüllt, so dass sich die Auswertung enorm erschwerte.
Als größtes Hindernis empfand ich aber, dass mir das Management nicht erlaubt hat, viele der Fragen, die für meine Diplomarbeit relevant waren, zu stellen. Durch dieses Problem erschwerte sich für mich die Durchführung der Analyse enorm. Daher entschied ich mich nach Rücksprache mit meinem Universitätsbetreuer, eine mündliche Befragung in Form von Interviews durchzuführen.

3.4. Interviews

Die Interviews wurden mit dem Ziel durchgeführt, einen detaillierten Überblick über den Qualitätssicherungsprozess des Unternehmens zu erhalten. Es sollte untersucht werden, ob und welche Maßnahmen zur Messung und zur Sicherung der Softwarequalität unternommen werden. Zudem sollten die Wartungsprobleme und der Werkzeugeinsatz zur Analyse des

Quellcodes ermittelt werden. Weiterhin sollte in den Interviews auf die einzelnen Angaben im Fragebogen näher eingegangen werden.

3.4.1. Vorgehen

Die Interviews wurden in Form eines Gespräches an zwei aufeinander folgenden Tagen durchgeführt und dauerten durchschnittlich 45 Minuten. Die Interviewpartner wurden nicht von mir, sondern in Vorfeld von dem Management des Unternehmens ausgesucht. Es wurden zwei Java- und zwei C-Entwickler befragt. Während der Befragung wurden offene Fragen gestellt, die ein freies Antwortverhalten des Interviewgastes ermöglicht haben.
Im Laufe der mündlichen Befragung wurde jede Antwort notiert und in einem kleinen Bericht zusammengefasst.

3.4.2. Konstruktion

Die Interviews wurden auf die selbe Art und Weise, wie der Fragebogen aufgebaut. Am Anfang wurden Fragen zu den aktuellen Tätigkeitsbereich der jeweiligen Entwickler gestellt. Als nächstes folgten Fragen über den Einsatz von Metriken und der verfolgten Qualitätsziele von „XYZ“ AG bei der Erstellung ihrer Software. Zum Schluss wurden noch Fragen über den Einsatz von Softwareanalyse-Werkzeugen und über die Probleme während der Softwarewartung gestellt. Die Interviews enthielten folgende Hauptfragen:

- Welche Qualitätsziele verfolgt der jeweilige Entwickler, während der Softwareerstellung?
- Werden Metriken zur Messung der Software-Qualität eingesetzt? Falls ja, welche?
- Ist der jeweilige Entwickler mit der Wartung betraut?
- Welche Probleme treten, während der Softwarewartung auf?
- Setzen Sie spezielle Werkzeuge zum Finden von bestimmten Fehlern ein?

Im großen und ganzem wurden bei allen Teilnehmern die selben Fragen gestellt, jedoch traten abhängig von den Antworten des Interviewpartners neue Fragen auf, oder die bestehenden wurden variiert.

3.5. Ergebnisse der Befragung

In diesem Kapitel werden die Ergebnisse der schriftlichen und der mündlichen Befragung vorgestellt.

3.5.1. Ergebnisse der schriftlichen Befragung

3.5.1.1. Teilnehmer der Befragung

Als Erstes werden die Teilnehmer der Befragung kurz beschrieben. Es werden Information über ihre IT-Erfahrung, die von Ihnen eingesetzten Programmiersprachen, sowie die Größe der von Ihnen durchgeführten Projekte dargestellt und kurz erläutert. Die Untersuchung

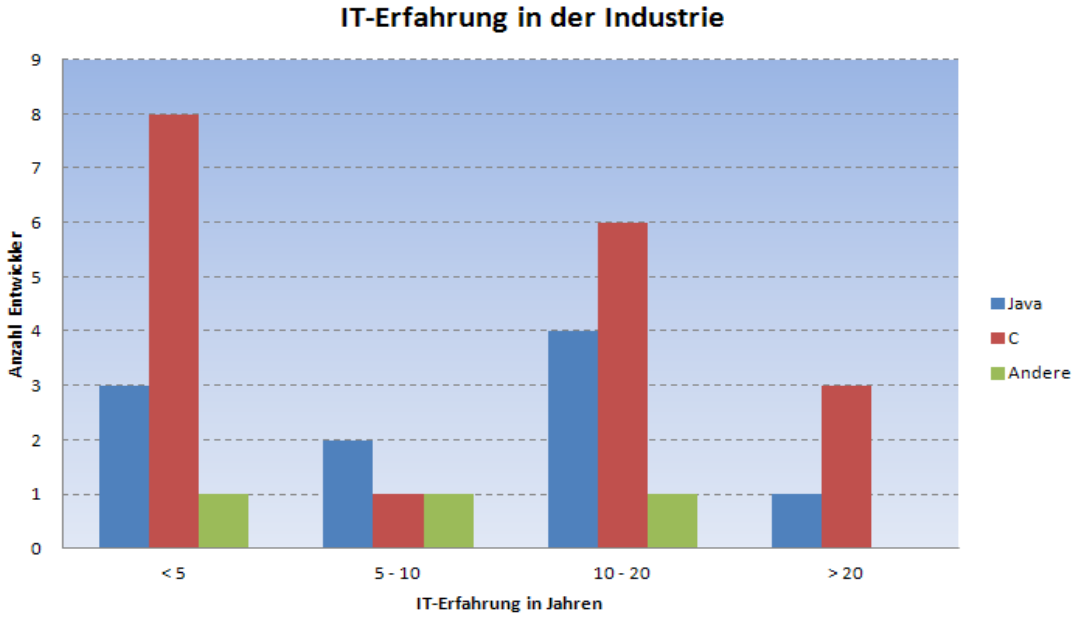
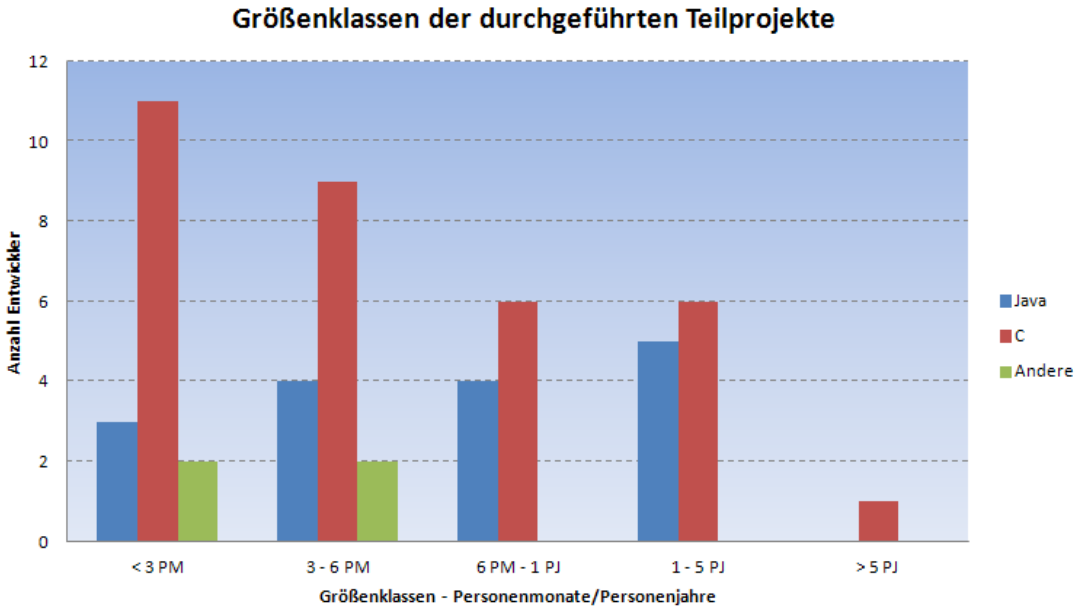


Abbildung 3.5.: IT-Erfahrung in der Industrie

zeigte, dass zwei Drittel der Befragten während der Arbeit die Programmiersprache C einsetzen. Der Rest beschäftigt sich hauptsächlich mit Java. Die Mehrheit der befragten C-Entwickler haben angegeben, dass sie weniger als fünf Jahre Erfahrung in der IT-Industrie haben. Dagegen gab der Großteil der Java-Entwickler an, zwischen fünf und zehn Jahre IT-Erfahrung zu besitzen. Die durchgeführten Projekte sind unterschiedlichen Größenklassen zuzuordnen. Die meisten Entwickler in der C-Abteilung arbeiten in Teilprojekten, die weniger als drei Personenmonate benötigen. Die Java-Entwickler hingegen arbeiten meistens in Projekten von ein bis fünf Personenjahren Dauer.



3.5.1.2. Stellenwert der Softwarequalität und Metrik-Kenntnisse

Als nächstes soll auf den Stellenwert, den die Softwarequalitätsanalyse bei den Entwicklern einnimmt und auf deren Kenntnisse über Softwarequalitätsmetriken näher eingegangen werden.

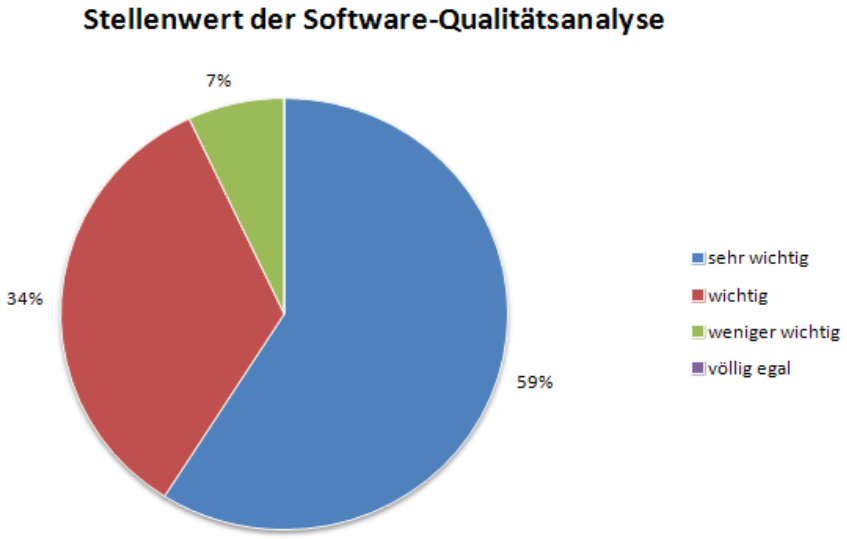


Abbildung 3.7.: Stellenwert der Software-Qualitätsanalyse

Für den größten Teil der Befragten sind die Softwarequalität und die Softwarequalitätsanalyse sehr wichtig. Es gibt aber leider auch Entwickler, die die Softwarequalitätsanalyse als weniger wichtig ansehen, darunter auch solche, die mehr als 20 Jahre Erfahrung in der IT-Industrie haben. Ein wenig mehr als die Hälfte der Entwickler konnte Softwarequalitätsmetriken benennen, wobei Mehrfachnennungen möglich waren. Die meist genannte Metrik war „Lines of Code“. Einige der befragten Personen haben die agile Software-Entwicklung und die „Continuous Integration“[1] fälschlicherweise als Softwarequalitätsmetriken angegeben. Dies zeigt, dass bei diesen Entwicklern das Wissen über Softwarequalitätsanalyse und -messung sehr begrenzt ist. Die Abkürzungen „DSQI“ und „LNCSC“ im Bild 3.8 stehen für „Design Structure Quality Index“ und „Lines of Code per Function“.

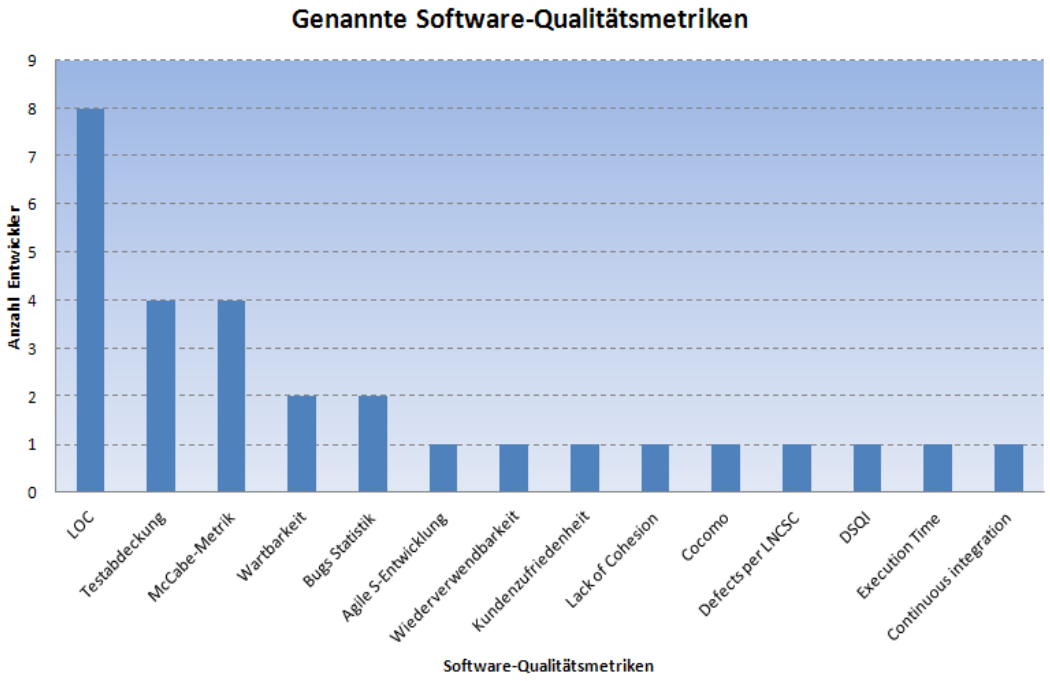


Abbildung 3.8.: Genannte Softwaremetriken

3.5.1.3. Software Engineering Methoden und größter Handlungsbedarf

Nun werden die bei „XYZ“ AG eingesetzten Methoden des Software Engineering und deren Beitrag zur Verbesserung der Softwarequalität erläutert. Außerdem wird aufgezeigt, wo der größte Handlungsbedarf im Qualitätssicherungsprozess besteht. Sowohl die Java- als auch die C-Programmierer haben das systematische Testen und die Einhaltung von Programmierkonventionen als die am meisten eingesetzten Methoden des Software Engineering angegeben.

[1] Ein Prozess der permanente Integration einer Anwendung

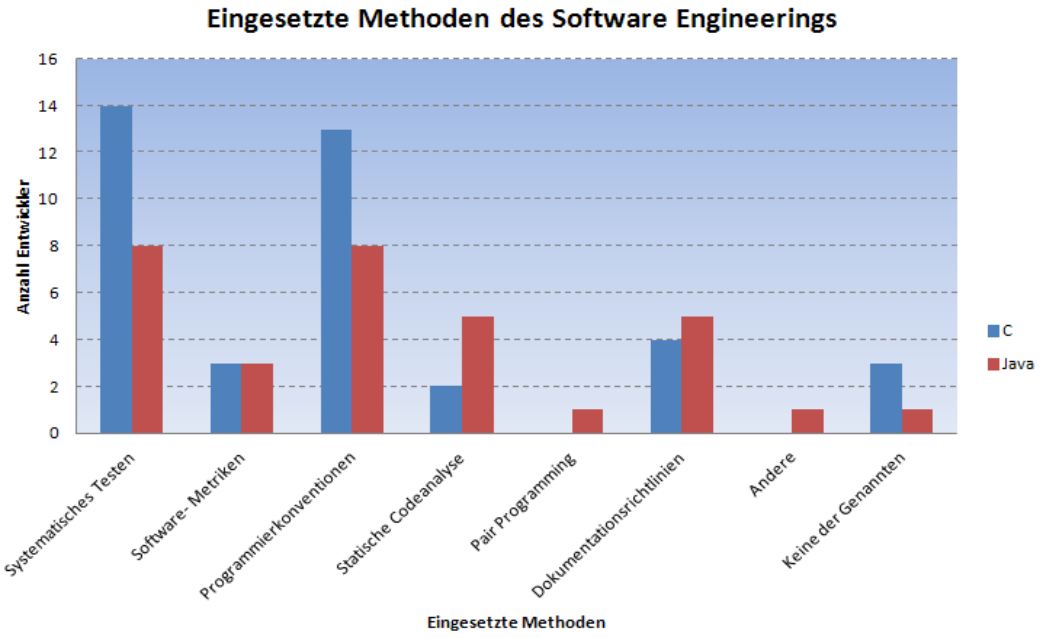
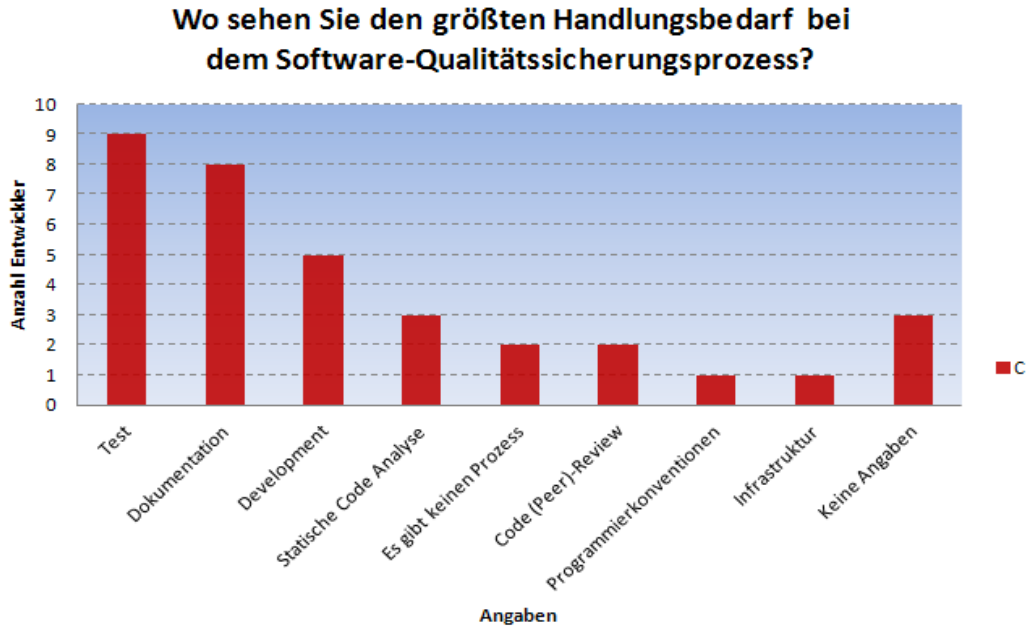


Abbildung 3.9.: Eingesetzten Methode des Software Engineerings

Ein wenig mehr als die Hälfte der Befragten(63%) ist der Meinung, dass die eingesetzten Methoden des Software Engineering zu Verbesserung der Software Qualität beitragen. Die restlichen Befragten(37%) stimmen dem gar nicht oder nur teilweise zu.



Den größten Handlungsbedarf im Softwarequalitätssicherungsprozess sehen sowohl die Java als auch die C-Entwickler im Test- und Basisentwicklungsbereich, gefolgt von der Dokumentation und der Planung. Die C-Entwickler geben an, dass in ihre Abteilung überhaupt keine Softwarequalitätsanalyse durchgeführt wird und sie sich die Einführung eines Qualitätssicherungsprozesses wünschen. Vor allem soll dieser Prozess in die Scrum Meetings des Unternehmens miteinbezogen werden.

Abbildung 3.11.: Größter Handlungsbedarf

3.5.1.4. Vorschläge der Entwickler zur bessere Softwarequalität

Im folgenden Abschnitt werden die Vorschläge der „XYZ"-Entwickler zur Verbesserung des Qualitätssicherungsprozesses aufgezeigt. Als Vorschlag zur Verbesserung der Softwarequalitätsanalyse im Scrum Umfeld wird die Einführung von Code Reviews genannt. Die Entwickler sind der Meinung, dass die Softwarequalitätsstandards in die „Definition of Done"[2] aufgenommen werden sollten. Ein paar der Befragten empfinden die Einbeziehung eines Qualitätssicherungsprozesses in das Scrum als einen großen Schritt in Richtung bessere Softwarequalität. Viele Entwickler sind der Meinung, dass man die Tests besser einplanen sollte und dass mehrere automatisierte Tests geschrieben werden sollten. Die Befragten sind der Ansicht, dass man Standards für Test, Coding und Dokumentation einführen und die

[2]Eine Checkliste von Aktivitäten, die zur Implementierung einer User Story gehören und die Qualität der Software beeinflussen

Einhaltung dieser Standards auch kontrollieren sollte. Die C-Entwickler vertreten außerdem den Standpunkt, dass eine statische Codeanalyse und Peer-Reviews eingeführt werden sollten, damit sich die Softwarequalitätsanalyse verbessern kann. Ein weiterer Verbesserungsvorschlag ist eine bessere und detailliertere Dokumentation der Architektur und der Funktionsweise der Software einzuplanen.

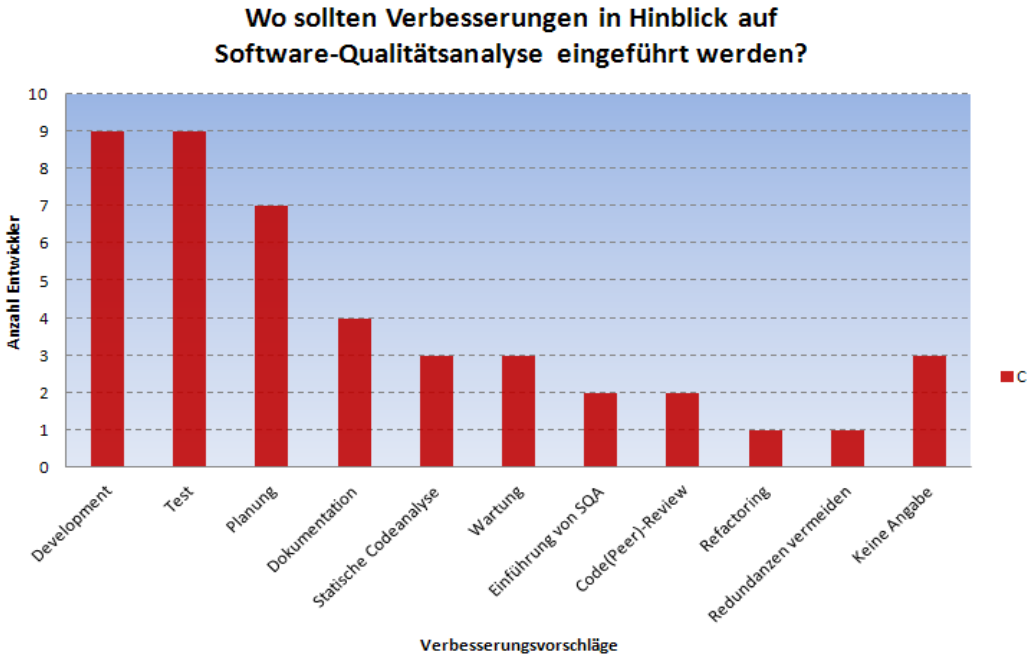


Abbildung 3.12.: Verbesserungsvorschläge der Mitarbeiter

Laut Angaben der Entwickler versuchen sie den Softwarequalitätssicherungsprozess zu verbessern, in dem sie eigene Tests schreiben und Programmierkonventionen einhalten.

3.5.1.5. Sonar-Einsatz

In den ersten Wochen der Diplomarbeit hat sich herausgestellt, dass die Java-Abteilung zur Softwarequalitätsanalyse das Werkzeug Sonar einsetzt. Aus diesem und dem Grund, dass es bei der Diplomarbeit, um den Einsatz eines Softwarequalitätsanalysewerkzeugs namens ConQAT ging, wurden Fragen im Fragebogen aufgenommen, die den Vergleich der beiden Werkzeuge erlauben. Der größte Teil der Sonar Anwender(60%) ist der Meinung, dass sich die Wartung durch das eingesetzte Werkzeug erleichtert. Der Einsatz von Sonar dient laut den Entwicklern vor allem der Überprüfung der Einhaltung von Programmierkonventionen während der Softwareentwicklung. Jedoch haben nicht alle Entwickler angegeben, dass sie Sonar kontinuierlich einsetzen. Nach Aussagen der Entwickler wird Sonar unterschiedlich

oft eingesetzt. Manche nutzen es nur 2 mal monatlich zum Ende der Sprintphase (Entwicklungsphase des Vorgehensmodells Scrum) hin, andere wiederum mehrmals wöchentlich, hauptsächlich bei neuen und größeren Codeblöcken.

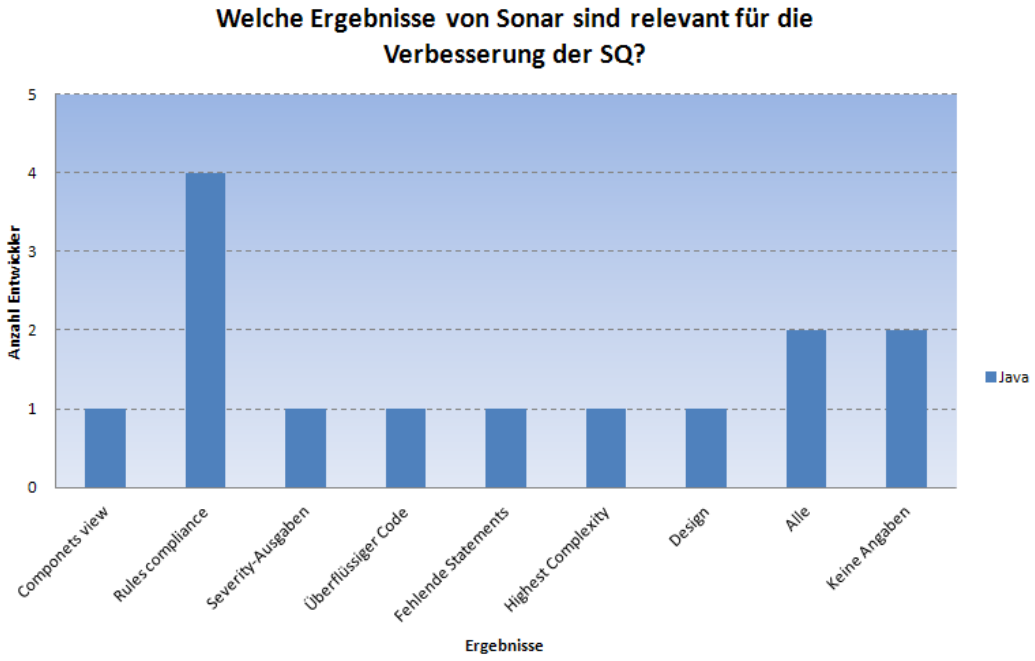


Abbildung 3.13.: Relevante Ergebnisse von Sonar

3.5.2. Ergebnisse der mündlichen Befragung

Die Interviews wurden mit Mitarbeiter durchgeführt, die im Vorfeld vom Unternehmensmanagement ausgesucht worden sind. Es wurden zwei Java und zwei C Entwickler befragt. Die Interviews wurden in Form einer Unterhaltung durchgeführt und dauerten knapp 45 Minuten. Laut Angaben der Interviewpartner werden im Unternehmen keine Metriken erhoben, sei es für Planung, Projekt, Kostenschätzung oder Produktmetriken. Die Befragten haben angegeben, dass im Unternehmen keine Qualitätsziele verfolgt werden und die Qualität des Codes nicht analysiert wird. Nach den Aussagen der Befragten werden während des Softwareentwicklungsprozesses, keine manuelle Inspektionen des Codes, der Dokumentation oder der Spezifikation durchgeführt. Die Java Entwickler gaben allerdings an, dass in ihrer Abteilung eine Art von „Code Reading“ stattfindet. Es zeigte sich auch, dass in den Scrum Meetings keine Qualitätssicherungsmaßnahmen diskutiert werden. Die Befragten teilten mit, dass sie die Architektur der entwickelten Software nicht kennen und Abweichungen von der Architektur nicht analysiert werden. Die C-Entwickler haben erwähnt, dass der Code meistens unkommentiert ist und keine richtige Dokumentation vorhanden sei.
Für die Verbesserung der Qualitätssicherung versuchen die Programmierer eigene Unit-Tests zu schreiben und die Programmierrichtlinien des Unternehmens einzuhalten. Den größten

Handlungsbedarf sehen die Befragten bei den Tests und bei der schwammigen Spezifikation. Einerseits seien die Test nicht richtig geplant, andererseits erschweren die fehlenden oder unklar definierten Anforderungen und Use Cases, die Arbeit der Entwickler. Daher wissen die Tester meistens nicht, ob ein beim Testen gefundener Fehler, wirklich Fehler sind. Im Allgemeinen waren die Entwickler der Meinung, dass der Qualitätssicherungsprozess nicht klar definiert oder gar nicht vorhanden sei.
Die Java Entwickler haben angegeben, dass sie das Softwarequalitätsanalysewerkzeug Sonar einsetzen. Auf die Frage, welche Regeln von Sonar relevant für die Analyse sind, konnte sie keine genaue Antwort geben, aus dem Grund, dass sie meinten nicht alle regeln zu kennen. Auf die Frage, was sie durch den Einsatz von Sonar erreichen können, meinte sie die Kontrolle der Einhaltung der Programmierkonventionen und vordefinierten Regeln. Die Ausgaben von Sonar werden aber nicht im Team diskutiert, vielmehr betrachtet jeder Entwickler die Ergebnisse von Sonar für sich selbst.
Aus Sicht der befragten C-Entwickler macht die Wartung den größten Teil der Arbeit während der Entwicklung aus. Die Entwickler schätzten die Kosten der Wartung auf 70 % der gesamten Entwicklungskosten. Die Java Entwickler hingegen haben angegeben, dass sie keine richtige Wartung durchführen, da die Java Abteilung noch keine richtige Software verkauft. Als größte Schwierigkeit während der Wartung haben die C-Entwickler die schlechte Dokumentation und den teilweise schlecht geschriebenen Code angegeben. Zu den meistgenannten Qualitätsdefiziten wurden die großen Funktionen und Dateien, sowie die Komplexität dieser Funktionen und Dateien genannt. Als eine andere Schwachstelle wurde die fehlende Dokumentation der Softwarearchitektur angegeben. Die Ursache für die Qualitätsdefizite des Codes ist die fehlende Zeit für Refactoring und Qualitätssicherung. Bezüglich der Softwarequalitätsanalyse haben die C-Entwickler gesagt, dass in ihrer Abteilung kein Werkzeug zur Codeanalyse eingesetzt wird.

3.6. Schlussfolgerungen

Im Unternehmen existiert kein Qualitätssicherungsprozess. Es werden keine Metriken zur Planung, Wartung oder allgemein zur Qualitätskontrolle und -steuerung erhoben. Während der Softwareentwicklung werden die nicht-funktionalen Anforderungen, vor allem in der C -Abteilung, nicht beachtet. Es existiert kein Qualitätsmodell an dem sie sich die Entwickler orientieren können. Es werden keine Qualitätsziele oder Qualitätsmerkmale für die Projekte definiert. Die Qualität der Software und der Entwicklungsprozesse wird auf keine Art und Weise gemessen. Die Qualitätssicherung besteht ausschließlich aus Tests. Leider werden nicht alle Testfälle dokumentiert.
Die Entwickler kennen die Architektur ihrer Software nicht, da die Dokumentation hierfür fehlt. Die Architekturabweichungen werden, weder mittels manuellen Inspektionen, noch durch Werkzeugunterstützung analysiert.
In der C-Abteilung findet keine Art von manuelle Inspektionen, wie Code- oder Spezifikationsreviews statt. In der Java-Abteilung werden auch keine Code Reviews durchgeführt, allerdings findet eine Art von Code Reading statt. Als Wartungsprobleme geben die Entwickler die großen und komplexen Funktionen, sowie die fehlende Dokumentation an.

Viele der Entwickler sind der Meinung, dass die fehlende Zeit zum Refactoring und zur Qualitätssicherung zu den Qualitätsdefiziten im Code geführt hat. Die Entwickler versuchen durch eigene Test und durch die Einhaltung der Programmierrichtlinien qualitativ hochwertige Software zu erstellen. Als weitere Schwachstellen werden auch die schwammige Spezifikation und die schlecht geplanten Tests angegeben. Diese Schwachstellen führten dazu, dass sich das Testen der Software enorm erschwert, da die Tester die Testausgaben nicht nach ihrer Richtigkeit und Wichtigkeit einordnen können.
Als positiv empfand ich, dass die Java Abteilung Sonar als Werkzeug zur Unterstützung während der Entwicklung einsetzt. Allerdings wird das Werkzeug nicht von allen Entwicklern kontinuierlich eingesetzt. In der C-Abteilung hingegen wird zur Unterstützung der Entwickler kein Werkzeug eingesetzt. Ein anderer positiver Punkt ist die Einführung von Scrum als Vorgehensmodell der Software Entwicklung. Leider werden aber in den Scrum Meetings keine Qualitätssicherungsmaßnahmen behandelt.
Zusammenfassend kann man sagen, dass die internen Qualitätsmerkmale, wie die Wartbarkeit oder die Weiterentwicklungsfähigkeit der Software nicht beachtet und als wichtig empfunden werden. Worauf man sich bei der Qualitätssicherung konzentriert, ist das Testen der äußeren Qualitätsmerkmale, wie z. B. die Funktionalität. Dies wird aber dadurch erschwert, dass die Anforderungen nicht klar definiert und die Tests nicht richtig eingeplant werden.

4. Kontinuierliche Qualitätsanalyse von Softwaresysteme

In diesem Kapitel werden die Gründe für eine kontinuierliche Qualitätsanalyse und ihre Umsetzung in der Praxis beschrieben.

4.1. Kontinuierliches Qualitäts-Controlling

Die Qualität der Software ist ein wesentlicher Faktor für den wirtschaftlichen Erfolg von Softwaresystemen, die langlebig und erweiterbar sein sollen. Sie hat vor allem entscheidenden Einfluss auf die Effizienz der Wartung und Weiterentwicklung [DHJ08]. Erfahrungsgemäß nimmt aber die Qualität der Software mit der Zeit ab.
Die kontinuierlichen Änderungen und Erweiterungen des Quellcodes, sich ständig ändernde Anforderungen und die Anpassungen der Software an unterschiedliche technische Umgebungen können zum schleichenden Qualitätsverfall führen.
Schon mit den kleinen Änderungen, die unter Zeitdruck durchgeführt werden, nimmt die Unordnung stark zu. Die Architekturgrundsätze werden nicht mehr eingehalten. Stellen im Quellcode werden der Einfachheit halber kopiert. Zudem werden auch die Programmierkonventionen beiseite geschoben
Wenn noch dazu die Änderungen am System von Entwicklern ausgeführt werden, die keine Kenntnisse über die zugrunde liegenden Architektur haben oder diese Architektur überhaupt nicht beachten, erhöht sich die Komplexität der inneren Struktur des Softwaresystems enorm. Als Folge dieser Versäumnisse entsteht ein System, das mit der Zeit von niemandem mehr richtig verstanden wird. Die Änderungen und die Erweiterungen lassen sich nur mit sehr großer Mühe und höherem Zeitaufwand durchführen. Dadurch sinkt die Produktivität und die Wartungskosten steigen enorm.
Um die Ordnung in solch einem System wieder herzustellen, wird viel Zeit und Mühe in der späteren Wartungsphase benötigt.
Um diesem Verfall der Qualität entgegenzuwirken, ist es notwendig gewisse Regeln zu definieren und durch eine kontinuierliche Analyse die Einhaltung dieser Regeln zu kontrollieren. Dabei soll nicht nur der Entwicklungsprozess, sondern auch das Produkt selbst analysiert und verbessert werden. Im Fall Software soll diese Überprüfung nicht nur auf Code-Ebene stattfinden, sondern die Dokumentation und die Architektur des Softwaresystems sollen ebenfalls einer kontinuierlichen Qualitätskontrolle unterzogen werden.
Auf Grund der Größe heutiger Softwaresysteme ist es empfehlenswert solch eine Qualitäts überprüfung sowohl durch manuelle Inspektionen, als auch mittels dafür geeigneten

Werkzeugen durchzuführen [Dei10]. Die Werkzeuge, die zur Operationalisierung der kontinuierlichen Softwarequalitätsanalyse dienen, werden Software Quality Dashboards genannt. Diese Werkzeuge sammeln und liefern Daten über den aktuellen Qualitätszustand eines Softwaresystems in Form von Tabellen, Trends- und Übersichtsgrafiken. Mit Hilfe dieser Daten können später Aussagen über die Qualität und die Weiterentwicklungsfähigkeit des Softwaresystems gemacht werden.

4.2. Umsetzung des Qualitäts-Controllings

Um eine sinnvolle kontinuierliche Qualitätsanalyse in der Praxis durchführen zu können, muss an erster Stelle ein dafür geeigneter Prozess vorhanden sein. Dieser Prozess soll die Qualitätsanalyse bei ihrer Umsetzung und Durchführung unterstützen. Im Falle, dass ein Qualitätssicherungsprozess im Unternehmen schon vorhanden ist, könnte die kontinuierliche Qualitätsüberprüfung als Teil dieses Prozesses integriert und umgesetzt werden. Falls der Qualitätssicherungsprozess nicht existiert, ist es empfehlenswert solch einen Prozess zu definieren, damit die Analyse tatsächlich kontinuierlich und nicht nur gelegentlich durchgeführt wird. Außerdem kann durch solch einen Prozess, die Aufgabenverteilung und die Kommunikation im Unternehmen bezüglich der Qualitätssicherung verbessert werden. Ein weiteres und wichtiges Kriterium für die kontinuierliche Qualitätsanalyse ist die Wahl von einem geeigneten Werkzeug. Es soll die automatische Ausführung der Analyse unterstützen. Dabei ist es wichtig, dass dieses Werkzeug die durch das Projekt vorgegebenen Anforderungen erfüllt. Das Werkzeug soll die festgelegten Qualitätskriterien maximal präzise analysieren und auswerten können, damit die kontinuierliche Qualitätsanalyse zum Erfolg wird. Vereinfacht kann das kontinuierliche Qualitäts-Controlling als eine Regelungsschleife betrachtet werden (siehe Bild 4.1) [DH11].

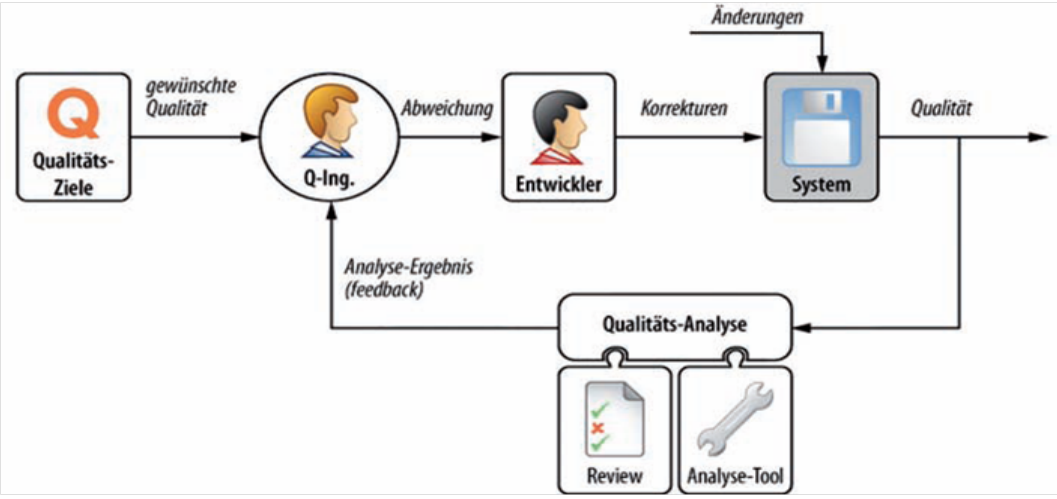


Abbildung 4.1.: Das Qualitäts-Controlling als Regelungsschleife (F.Deißenböck, B.Hummel)

Im Mittelpunkt dieser Schleife steht das zu analysierende Softwaresystem, das permanent Einflüssen von außen ausgesetzt ist. Diese Einflüsse können z. B. die neuen Anforderungen oder die durch die Entwickler durchgeführten Codeänderungen sein. Man nimmt zudem an, dass das Softwaresystem eine inhärente Qualität besitzt, die durch eine Qualitätsanalyse festgestellt werden kann. Für die Einhaltung der Qualitätsziele ist ein Qualitätsingenieur verantwortlich. Er definiert Regeln und fordert die Entwickler auf, diese einzuhalten. Außerdem ergreift er auf die Probleme abgestimmte Maßnahmen zur Qualitätsverbesserung. Die Rolle des Qualitätsverantwortlichen kann entweder von den Entwicklern selbst oder von dafür entsprechend ausgebildeten Experten übernommen werden. Bei großen Softwareprojekten mit komplexen Analyseszenarien ist es empfehlenswert die Rolle des Qualitätsingenieurs einer dafür entsprechend ausgebildeten Person zu übertragen.
Um bei Bedarf rechtzeitig auf die Qualitätsdefizite reagieren zu können, ist es sinnvoll, diese Schleife kontinuierlich und nicht nur gelegentlich zu durchlaufen.
Die wichtigsten zwei Komponenten dieser Schleife sind die vordefinierten Qualitätsziele und die durchzuführende Qualitätsanalyse.
Vor jedem Projekt sollte klar sein, welche Qualitätsziele zu verfolgen sind. Dabei unterscheiden sich die Qualitätsziele in Abhängigkeit vom jeweiligen Projektkontext. Deshalb ist es wichtig sie sorgfältig zu analysieren und nur projektspezifische Ziele zu definieren, die wirklich betrachtet und verfolgt werden sollen.
Die Qualitätsanalyse ist die zweite wichtige Komponente der Regelungsschleife. Sie verschafft einen detaillierten Überblick über den aktuellen Qualitätszustand des Softwaresystems.
Eine wirksame Qualitätsanalyse besteht dabei aus zwei Teilen: den automatischen werkzeugunterstützten Analysen und den manuellen Inspektionen.
Durch die automatische Analyse kann in kurzer Zeit ein schneller Überblick über die gesamte Quellcodemenge vermittelt werden. Diese Analyse kann aber nur einen sehr begrenzen Teil aller projektspezifischen Qualitätskriterien überprüfen. Deshalb ist es sinnvoll sie durch manuelle Inspektionen zu ergänzen, die in Form von Reviews, wie z. B. Pair-Programming oder Peer-Reviews, durchgeführt werden können.
Bei der automatischen Überprüfung ist die richtige Wahl eines Qualitätsanalyse-Werkzeugs von enormer Bedeutung. Es ist wichtig, dass das Analysewerkzeug die projektspezifischen Qualitätskriterien überprüfen und in einer einheitlichen Form darstellen kann. Zudem sollten solche Metriken definiert und jene Qualitätskriterien überprüft werden, die auf unmittelbare Probleme im Softwaresystem hinweisen. Beispiele für solche Analysen sind die Erkennung von redundanten Code-Teilen, die Identifikation von Architekturabweichungen, die Suche nach fehlerträchtigen Code-Konstruktionen oder die einfache Ermittlung von Funktions- und Dateigrößen. Diese Analysen sind deutlich nachvollziehbarer für die Entwickler als der Einsatz von abstrakten Metriken. Somit lassen sich Problembereiche besser identifizieren und Gegenmaßnahmen viel einfacher ergreifen.
Bei den Werkzeugen ist es enorm wichtig, diese an den jeweiligen Projektkontext anzupassen und die „Falsch-Positiv“ Rate so gering wie möglich zu halten. Dadurch wird die Akzeptanz der Entwickler gegenüber dem Werkzeug gewahrt.
Die Operationalisierung der kontinuierlichen Qualitätsanalyse erfolgt durch sogenannte Software Quality Dashboards und in zeitlicher Abstand erstellten Qualitätsberichten. Die Dashboards haben die Aufgabe, die aggregierten Daten während der Analyse in Form von Übersichts- oder Trendgrafiken zu visualisieren. Dadurch verschaffen sie einen Überblick

über den aktuellen Qualitätszustand des Softwaresystems.
Die Qualitätsberichte dagegen dienen als eine Dokumentationsart und als Kommunikationsmittel zwischen dem Qualitätsingenieur und den Entwicklerteams [DH11].

4.2.1. Einsatzszenarien in der Praxis

Das Qualitäts-Controlling wird in der Praxis vor allem in zwei wesentlichen Szenarien eingesetzt. Zum einen dient es als Kontrollmechanismus für die Qualitätsäuberprüfung der von den externen Partner zugelieferten Softwarekomponenten, zum anderen zur Überprüfung und Verbesserung der eigenen Entwicklungs- und Wartungsqualität.
Bei dem ersten Szenario wird die Qualitätskontrolle durch den Auftraggeber durchgeführt. Die Qualitätskriterien, die hierbei geprüft werden, sind meist vertraglich vorgegeben. Die Qualitätsberichte der Zulieferer, die während des kontinuierlichen Qualitäts-Controllings erstellt wurden, könnten in diesem Fall als ein Teil der Abnahmekriterien angesehen werden.
Das andere Szenario ist die Qualitätssteuerung der eigenen Entwicklungs- oder Wartungsprojekte. Hierbei ist es empfehlenswert projektspezifische Qualitätsziele, wie z. B. die „Verbesserung der Wartbarkeit“ zu definieren und diese mit Hilfe eines Qualitätsmodells in mehrere konkretere Qualitätskriterien und Metriken herunter zu brechen.
Durch diese Aufteilung des Qualitätsmodells versucht man die Software-Qualität in einer strukturierten Art und Weise zu vereinfachen, um sie leichter messbar und bewertbar zu machen.

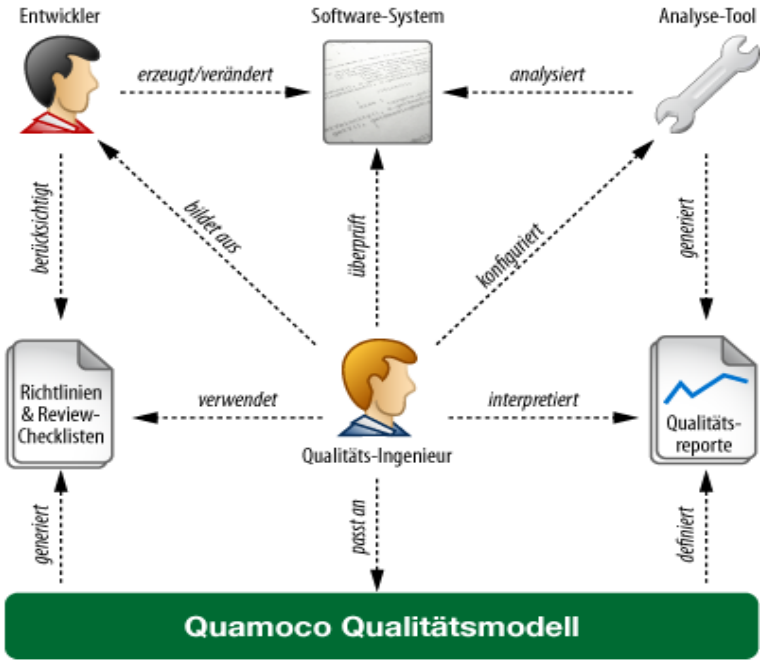


Abbildung 4.2.: Unternehmensinternes Qualiäts-Controlling (https://quamoco.in.tum.de)

Um so ein Qualitätsmodell in der Praxis zu operationalisieren, kann ein dafür geeignetes Werkzeug verwendet werden. Solch ein Werkzeug kann zur automatisierten Überprüfung des Qualitätsmodells eingesetzt werden und dadurch einen großen Analyseaufwand ersparen [DH11]. Die Abbildung 4.2 zeigt eine detailliertere Darstellung des unternehmensinternen Qualiäts-Controllings.
Hierbei wird die zentrale Rolle für die Sicherstellung der Qualität von einem Qualitätsingenieur übernommen. Dieser definiert im Abhängigkeit von dem Projektkontext ein Qualitätsmodell, welches die Qualitätsziele und die Qualitätsmerkmale, sowie die zu messenden Metriken beinhaltet. Dabei ist es wichtig, dass das Qualitätsmodell in maschinenlesbaren Form vorliegt, damit die Qualitätsüberprüfungen mit geringem Zeitaufwand durchgeführt werden können. Die Abbildung 4.3 zeigt ein aktivitäten-basiertes Qualitätsmodell mit dem Ziel eine hohe Wartbarkeit zu erreichen [WDFJ08]. Auf der einen Seite des Modells sind die Systemeigenschaften und -artefakte zu sehen, auf der anderen Seite hingegen die Wartungsaktivitäten. Diese, als Matrix dargestellte, Relation zeigt dann, wie sich die unterschiedlichen Systemartefakte auf die Wartungsaktivitäten auswirken [Dei10].

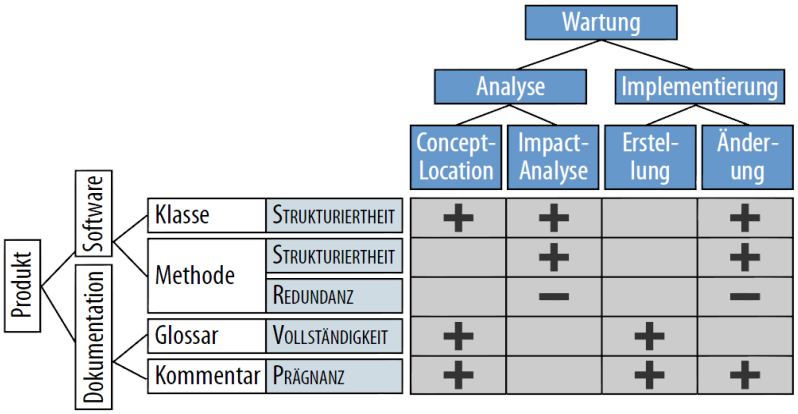


Abbildung 4.3.: Aktivitäten-basiertes Qualitätsmodell (F. Deißenböck)

Aus solch einem Qualitätsmodell werden anschließend Richtlinien und Checklisten generiert, die im Rahmen von manuellen Inspektionen und Reviews verwendet werden können. Weiterhin wird das Qualitätsmodell zur Anpassung des Analysewerkzeugs verwendet. Das Werkzeug stellt dann die Ergebnisse in Form von Quality Dashboards dar und unterstützt dadurch den Qualitätsverantwortlichen in seiner Arbeit [Dei10].

4.2.2. Vorschläge bei der Umsetzung

Experten wie Florian Deißenböck und Benjamin Hummel, die das kontinuierliche Qualitäts-Controlling in mehreren Unternehmen eingesetzt haben, machen auf einige wichtige Punkte bei der Umsetzung des Controllings aufmerksam [DH11].

- **Eine möglichst frühe Einbeziehung der Entwickler:**
  Um zusätzliche Aufwände zu vermeiden, die durch die Einführung des Qualitäts-

Controllings entstehen könnten, sollten die Entwickler so früh wie möglich in den neuen Controllingsprozess eingeführt werden. Außerdem sollte frühzeitig klar werden, wie die Messwerte zu bestimmen sind und wie die Ergebnisse den einzelnen Entwickler unterstützen sollen.

- **Eine effektive Mischung aus automatisierten Analysen und manuellen Inspektionen:**
  Um den Zeitaufwand für die kontinuierliche Qualitätsanalyse möglichst gering zu halten, sollte ein hoher Automatisierungsgrad durch den Einsatz eines Analysewerkzeugs gewährleistet werden. Weil aber eine vollständige Automatisierung weder sinnvoll noch möglich ist, sollte sie durch manuelle Inspektionen ergänzt werden. Erfahrungsgemäß kann man durch eine manuelle Inspektion viel effizienter die tiefer liegenden Probleme finden. Außerdem lässt sich die Erfahrung eines Entwicklers durch ein Werkzeug sehr schwer ersetzen.
- **Eine projektspezifische Anpassung des Kontrollwerkzeugs:**
  Die Qualitätskriterien hängen oft von dem Projektkontext ab. Daher sollte das Werkzeug mit den durch das Projekt gegebenen Qualitätsanforderungen mithalten können. Solch eine Anforderung könnte das Ausschließen von generiertem Code sein. Hierfür sollte die Möglichkeit gegeben sein, das Qualitätsanalysewerkzeug dementsprechend anzupassen.

4.3. Werkzeugunterstützung

Um dem Software Qualitätsverfall entgegenzuwirken, werden oft hunderte von Qualitätskriterien definiert, die kontinuierlich überprüft werden müssen. Viele von diesen Regeln lassen sich durch automatisierte Qualitäsanalysewerkzeuge überprüfen. Beispiele hierfür sind die Ermitllung von Architekturabweichungen, die Erhebung von Softwaremetriken oder das Auffinden von Copy&Paste Codestellen.
Um eine Qualitäsanalyse möglichts effizient und ohne Produktivitätssenkungen durchführen zu können, müssen aber alle hierfür geeigneten Werkzeuge bestimmten Anforderungen gerecht werden.

4.3.1. Anforderungen an das Qualitätsübewachungswerkzeug

Es existieren viele kommerzielle und akademische Werkzeuge zur Softwarequalitätsanalyse. Aufgrund der Vielfalt der Qualitätskriterien ist eine Analyse mit nur sehr großem Aufwand durchzuführen. Ein weiterer Grund dafür ist die enorme Datenmenge, die so aggregiert und visualisiert werden muss, dass sie effizient und effektiv während der kontinuierlichen Qualitätsüberprüfung analysiert werden kann.
Damit der Analyseaufwand nicht zum Verhängnis wird, müssen die eingesetzten Analysewerkzeuge gewisse Kriterien erfüllen [Dei09]. Als nächstes werden einige der wichtigsten Anforderungen an ein Qualitätsanalysewerkzeug kurz erläutert.

4.3.1.1. Integrationsfähigkeit

Um einen detaillierten und besseren Überblick über den aktuellen Qualitätszustand eines Softwaresystems bieten zu können, sollte ein „Software Quality Dashboard“ in der Lage sein, gleichzeitig mehrere andere automatische Analysewerkzeuge einzubinden. Zudem sollte auch die Integration der Ergebnisse einer manuellen Inspektion möglich sein

4.3.1.2. Autonomer Betrieb

Um zusätzliche Kosten und zusätzlichen Zeitaufwand für die Qualitätsanalyse zu vermeiden, sollte die Auswertung der Analyseergebnisse autonom ablaufen. Dafür sollte es möglich sein das Softwareanalysewerkzeug in die Build-Prozesse zu integrieren, damit es bei jedem Build-Vorgang automatisch ausgeführt werden kann.

4.3.1.3. Vielfältigkeit

Abhängig von dem Projekt werden unterschiedliche Qualitätskriterien definiert, die verschiedene Softwareartefakte, wie z. B. Quellcode, Build-Skripte oder Dokumentation, betreffen. Daher sollte sich die Analysemöglichkeit eines Werkzeugs nicht nur auf bestimmte Artefakttypen beschränken.

4.3.1.4. Flexibilität

Das Werkzeug sollte in verschiedenen Projekten leicht einsetzbar sein. Dazu muss es flexibel genug sein, um den sich ständig ändernden Anforderungen gerecht zu werden und aus den Resultaten der verschiedenen Analysen ein Gesamtbild zu erstellen.

4.3.1.5. Aggregation und Visualisierung

Ein Werkzeug sollte in der Lage sein, die Analyseergebnisse so zu visualisieren, dass sie in kurzer Zeit überprüft werden können. Dabei sollte auch die Möglichkeit bestehen, ein Qualitätsdefizit im Detail zu betrachten. Daher sollten die Software Quality Dashboards leistungsfähige Mechanismen zur Visualisierung, wie z. B. Trends oder Treemaps, anbieten.

4.3.1.6. Erweiterbarkeit

Es gibt kaum ein Werkzeug, das die Analyse aller existierenden Softwareartefakte unterstützt. Daher ist es wichtig, dass das Analysewerkzeug einen Mechanismus anbietet, mit dem die Anwender bei Bedarf weitere Analysen und Bewertungen hinzufügen können. Darüber hinaus sollte auch die Möglichkeit bestehen die Funktionalität des Werkzeugs anzupassen oder zu erweitern.

4.3.1.7. Performance

Häufig ist es der Fall, dass Softwaresysteme mit mehreren Millionen Zeilen Code analysiert werden müssen. Damit die Performance hierbei keine Einschränkung darstellt, sollte ein Analysewerkzeug fähig sein, die Qualitätsanalyse in akzeptabler Zeit durchzuführen.

4.3.2. Analyse Werkzeuge

Es gibt eine Reihe von Werkzeugen, die dafür gedacht sind die Qualität der Software zu analysieren. Die bekanntesten Vertreter hierfür sind der Sotograph[1], Sonar[2] und iPlasma[3]. Eine kontinuierliche Qualitätsanalyse lässt sich aber durch diese Werkzeuge nur bedingt bewerkstelligen, da sie nur einen interaktiven Charakter haben [DS06]. Zudem existieren auch leistungsfähige Werkzeuge, wie JDepend[4], CheckStyle[5] und PMD[6], die aber nur in einem bestimmten Analysegebiet einsetzbar sind. Um eine hochwertige Qualitätsanalyse durchzuführen, sollte die Möglichkeit bestehen alle diese Werkzeuge zu integrieren und die Analyse zu einem späteren Zeitpunkt möglichst einfach zu verfeinern. Aus diesen Gründen wurde für die kontinuierliche Qualitätsanalyse im Rahmen der Diplomarbeit bei „XYZ“ AG das Qualitätsanalysewerkzeug ConQAT eingesetzt.

4.4. Manuelle Inspektionen

Um eine erfolgreiche Qualitätsanalyse durchführen zu können, besteht zusätzlich die Notwendigkeit manueller Inspektionen, die aus mehreren Gründen enorm wichtig für eine gut durchdachte Qualitätssicherung sind.
Zum Einen lassen sich durch maschinelle Verfahren nur ein begrenzter Teil der gesamten Qualitätskriterien überprüfen. Die tiefer liegenden Probleme können aber nur durch manuelle Überprüfungen identifiziert werden. Zum Anderen können die Kreativität und die Erfahrung der Entwickler durch automatisierte Analysen nicht ersetzt werden [DH11]. Dabei ist die Art der durchzuführenden manuellen Inspektionen dem Unternehmen oder dem Entwicklerteam überlassen. Es könnte sich hierbei, abhängig vom Projekt, um Inspektionen des Quellcodes während der Entstehung (Pair Programming), oder um umfangreiche Code Reviews handeln.
Für den optimalen Erfolg der kontinuierlichen Qualitätsanalyse ist es von enormer Bedeutung, dass sich die automatisierten Analysen und die manuellen Inspektionen während der Entwicklungs- und Wartungsprozesse ergänzen. Dadurch kann ein vollständiges Bild vom Qualitätszustand des Softwaresystems gewonnen werden, um bei Bedarf problemspezifische

[1] http://www.hello2morrow.com/products/sotograph/
[2] http://www.sonarsource.org/
[3] http://loose.upt.ro/reengineering/research/iplasma
[4] http://www.clarkware.com/software/JDepend.html
[5] http://checkstyle.sourceforge.net/
[6] http://pmd.sourceforge.net

Maßnahmen ergreifen zu können. Diese Diplomarbeit behandelt hauptsächlich die werkzeugunterstützte Softwarequalitätsanalyse, daher wird auf eine weitere Beschreibung der manuellen Inspektionen verzichtet.

5. ConQAT

In diesem Kapitel wird das „Continuous Quality Assessment Toolkit“ (ConQAT) im Detail betrachtet, sowie einiger seiner Analysemöglichkeiten.

5.1. Continuous Quality Assessment Toolkit (ConQAT)

Alle oben erwähnten Anforderungen werden von dem Qualitätsanalysewerkzeug „ConQAT“ erfüllt. Das Toolkit ist an der Technischen Universität München entwickelt worden.
ConQAT[1] steht für Continuous Quality Assessment Toolkit. Das Werkzeug unterstützt die kontinuierliche Qualitätskontrolle bei der Entwicklung und der Wartung von Softwaresystemen. ConQAT bietet die Möglichkeit, Quellcode zu analysieren, der in verschiedenen Programmiersprachen geschrieben sein kann. Zu den unterstützten Sprachen zählen **JAVA, ABAP, ADA, C#, C/C++, COBOL, Visual Basic** und **PL/SQL**.
ConQAT ist plattformunabhängig und steht unter der Apache License 2.0 [DFH+10]. Das Toolkit ist in der Programmiersprache **JAVA** als modulares und leicht erweiterbares System implementiert worden. Es unterstützt den Aufbau von vorgegebenen oder selbst erstellten Komponenten (ConQAT-Blöcke), die zur Ermittlung, Aggregation, Historisierung und Visualisierung von Metriken und Messergebnissen dienen. Die Flexibilität und die umfangreiche Funktionalität von ConQAT erlaubt außerdem eine Beschreibung und Verfolgung von einem vorgegebenen oder selbst definierten Qualitätsmodell.
Darüber hinaus bietet das Werkzeug die Integration bekannter leistungsfähiger Werkzeuge wie PMD, FindBugs, JUnit oder des Test-Abdeckungs-Werkzeugs Cobertura [DS06]. Zudem können auch die Reports von dem statischen Analysewerkzeug PC-LINT (C/C++) ausgelesen und in einer einheitlichen Form dargestellt werden.
Die Ergebnisse von ConQAT werden in Form von XML-Dateien und HTML-Seiten ausgegeben. Dabei besteht auch die Möglichkeit die Resultate als Graphiken oder Trends darzustellen. Dadurch können die für die Qualitätssicherung verantwortlichen Personen unterstützt werden und den Überblick über den aktuellen Qualitätszustand eines Softwaresystems behalten [Dei10].

[1] http://conqat.in.tum.de

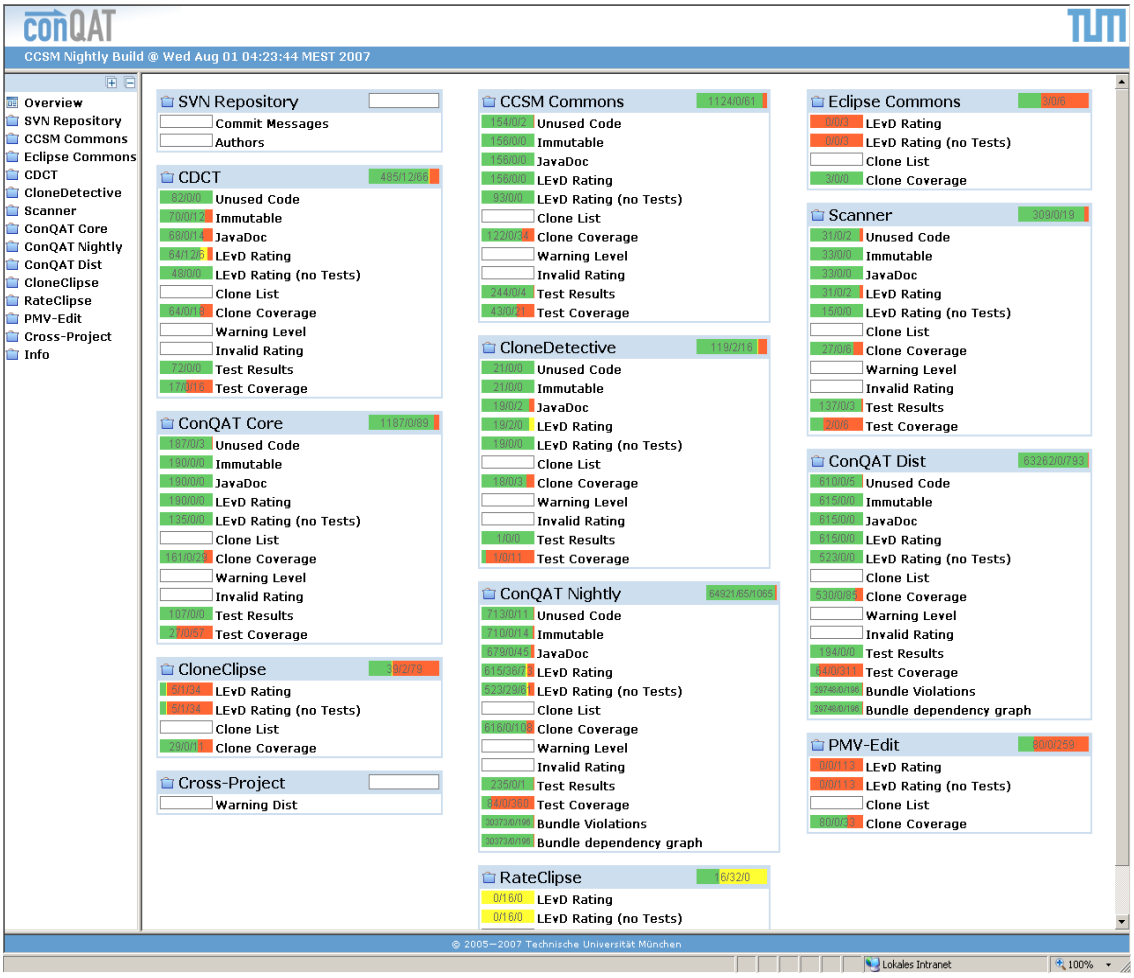


Abbildung 5.1.: Beispiel für ConQAT Nightly-Build (conqat.in.tum.de)

5.1.1. Design und Architektur

ConQAT erreicht die weiter oben geforderte Flexibilität und Erweiterbarkeit durch eine Plug-in-ähnliche Schnittstellenarchitektur. Der Werkzeugkasten ist in Anlehnung an das Pipes&Filters-Konzept implementiert worden. Dadurch wird das Hinzufügen und das Entfernen von Analyse-Modulen enorm erleichtert und der Typ der verarbeiteten Informationen wird nicht eingeschränkt. Die wichtigsten Komponenten der ConQAT-Architektur sind die Prozessoren.
Bei den Prozessoren handelt es sich um in Java implementierte Module, welche die eigentliche Analyse realisieren. Diese Analysemodule arbeiten wie Funktionen, welche mehrere Dateneingaben verarbeiten und nur eine einzelne Datenausgabe produzieren. Der Datenfluss zwischen den einzelnen Prozessoren kann als ein gerichteter Graph angesehen werden.
ConQAT bietet eine Bibliothek mit mehr als 300 Prozessoren mit unterschiedlichen Analysemöglichkeit.

Der Verarbeitungsablauf der Daten durch die Prozessoren wird über Konfigurationsdateien angegeben. Diese Dateien werden von dem ConQAT-Driver ausgelesen, der die zentrale Instanz in der ConQAT-Architektur darstellt. Der Driver (ConQAT-Engine) instanziiert das per XML-Datei beschriebene Prozessornetzwerk und kümmert sich darum, die Prozessoren richtig zu verbinden [DJH+08]. Der rechte Teil der Abbildung 5.2 gibt einen Überblick über die Architektur von ConQAT.

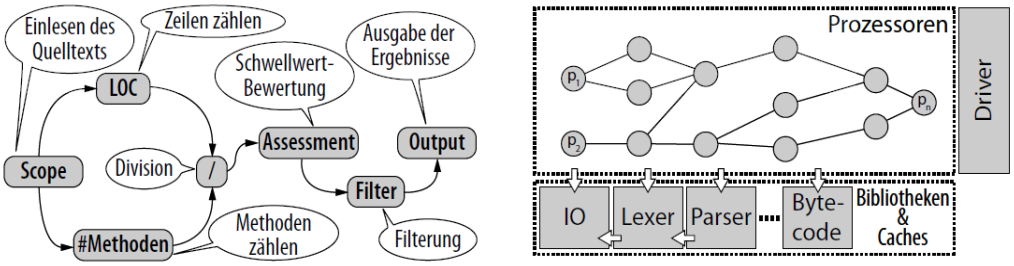


Abbildung 5.2.: Beispielkonfiguration und ConQAT-Architektur (F.Deißenböck)

Um ganze Arbeitsabläufe abbilden und wiederverwenden zu können, bietet ConQAT die Möglichkeit, die semantisch zusammengehörige Prozessoren zu Blöcken zusammenzufassen. Der linke Teil der Abbildung 5.2 zeigt wie einfache Prozessoren zu Blöcken kombiniert werden können. ConQAT enthält bereits vordefinierte Blöcke, so dass mit wenig Aufwand komplexe Analysen durchgeführt werden können. Diese Architekturkonstruktion bietet einen hohen Wiederverwendungsgrad und ermöglicht die Analyse von Bytecode-, Dokumentation-, Modellspezifikation- und Quellcodedaten unterschiedlicher Projekte.
Durch diesen Ansatz bereiten auch die umfangreicheren Analysen kein Problem. Um zu verhindern, dass ConQAT die rechenintensiven Aufgaben, wie das Parsen des Quelltextes, nicht mehrfach ausführt, wird eine intelligente Hierarchie von dynamischen Caches verwendet. Dadurch werden beim Einsatz im Nightly Build auch die umfassenden Analysen in akzeptabler Zeit durchgeführt [DS06].

5.1.2. Funktionalität

ConQAT bietet eine Reihe von Analysemöglichkeiten sowohl auf Architekturebene, als auch auf Code Level. Als nächstes werden einige der wichtigsten ConQAT-Funktionalitäten vorgestellt.

5.1.2.1. Architecture Conformance Analysis

Die Architektur einer Software gibt vor, aus welchen Komponenten ein Softwaresystem aufgebaut ist. Zudem legt sie fest, wie die Komponenten miteinander in Beziehung stehen sollen. Die einzelnen Systemkomponenten können über öffentlichen Schnittstellen mit anderen

Komponenten interagieren. Daher bestimmt die Architektur auch welche Abhängigkeiten zwischen den einzelnen Bestandteilen eines Softwaresystems nicht erlaubt sind, da sie die Wartbarkeit, die Flexibilität und die Erweiterbarkeit des Systems negativ beeinträchtigt würden. Die gute Architektur einer Software ist von enormer Bedeutung für ihre spätere Wartbarkeit und Weiterentwicklungsfähigkeit und bringt klare Vorteile, wie z. B. besser Wartbarkeit und Weiterentwicklungsfähigkeit mit sich.
Die Architektur hilft kleine Komponenten zu entwickeln, die sich leichter lenken und verarbeiten lassen als die großen monolithischen Bestandteile. Außerdem können alle Komponenten unabhängig voneinander konstruiert werden, was die Ersetzung und die Weiterentwicklung einzelner Bauteile enorm erleichtert. Trotz der Vorteiler einer klar definierten Architektur wird sie in vielen Unternehmen nicht beachtet und kontrolliert.
ConQAT bietet die Möglichkeit, eine Architekturkonformitätsanalyse durchführen zu können. Dadurch können die Architekturabweichungen und -verletzungen festgestellt und problemspezifische Maßnahmen ergriffen werden.

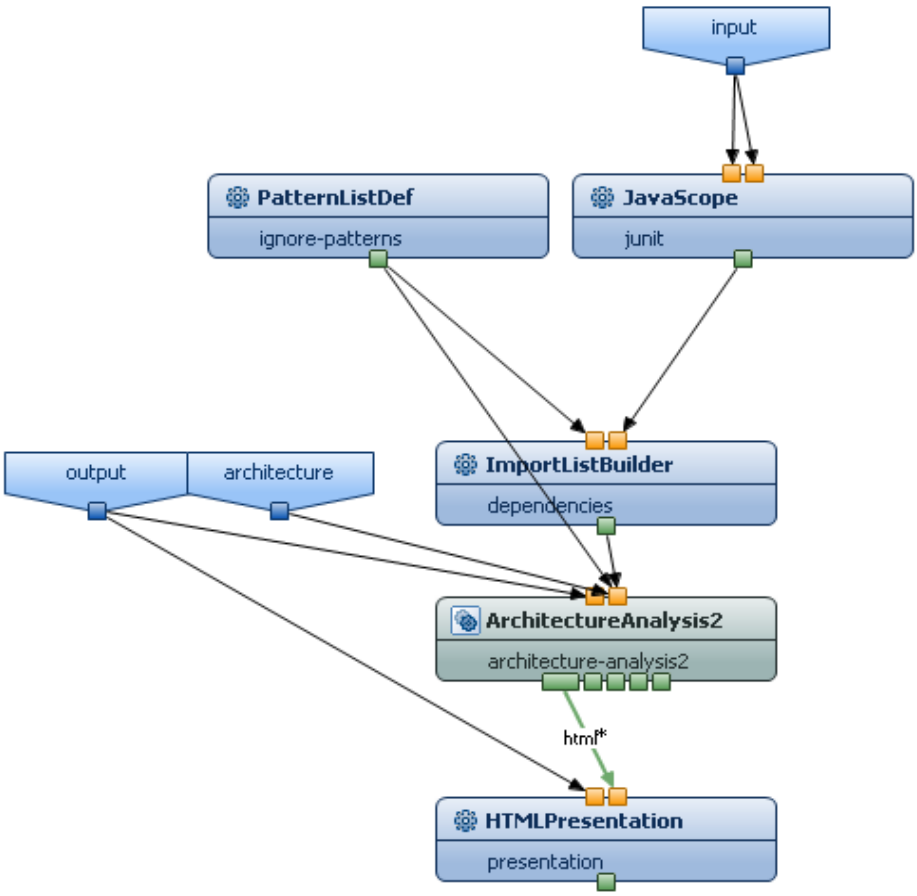


Abbildung 5.3.: ConQAT-Architekturanalyse-Block (conqat.in.tum.de)

Unter einer Architekturabweichung wird das Divergieren zwischen dem Ist- und dem Soll Zustand verstanden. Die Divergenz zu kennen ist aus mehreren Gründen wichtig.

Erstens können dadurch die Stellen ermittelt werden, an denen man arbeiten soll, um die Innere Qualität des System zu verbessern. Zweitens könnten die Divergenz-Kenntnisse hilfreich sein, wenn man Entwicklungstätigkeiten in Problemzonen, wie z. B. Bereichen mit Abhängigkeitszyklen, durchführen möchte. Im Allgemein könnten durch die Divergenz-Kenntnisse Aussagen über die Wartbarkeit des Softwaresystems getroffen werden. Der Ist-Zustand (Java, C#) kann bei ConQAT beispielsweise durch den Quellcode geparst werden. Hierfür muss ein Analyse-Block erstellt werden, damit der für den Ist-Zustand nötige Quellcode ausgelesen werden kann. Die Abbildung 5.3 zeigt ein Beispiel für solch einen Block.
Um den Soll-Zustand durch ConQAT zu ermitteln, müssen Konfigurationsdateien angelegt werden. Hier wird beschrieben, wie die einzelnen Bestandteile (Packages und Dateien) zueinander in Beziehung stehen. ConQAT ermöglicht mittels einem Editor (auf Eclipse-Basis) den Soll-Zustand als Graphen darzustellen. Dabei ist es möglich die Beziehungen zwischen den einzelnen Komponenten zu erlauben, tolerieren oder verbieten. Das Bild 5.4 stellt einen Beispielgraph des Soll-Zustands dar. Die gelben, gerichteten Linien stellen die tolerierbaren, die Grünen die erlaubten und die Roten die verbotenen Beziehungen dar.

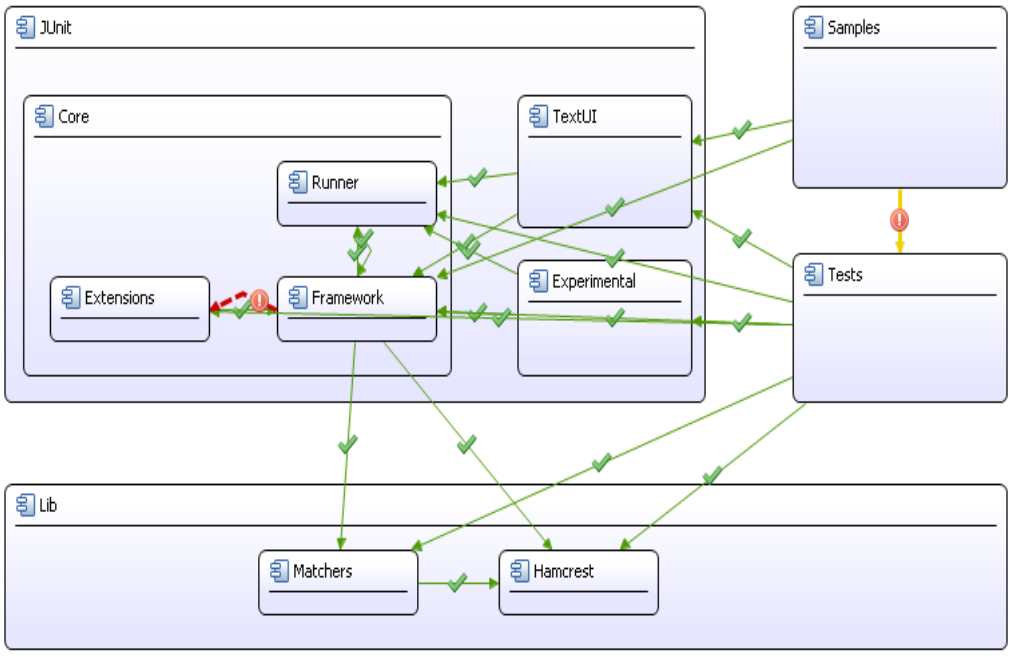


Abbildung 5.4.: Beispielgraph Soll-Zustand (conqat.in.tum.de)

Die erstellten Ist- und Soll-Zustandsgraphen können anschließend miteinander verglichen werden. Die gefundenen Architekturverletzungen werden dem Benutzer in Form von Graphen dargestellt. Dabei ist zu beachten, dass nicht jeder gefundene Fehler eine Architekturabweichung sein muss. Es ist möglich, dass die Architekturspezifikation nicht stimmt und nachgebessert werden muss.

5.1.2.2. Clone Detection

Mit Codeduplizierung wird in der Programmierung die unnötige und mehrfache Wiederholung des gleichen oder ähnlichen Sourcecodes im Quelltext bezeichnet. Verschiedene wissenschaftliche Studien zeigen, dass ein beachtlicher Teil (5-10%) des Programmcodes aus dupliziertem Code besteht [Bak95] [JDHW09] [Sch07b]. Tatsächlich könnten aber die duplizierten Stellen im Quellcode bis zu 50% oder sogar mehr sein [Jür12].
Die Codewiederholungen wirken sich aber negativ auf mehrere innere Qualitätsmerkmalen eines Softwaresystems aus. Die Wartbarkeit, die Testbarkeit, die Lesbarkeit oder auch die Erweiterbarkeit eines Systems leiden unter dem Vorkommen von Code-Duplikaten. Zudem erhöht sich auch die Größe und die Komplexität der Software unnötig. Das Ganze hat zur Folge, dass die innere Qualität des Softwaresystems mit der Zeit verfällt. Als nächstes werden einige der negativen Auswirkungen der Codeduplizierung vorgestellt:

- Die Größe des Sourcecodes wächst unnötig und damit kommt es zu einer Komplexitätssteigerung.
- Durch das Kopieren von Codefragmenten werden Fehler verschleppt.
- Die Wartungskosten werden negativ beeinflusst, da die Änderungen wie Bugfixes in allen duplizierten Stellen durchgeführt werden müssen. Code-Duplikate, die während der Änderungen vergessen worden sind, könnten zu Nebeneffekten führen. Fehler die als behoben angenommen worden sind, könnten wieder an anderen Stellen im Quellcode auftauchen [JDHW09].
- Der Aufwand für die Testbarkeit erhöht sich enorm, da jede Codeduplizierung separat getestet werden muss. Wie bei den Änderungen könnte es aber auch hier der Fall sein, dass nicht alle duplizierten Stellen durch die Tests abgedeckt werden.
- Die duplizierten Codestellen könnten zu „totem Code“ führen und damit die Lesbarkeit, die Verständlichkeit und die Wartbarkeit erschweren [Sch07b].
- Teile des duplizierten Code könnten in der neuen Stelle zum „semantischen“ Fehler führen, da sie dort ungeeignet sein könnten [Sch07b].

Die Gründe für das Auftreten der Code-Duplikate können unterschiedlich sein. Sie sind meistens mit fehlender Zeit im Projekt und wenig Personalressourcen verbunden. Das fehlende Verständnis der Programmierer für ein Problem oder die Vermeidung der Fehler (wenn der Programmierer unzureichendes Wissen über den Code hat) führt ebenfalls zu Codeduplizierungen. Besonders bei dem Einsatz von Programmiersprachen, die keinen Mechanismus zur Wiederverwendung besitzen, ist dieses Phänomen sehr oft zu beobachten [Sch07b].
Das Toolkit ConQAT bietet die Möglichkeit zur Suche nach dupliziertem Code, durch die Anwendung komplexer Algorithmen und integrierten Funktionalitäten. Das Werkzeug kann unter anderem Text-Dokumente und Sourcecode, der in ABAP, ADA, C, C/C++, Cobol, Java, Visual Basic, PL1 and PL/SQL geschrieben ist, nach Code-Duplikaten durchsuchen. Für

alle anderen Programmiersprachen besteht die Möglichkeit, eine Duplikatenanalyse durchzuführen, in dem der Quellcode als einfaches Text-Dokument behandelt wird [DFH+10].

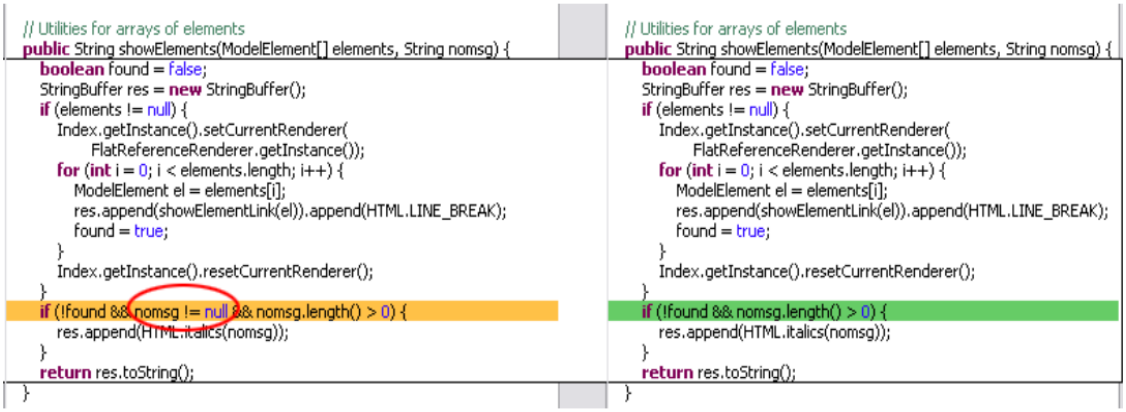


Abbildung 5.5.: Beispiel für Codeduplizierung (ConQAT-Buch)

ConQAT kann sowohl syntaktisch (**Ungapped clones**) als auch semantisch (**Gapped clones**) gleiche Quellcodestellen analysieren, finden und dementsprechend darstellen. Die Metriken, die ConQAT bei der Duplikatensuche berechnen kann sind folgende (aus dem ConQAT-Buch [DFH+10]):

- **LoC** Lines of code.
- **Clone LoC** Cloned lines of code.
- **Clone Count** Number of clones.
- **Units** Number of units contained in the analyzed files. Typically, units are statements.
- **Clone Units** Cloned units, i.e., cloned statements.
- **UnitCoverage** Probability that an arbitrarily chosen statement is part of a clone.
- **RFSS** Redundancy free source statements. That is, the number of statements from a hypothetical system from which all cloning is perfectly removed.

Dabei ist zu beachten, dass die Werte der Metriken (Clone LoC, Clone Count, Clone Units, UnitCoverage) so niedrig wie möglich gehalten werden sollten. Eine Ausnahme hierzu ist die RFSS Metrik. Bei dieser sollten die Werte so hoch wie möglich sein.
ConQAT erlaubt es die minimale Länge der zu untersuchenden Code-Duplikate optional zu definieren. Außerdem bietet ConQAT die Möglichkeit Code-Duplikate, die explizit gewollt sind, durch den Prozessor „BlackListFilter“ von einer weiteren Analyse auszuschließen. Hierfür ist der Eclipse-Editor notwendig.
Mit der Option „Rejected“ können alle gewollten Code-Duplikate und „False Positive“ markiert werden. Im Hintergrund findet dann eine Berechnung der Hashwerte der markierten Stellen statt. Mit Hilfe dieser Hashwerte werden die Codewiederholungen von der Analyse

ausgeschlossen. Weiterhin besteht ebenfalls die Möglichkeit über reguläre Ausdrücke gewollten Codeduplizierung zu ignorieren.
Die nach der Analyse gefundenen Duplikate werden im Anschluss in Form von Treemaps (siehe Punkt 5.1.2.3) und Tabellen dargestellt. Zudem können die Ergebnisse als XML-Dateien gespeichert und von jedem beliebigen Entwickler in dem Unternehmen betrachtet werden. Voraussetzung hierfür ist das ConQAT im Vorfeld installiert wurde. Über den Eclipse-Editor können letztendlich die gefundenen Duplikate gegenübergestellt werden. Mit Hilfe des Editors wird auch ersichtlich, über wie viele Instanzen (Dateien) sich ein Code-Duplikat verstreut.

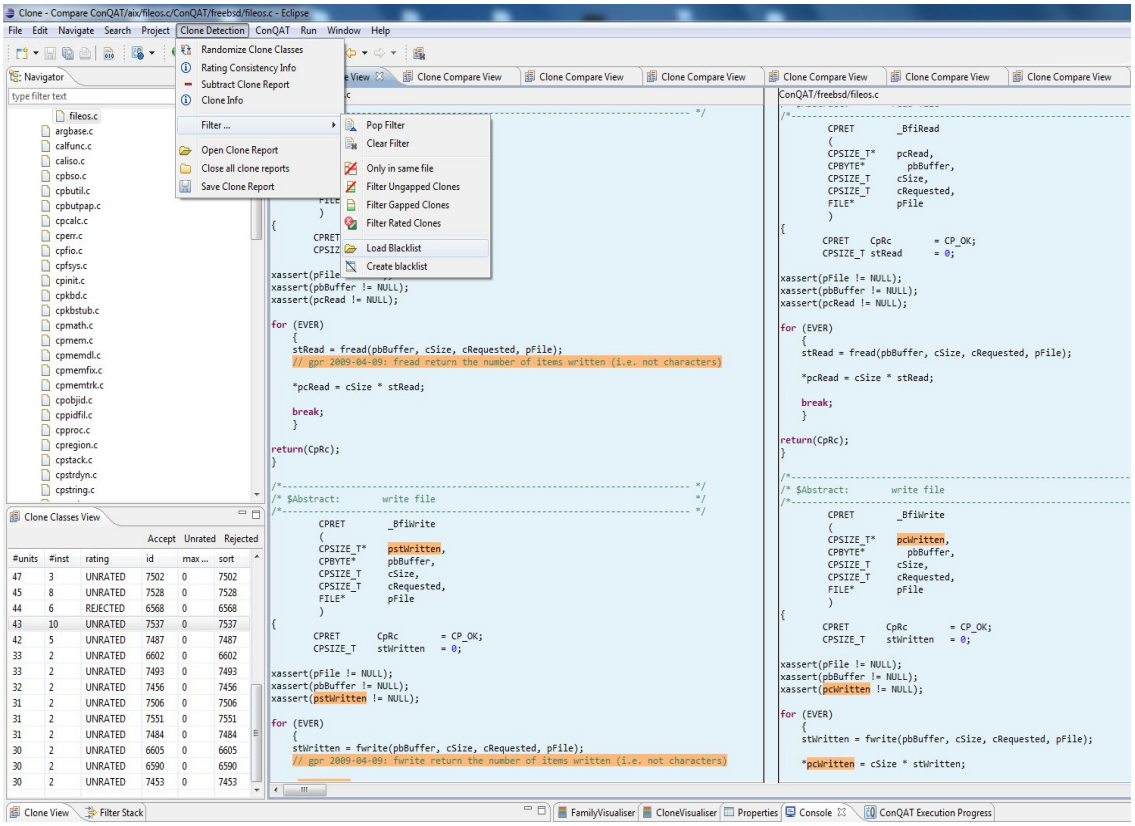


Abbildung 5.6.: Eclipse-Editor Ansicht

5.1.2.3. Visualisierungsmöglichkeiten

Die Visualisierung der Analyseergebnisse ist einer der wichtigsten Schritte des kontinuierlichen Qualitäts-Controllings. ConQAT ermöglicht die Darstellung der Resultate durch dafür bereitgestellten Prozessoren und Blöcke. Eine der am häufigsten verwendeten Visualisierungen bei diesem Analysewerkzeug ist die **Html Darstellung**. Durch diese Visualisierungsmethode wird ein Html Dashboard erzeugt, das eine aggregierte Sicht auf alle Analyseergebnisse ermöglicht (siehe Bild 5.1). ConQAT bietet zudem die Möglichkeit, die

Resultate in Form von Ampelbewertungen, Übersichtsgrafiken (Treemaps) oder Trends darzustellen. Die Übersichtsgrafiken werden auch **Treemaps** (siehe Bild 5.7) genannt. Sie dienen der Visualisierung hierarchischer Baumstrukturen. Die Darstellung dieser Strukturen erfolgt in Form von ineinander verschachtelten Rechtecke. Dabei entspricht jede einzelne Datei des Softwaresystems einem Rechteck. Die Größe des Rechtecks stellt hierbei die Größe der dementsprechenden Datei dar. Zudem ist es auch ersichtlich, welcher dieser Rechtecke einen Ordner, Unterordner oder ein Package darstellen und welche Probleme sich innerhalb eines Rechtecks befinden.
Die Gewichtung der Aussagen wird nach dem „Ampel-Prinzip“ (über Farben) bestimmt. Die rote Farbe signalisiert, dass an einer bestimmten Stelle dringender Handlungsbedarf besteht. Die gelbe Farbe steht für einen tolerierbaren Bereich. Die grüne Farbe zeigt, dass sich der jeweilige Bereich in einem optimalen Zustand befindet. ConQAT ermöglicht die Treemap-Darstellung z. B. bei der Clone-, Comment-Ratio oder Task Tags-Analysen. Außerdem besteht auch die Möglichkeit die Kommentar-Sprachen (Deutsch, Englisch usw.) ebenfalls als Treemaps darzustellen.

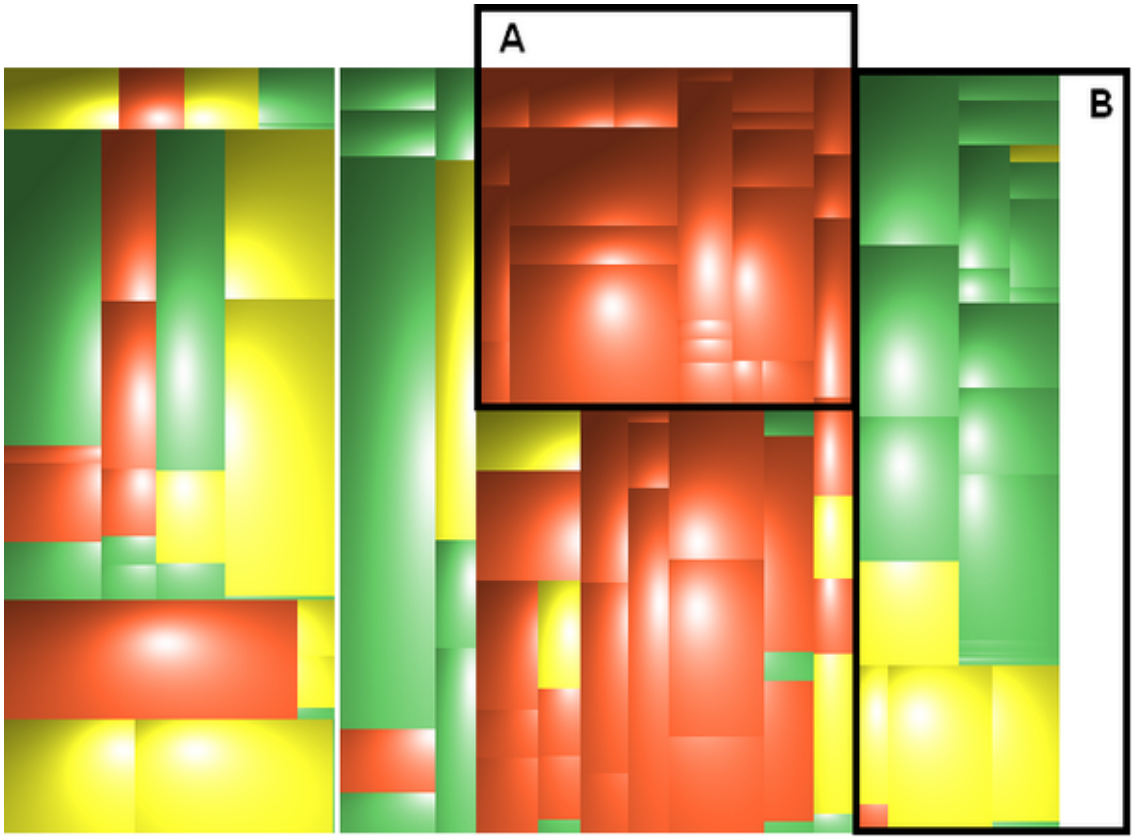


Abbildung 5.7.: ConQAT-Treemap Ansicht (conqat.in.tum.de)

In vielen Fällen ist es sinnvoll nicht nur bestimmte Qualitätsdefizite zu analysieren, sondern auch ihre Entwicklung über die Zeit zu verfolgen. Für die Projektleitung oder das Management könnte z. B. interessant sein, ob sich die Clone-Rate über die Zeit erhöht. Für die

Entwickler hingegen könnten die erreichten positiven Qualitätsergebnisse als Motivation für weitere Verbesserungen dienen.
Um dieser Anforderung gerecht zu werden, bietet ConQAT die Möglichkeit mit Hilfe von Trendanalysen die positiven und negativen Entwicklungen bestimmter Qualitätskriterien zu visualisieren (siehe Bild 5.8).
Der einfachste Weg zur Historisierung der zu visualisierenden Daten ist ihre Speicherung in einer Datenbank. ConQAT unterstützt eine Datenbankverbindung mithilfe der **JDBC**[2]-Datenbankschnittstelle.
Das Werkzeug verfügt über die Prozessoren **HSQLDBDatabaseConnector** und **SQLServerConnection**, die zur Herstellung einer Verbindung zu der mit ConQAT mitgelieferten **HSQLDB**[3] oder zu einem externen **Microsoft SQL Server**[4] verhelfen. Zudem unterstützt ConQAT die Verbindung zu anderen JDBC fähigen Datenbanksystemen durch den Prozessor **JDBCDatabaseConnector**.

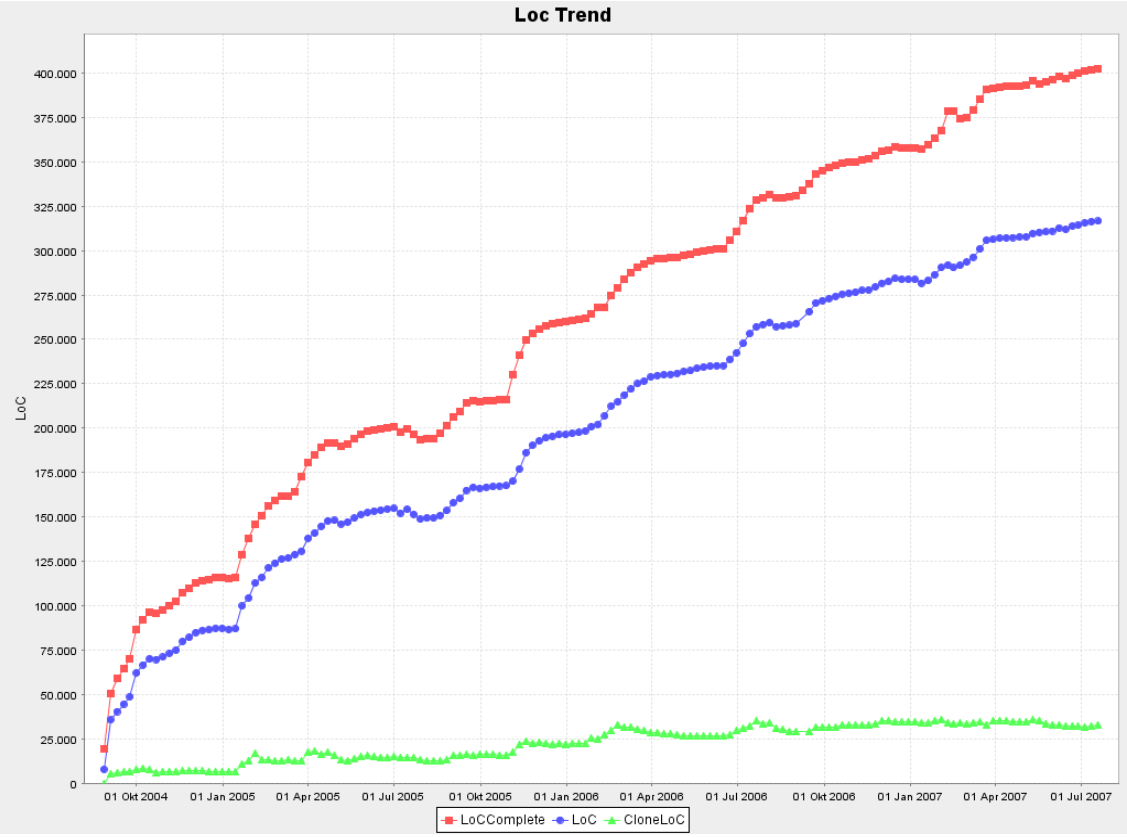


Abbildung 5.8.: ConQAT- LOC Trend (conqat.in.tum.de)

[2]Java Database Connectivity
[3]http://hsqldb.org/
[4]http://www.microsoft.com/sqlserver/en/us/default.aspx

5.2. Installation und Ausführung

Für die Installation des Werkzeugs ist die Java-Version **1.6** oder höher erforderlich. ConQAT läuft auf allen Plattformen (**Windows, Mac OS X, Linux GTK 64bit**), die diese Java-Version unterstützen. Um neue Blöcke auf Basis bereits bestehender Blöcke oder Prozessoren bauen zu können, benötigt ConQAT ein Installationspaket, das eine Eclipse-Version 3.5 und alle vorhandenen ConQAT-Bibliotheken beinhaltet. Das Installationspaket kann an jedem beliebigen Ort (Verzeichnis) auf dem Rechner gespeichert und entpackt werden.
Zudem ist es auch möglich ConQAT in einem bereits verfügbares Eclipse[5] zu integrieren. Dafür ist es allerdings eine Eclipse-Version 3.7 oder höher notwendig. Die Installation lässt sich am einfachsten über den Eclipse Update-Manager (**Help->Install New Software**) mit der Adresse: **„http://www4.in.tum.de/~ccsm/conqat_ update_ site/“** durchführen. Nach der Durchführung des Updates ist es notwendig die ConQAT-Engine zu installieren, die alle benötigten Bibliotheken beinhaltet. Dafür muss der Pfad zum Ordner „bundles“ von der entpackten Engine-Datei in Eclipse unter (**Window ->Preferences ->ConQAT**) angegeben werden.

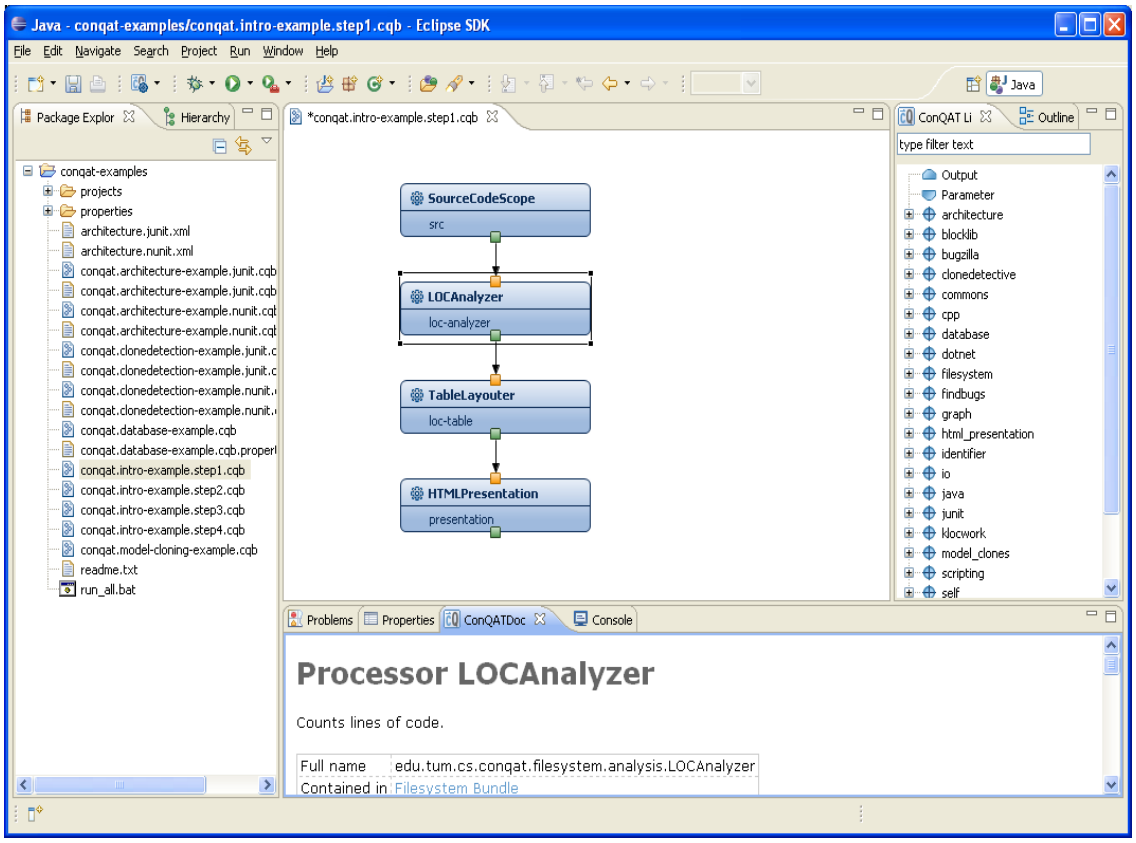


Abbildung 5.9.: Eclipse-Workbench (conqat.in.tum.de)

[5]http://www.eclipse.org/

Einige ConQAT-Analysen erfordern außerdem das Program „dot“ aus dem GraphViz-Paket[6] oder das Microsoft .NET Framework 2.0[7].
ConQAT kann sowohl in 32-Bit- als auch in 64-Bit-Systemen ausgeführt werden. Die Bedienung des Werkzeugs erfolgt über die Eclipse-GUI (siehe Bild 5.9).
Vor Ausführung einer Analyse muss als erstes ein **ConQAT-Block** erstellt werden. Ein Block ist in diesem Fall eine XML-Datei, welche die Ausführungsreihenfolge und die Abhängigkeiten der sogenannte **Prozessoren** festlegt. Die Prozessoren sind Java-Klassen, welche die eigentliche Analyse durchführen. Mit Hilfe der Eclipse-Editor können diese Prozessoren mit gerichteten Verbindungen zu einem Block verknüpft werden. Die Abbildung 5.9 zeigt ein Beispiel für einen Block. Dabei stellt jedes der einzelnen Blaurechtecke einen Prozessor dar. Alle vier Prozessoren bilden zusammen einen Block. Zudem verfügt jeder der Prozessoren über mehrere Parameter, die in einer „Property-View“ eingestellt werden können (siehe Bild 5.10).

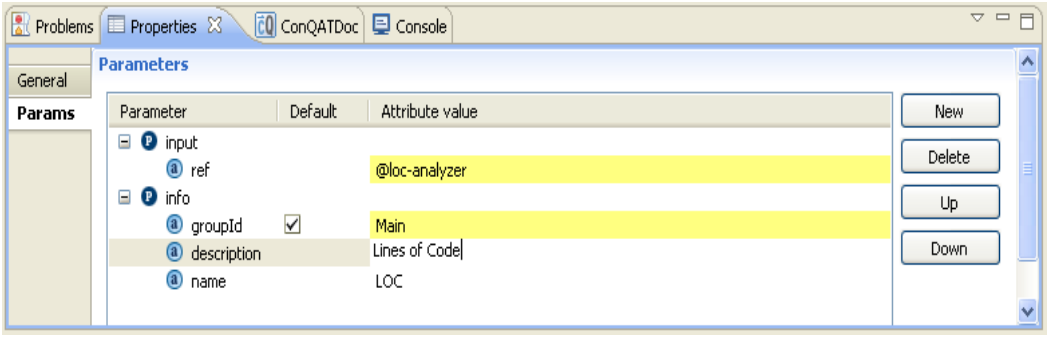


Abbildung 5.10.: Property-View (conqat.in.tum.de)

Zum Ausführung der tatsächlichen Analyse muss zu jedem Block eine sogenannte **Run Configuration** erstellt werden. Dabei ist zu achten, dass eine Run Configuration zu genau einem ConQAT-Block erstellt werden kann (siehe Bild 5.11). Mit dem Button **Launch ConQAT Analysis** kann dann die Eclipse-basierte Analyse gestartet werden. Dabei werden die Ergebnisse in einem im Vorfeld als Parameter angegebenen Ordner gespeichert.
ConQAT bietet die Möglichkeit eine Analyse ohne den Eclipse-Editor auszuführen. Hierfür ist die Installation der ConQAT-Engine ausreichend. Hierbei ist es nicht möglich neue Blöcke zu erstellen, sondern nur die bestehenden Blöcke oder Run Configurations auszuführen. Für diese Analyseart stellt ConQAT eine Stapelverarbeitungsdatei (**conqat.bat** für Windows und **conqat.sh** für Linux) zu verfügung. Die Analyse wird dann über die Kommandozeile mit dem Befehl **conqat.bat -f <config-file-name >** gestartet. Die umfangreicheren Analysen werden gewöhnlich durch diese Analysemethode ausgeführt.

[6]http://www.graphviz.org/
[7]http://www.microsoft.com/de-de/download/details.aspx?id=19

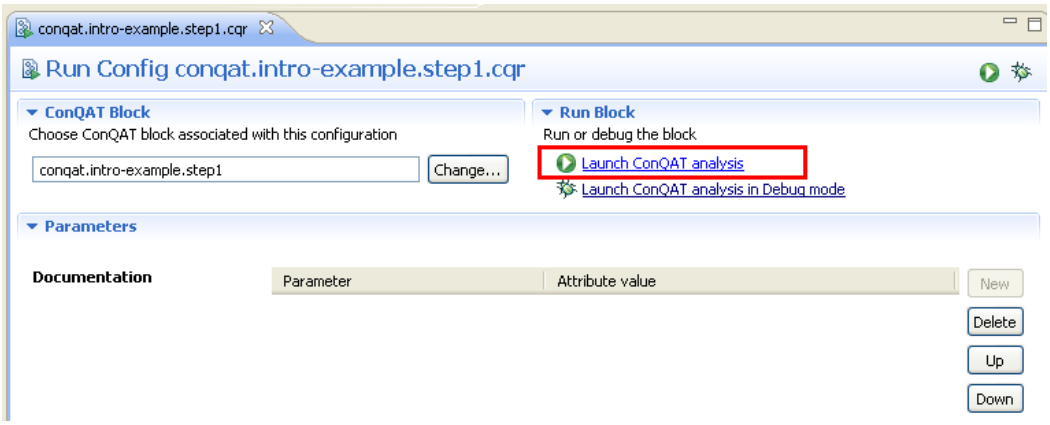


Abbildung 5.11.: Run Configuration-Dialog (conqat.in.tum.de)

5.3. Analyseergebnisse

ConQAT bietet die Möglichkeit die Ergebnisse einer Analyse in Form von Tabellen, Treemaps oder Charts darzustellen und als HTML-Seiten oder XML-Dateien zu speichern. Dadurch unterstützt ConQAT den Aufbau von den sogenannten „Quality Dasboards“, welche zum Steuern und Planen der IT-Projekte verwendet werden. Die Quality Dasboards ermöglichen nicht nur den Projektleitern einen Überblick über die qualitätsrelevanten Kriterien in einem Unternehmen, sondern auch den Softwareentwicklern. Die Abbildung 5.12 zeigt die verschiedenen Visualisierungen die durch ConQAT erstellt werden können.

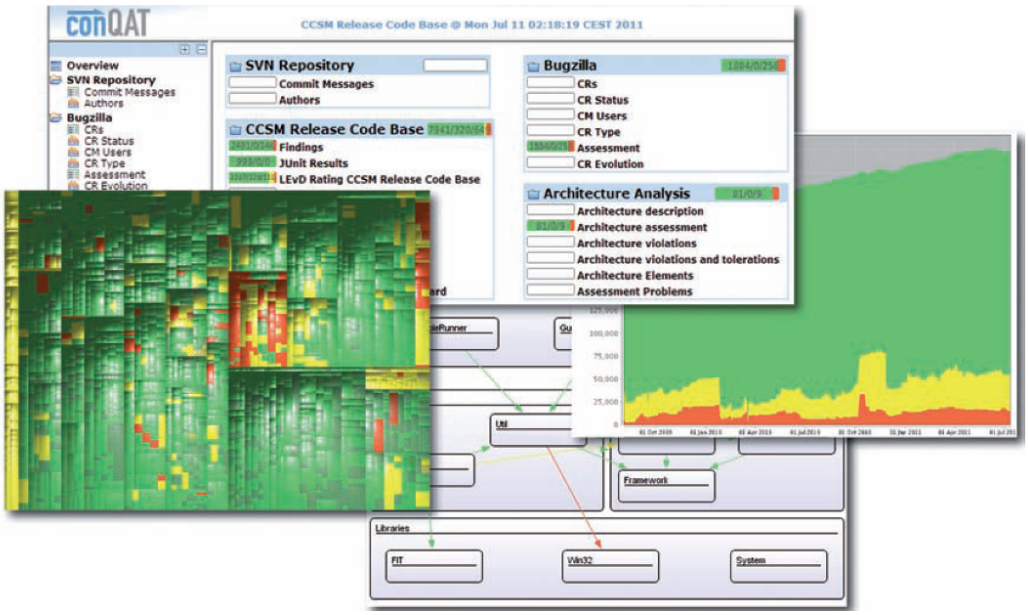


Abbildung 5.12.: ConQAT-Visualisierungsmöglichkeiten (F.Deißenböck, B.Hummel)

6. Einsatz von ConQAT im Unternehmen

Der wesentlichen Teil dieser Arbeit bestand darin, das Qualitätsanalysewerkzeug ConQAT beim Industriepartner „XYZ“ AG einzuführen und Analyseregeln bzw. Metriken zur Durchführung der kontinuierlichen Qualitätsanalyse vorzuschlagen.
In diesem Kapitel wird die Einführung von ConQAT und die Umsetzung der durchgeführten Analysen im Unternehmen näher erläutert.

6.1. Vorgehensweise

Das Software-Unternehmen „XYZ“ hat sich das Ziel gesetzt, die innere Qualität seiner Software-Produkte zu verbessern und dadurch die Wartung der Software zu erleichtern. Zum Erreichen des Ziels wurde im Rahmen dieser Diplomarbeit das in Kapitel 6 beschriebene Analysewerkzeug ConQAT eingeführt.
Zu Beginn war geplant ConQAT in der Java-Abteilung einzusetzen. Aus den Gesprächen mit den Leitern der Java- und C-Abteilung wurde schnell klar, dass die Java Abteilung bereits das Werkzeug Sonar zur Softwarequalitätsanalyse einsetzt.
Weil außerdem in der C-Abteilung kein Werkzeug zur Qualitätsanalyse vorhanden war, wurde entschieden, dort ConQAT einzuführen.
Nach einer kurzen Einarbeitung in das Werkzeug wurden die ersten ConQAT-Analyseblöcke (siehe Kapitel 6) erstellt, die zur Durchführung der Analyse notwendig sind. Zur gleichen Zeit wurde mir von dem Unternehmen in C geschriebener Programmcode zur Analyse zur Verfügung gestellt. Dadurch hatte ich die Möglichkeit, die von mir erstellten Analyseblöcke auszuprobieren. Aus dem Grund, dass dieses Werkzeug eher für die Analyse objektorientierter Sprachen konzipiert worden ist, war es mir nicht möglich alle seine verfügbaren Analysemöglichkeiten bei der Analyse des C-Codes anzuwenden. Alle erstellten Blöcke, die während der Analyse zum Einsatz kamen, wurden zu einem Analyseblock zusammengefasst (siehe Anhang C). Dadurch wurde der Analysevorgang vereinfacht und beschleunigt, da mehrere Analysen gleichzeitig durch eine Run Configuration ausgeführt werden konnten.
Nach dem alle für C nützlichen Analyseblöcke erprobt wurden, wurde gemeinsam mit meinem Betreuer bei „XYZ“ AG entschieden, ConQAT in einem dreier Team der C-Abteilung einzusetzen. Für die Erstellung der Analyseblöcke ist die Entwicklungsumgebung Eclipse notwendig. Daher wurde Eclipse zusammen mit ConQAT bei einem Mitglied dieses Teams installiert und eingerichtet. Zudem habe ich dem Mitglied die Funktionsweise der Werkzeuge erklärt und die von mir erstellten Analyseblöcke erläutert. Die Ausführung von ConQAT wurde anschließend von einem der Entwickler durch ein Skript vollständig automatisiert und von da an täglich ausgeführt. Für die Analyse des Codes wurde die neueste Version

2011.9 des Werkzeugs unter Linux verwendet. In den Daily Scrum Meetings wurde von den Entwicklern über die Probleme berichtet, die während der Ausführung von ConQAT auftraten. Einige davon konnten leicht behoben werden, andere erforderten die Unterstützung des Herstellers (TU München).

6.2. Durchgeführte ConQAT-Analysen

Als nächstes werden die durchgeführten ConQAT-Analysen im Unternehmen vorgestellt. Zudem werden die Ergebnisse und die Probleme während der Analyse kurz beschrieben.

6.2.1. LOC und SLOC

Das einfachste und nach wie vor am häufigsten verwendeten Umfangsmetrik ist „Lines of Code“ (LOC). Diese Metrik wird oft als Maß für die Komplexität eines Softwareprodukts eingesetzt, da es angenommen wird, dass die Übersichtlichkeit eines Softwaresystems, mit steigender Länge des Codes schlechter wird. Dabei ist zu beachten, dass die eingesetzte Programmiersprache auch eine sehr große Rolle spielt, da die erforderliche Quellcodezahl von der jeweiligen Sprache abhängig ist. Die Metrik und die Probleme, die sie mit sich bringt, wurden im Punkt 2.5.3.1 ausführlich beschrieben.
ConQAT bietet eine Analysemöglichkeit der LOC-Metrik mit Hilfe des „LOCAnalyzer-Blocks“. Dieser Block wird verwendet, um alle vorhandenen Zeilen Code inklusive die leere Zeilen und die Kommentare zu zählen. Durch den ConQAT-Prozessor „SLOCAnalyzer“ (Source Lines of Code) besteht noch dazu die Möglichkeit, die Kommentare und die leeren Zeilen von der Analyse auszuschließen. Somit wird nur die Anzahl der ausführbaren Codezeilen berücksichtigt. In dieser Diplomarbeit wurde die LOC-Metrik verwendet, um die gesamten Zeilen Code des zu analysierenden Softwareproduktes und die Größe der einzelnen C-Dateien zu ermitteln. Außerdem wurde sie in Kombination mit anderen Metriken, wie z. B. „Comment Ratio“ (siehe Punkt 6.2.5) angewendet, um gewisse Softwareeigenschaften präziser beurteilen zu können. Durch die LOC-Metrik wurden ein wenig mehr als 900.000 LOC ermittelt. Das war die Größe der Codebasis, die in dem dreier Team bei „XYZ“ AG analysiert wurde. Zudem wurde auch ein Softwareprodukt von „XYZ“ AG analysiert, das aus mehr als 5 Millionen Zeilen Code bestand. Dabei wurden auch C-Dateien ermittelt, die eine sehr große Programmgröße haben. Die größten dieser Dateien sind zwischen 5000 und 42.000 Zeilen lang. Zudem wurden auch übergroßere Funktionen gefunden, die über mehrere C-Dateien verstreut waren. Somit lagen die Dateien und die Funktionen deutlich über die in den Programmierrichtlinien angegeben Größen.

6.2.2. Clone Detection

Als Code Duplikate werden Stellen im Sourcecode bezeichnet, die mehrfach vorkommen und dadurch die Wartbarkeit des Codes enorm erschweren. Der Grund dafür ist, dass die

Codeduplizierungen gewöhnlich mehrere Fehler mit sich schleppen und die inkonsistenten Änderungen wie Bugfixes zu unerwarteten Nebeneffekte führen könnten. In Punkt 5.1.2.2 wurden ausführlich die Gründe und die Auswirkungen der Code Duplikate auf die Wartbarkeit eines Softwaresystems beschrieben. ConQAT kann sowohl syntaktisch (**Ungapped clones**) als auch semantisch (**Gapped clones**) gleiche Quellcodestellen analysieren, finden und dementsprechend darstellen. Zudem bietet ConQAT die Möglichkeit Code-Duplikate, die explizit gewollt sind, durch den Prozessor „BlackListFilter“ oder durch den Einsatz von „regulären Ausdrücken“, von einer weiteren Analyse auszuschließen. Für die Duplikatensuche wurde der in den ConQAT-Bibliothek vorhandenen Block **StatementCloneChain** verwendet. Dieser Block kann unabhängig von der Programmiersprache eingesetzt werden. Allerdings besteht damit nur die Möglichkeit eine Ungapped Clone Detection (auf semantische Ebene) durchzuführen. Die Metriken, die ConQAT bei der Duplikatensuche berechnen kann, wurden bereits im Punkt 5.1.2.2 erläutert.

Element	LoC	Clone LoC	Units	Clone Units	RFSS	UnitCoverage	Clone Count
ConQAT	906,085	221,358	392,467	101,266	316,547	0.258	27,026
afp	121,931	14,044	51,527	9,470	44,653.329	0.184	1,389
dbg	1,232	570	421	235	293.708	0.558	13
apr	58,787	20,184	26,196	8,539	19,689.393	0.326	2,009
base	77,753	15,072	27,410	6,432	23,319.05	0.235	755
aix	384	279	127	117	23.187	0.921	33
freebsd	396	320	124	117	19.13	0.944	37
hpux	385	313	133	116	29.422	0.872	32
linux	493	262	144	115	40.197	0.799	27
macosx	2,163	791	812	247	628.755	0.304	43
nsk	870	247	301	110	203.012	0.365	22
openbsd	403	305	133	117	28.13	0.88	37
os2	4,843	976	1,269	287	1,080.921	0.226	30
os390oe	713	303	242	127	136.354	0.525	31
os400	1,549	294	658	145	571.032	0.22	12
solaris	398	275	124	115	22.032	0.927	27
unix	7,735	1,807	2,805	603	2,459.371	0.215	51
win	10,859	2,838	4,139	1,409	3,336.29	0.34	115
css3	65,561	33,579	20,082	9,013	13,078.559	0.449	1,530
css21	30,716	18,623	8,190	4,788	4,077.665	0.585	986
value	5,120	901	2,329	499	1,968	0.214	66
format2	66,705	3,782	22,963	1,833	21,706.243	0.08	283
mfffilt	258,152	86,614	121,599	40,746	88,861.214	0.335	17,084
mmd	37,115	7,059	16,789	3,610	14,431.429	0.215	438
oox	76,753	33,621	33,819	14,801	21,278.348	0.438	9,047
utl	3,308	729	1,351	379	1,085.265	0.281	59
wml	68,606	31,605	30,113	13,750	18,367.766	0.457	8,817
svg	23,024	4,115	11,428	2,408	9,920.13	0.211	304
xff	34,127	10,932	17,499	5,273	13,190.04	0.301	1,914
xfo	65,241	27,253	31,907	12,954	20,958.995	0.406	5,209
xif	21,892	3,634	10,157	1,700	9,082.272	0.167	172
namegen	1,700	130	738	64	696.572	0.087	7
parea	99,665	24,917	46,855	12,716	37,569.342	0.271	2,254
pdf	130,426	14,841	63,980	8,821	58,645.962	0.138	942
rmg	25,405	8,195	11,117	3,632	8,327.335	0.327	773

Abbildung 6.1.: Unit Coverage

Eine der wichtigsten Metriken die dabei beschrieben wurden, ist die „Unit Coverage“. Diese Metrik gibt die Wahrscheinlichkeit an, mit der ein willkürlich ausgewähltes Statement Teil eines Clones ist. Koschke [GGIW12] berichtet über mehreren Studien mit einer Unit Covarage zwischen 7 und 23%, und eine Studie mit eine Wahrscheinlichkeit von 56%, welche er als extrem einschätzte. Während der Analyse bei „XYZ“ AG wurden mehrere Code-Duplikate gefunden. Hauptsüchlich konnten exakte Code-Kopien (Copy&Paste Code) und syntaktisch identische Code-Kopien gefunden werden. Die syntaktisch identische Code-Kopien haben sich nur an ihre Variablen oder an ihre Funktion-Bezeichner (identifier) unterschieden. Die Abbildung 6.1 zeigt die Unit Coverage des Quellcodes, der bei „XYZ“ AG analysiert wurde. Sie variiert zwischen 8 und 94 %. Dadurch wird es deutlich, dass sich an manchen Stellen im Sourcecode viele Code-Duplikate befinden.

6.2.3. Magic Number

Als „Magic Number“ wird in der Programmierung das Vorkommen von Zahlen im Quellcode verstanden, deren Bedeutung von dem Betrachter des Codes nicht erkennbar ist. Martin Fowler bezeichnet die Magic Number als eine der ältesten „Krankheiten“ in der Programmierung und rät diese durch Konstanten zu ersetzen [FBBO02].
Oft ist der Fall, dass die „Magische Zahlen“ innerhalb einer Datei oder Klasse mehrfach verwendet werden. Nur, wenn der Wert dieser Zahl geändert werden soll, müssen damit auch alle Stellen im Quellcode geändert werden, an der diese Zahl auftaucht. Dabei ist die Wahrscheinlichkeit sehr hoch, dass Zahlen an bestimmten Stellen vergessen und nicht geändert werden. Dadurch könnten unerwarteten Nebeneffekte im Programmcode auftauchen. Andererseits sogar wenn keine Änderungen vorgenommen werden sollten, ist es trotzdem schwer für den Leser des Codes der Sinn der Verwendung des vorliegenden Zahlenwertes zu verstehen. Die Verwendung Magischer Zahlen macht den Programmcode schwer lesbar, unverständlich und fehleranfällig.

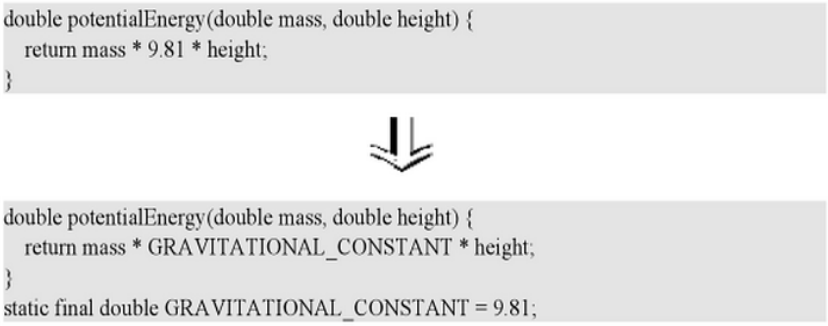


Abbildung 6.2.: Magic Number (Fowler, et al., 2002 S. 204)

In der Abbildung 6.2 stellt die Zahl 9,81 eine Magic Number dar. Durch die Verwendung der Konstante „GRAVITATION_CONSTANT“ wird dieses Problem behoben. ConQAT ermöglicht die Suche nach Magic Number mithilfe des Prozessors „RedundantLiteralAnalyzer“

und stellt die Ergebnisse tabellarisch dar. Dabei bietet ConQAT die Möglichkeit, die ungewollten Zahlen, wie z. B. -1, 0 und 1, aus der Analyse auszuschließen.
Während der durchgeführte Analyse mit ConQAT wurden Magic Numbers in mehreren Stellen im Sourcecode identifiziert.

6.2.4. Identifier

Die Identifikatoren (Bezeichner) sind das Vokabular eines Softwareprogramms. Etwa 70% des Quellcodes der Softwaresysteme besteht aus Identifikatoren [DP06b]. Daher ist es enorm wichtig die richtige und passende Wahl der Bezeichner-Namen zu treffen, da sie das Verstehen eines Sourcecodes essentiell beeinflussen könnten. Schlecht ausgewählte Bezeichner-Namen erschweren in enorme Maßen die Lesbarkeit des Programms und dadurch beeinträchtigen sie die Wartbarkeit in einer späteren Phase des Softwareentwicklungsprozesses. Generell sollten Bezeichner mit mehreren Bedeutungen (Homonyme) vermieden werden. Zudem sollten sie präzise und korrekt ausgewählt und nicht das selbe Konzept (Synonyme) in einem Programm repräsentieren [DP06a]. Beispiele für Homonyme und Synonyme könnten z. B. die Bezeichner **path** oder **list_head** und **list_first** sein [Voh10]. ConQAT bietet die Möglichkeit alle gefundenen Bezeichner in tabellarischer Form darzustellen und die Häufigkeiten deren Auftretens zu berechnen. Somit könnte diese Analysemöglichkeit als Ergänzung einer Manuellen Inspektion dienen. Mit Hilfe der ConQAT-Analyse wurden an mehreren Stellen im Code schlecht und unpräzise formulierte Bezeichner gefunden.

6.2.5. Comment Ratio

Die Kommentare im Programm sind für die Wartung eines Softwaresystem von einer sehr großen Bedeutung. Sie erleichtern die Lesbarkeit und die Verständlichkeit des Quellcodes und verbessern damit die Wartbarkeit des Softwareprodukts. ConQAT bietet die Metrik „Comment Ratio“, um das Verhältnis der Kommentare zu den Quelltext berechnen zu können. Für diese Aufgabe ist der Prozessor „CommentRatioAnalyzer“ zuständig. Allerdings sind weitere Prozessoren notwendig, wie z. B. „AssessmentColorizer“ und „TreeMapLayouter“, um die Ergebnisse dieser Metrik zu visualisieren. Die Resultate nach der Berechnung dieser Metrik können in tabellarischer Form oder als Treemaps dargestellt werden. Bei den Treemaps ist es wichtig zu wissen, dass die rot markierten Kästchen die unzureichend kommentierte Programmdateien darstellen. Die gelben und die grünen Kästchen visualisieren dagegen tolerierbaren und ausreichend kommentierte Codefragmente. Dabei hat der Anwender des Werkzeugs selbst die Möglichkeit das prozentuale Verhältnis zu bestimmen. Für diese Arbeit wurde das prozentuale Verhältnis wie folgt definiert:

- **0-20%:** - unzureichend
- **20-40%:** - tolerierbar
- **über 40%:** - ausreichend

Dabei ist zu beachten, dass diese Metrik keine Aussage über die Qualität der Kommentare macht. Sie zeigt nur, wie viel Kommentare ein Modul hat. Durch die ausgeführte ConQAT-Analyse wurden mehrere C-Dateien identifiziert, die eine unzureichende Kommentar-Rate aufweisen. Darunter waren auch Module mit einer Kommentar-Rate von 1% bei einer Code-Länge von 20.000 Zeilen (siehe Bild 6.3).

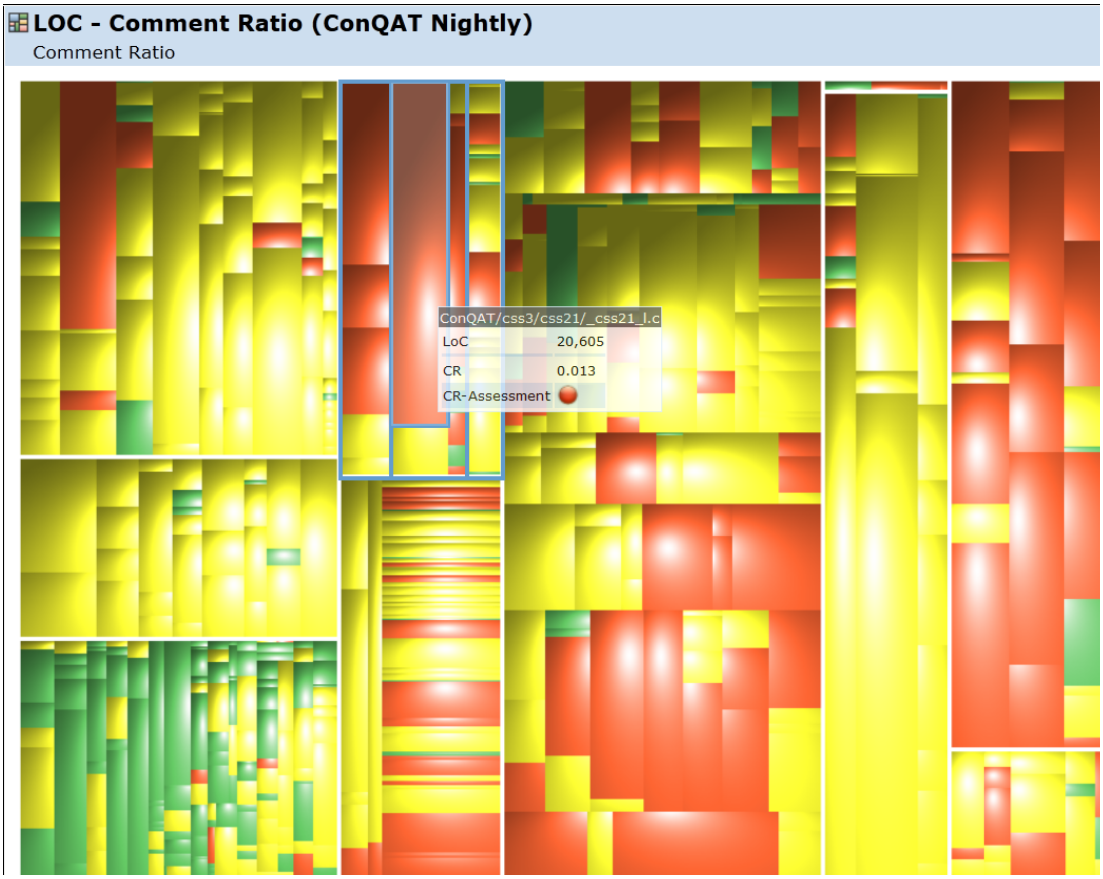


Abbildung 6.3.: Comment Ratio der „XYZ“-Software

6.2.6. Comment Language

Heutzutage werden die einzelnen Softwarekomponente unabhängig voneinander in verschiedenen Ländern oder durch verschiedenen Personen entwickelt. Daher werden oft Software Produkte geliefert, deren Kommentare in einer fremden Sprache verfasst sind. Es ist sicherlich schwer für einen Entwickler aus Deutschland in chinesisch geschriebene Text zu verstehen. „XYZ“ AG bekommt keine Software zugeliefert, dafür aber beschäftigen sie Mitarbeiter aus vielen verschiedenen Ländern. Daher ist Englisch als offizielle Unternehmenssprache etabliert worden. Aus diesem Grund wurde ein Analyseblock konstruiert, welcher die Ermittlung der Kommentar-Sprache ermöglicht hat. Zu diesem Zweck wurde der Block „LanguageTreeMap“ verwendet, der in der ConQAT-Bibliothek vorhanden war.

Durch die Analyse wurden mehrere Module gefunden in denen andere Sprachen als die offizielle englische Sprache vorkommen.

6.2.7. Task Tags

Task Tags werden in der Software-Entwicklung als eine Methodik angesehen, um bestimmte Aufgaben, direkt im Code zu markieren. Gewöhnlich werden Codestellen mit Tags versehen bei denen Nacharbeit nötig ist.
Die am häufigsten verwendeten Task Tags sind **TODO, FIXME, XXX und HACK**.
Mit **TODO** werden Stellen im Quellcode markiert, die noch erweitert werden müssen. Mit **FIXME** werden Codepassagen markiert, die ausgebessert werden müssen, bevor die Software zum Einsatz kommt. Codestellen, die chaotisch und allgemein schlecht geschrieben sind und dringend verbessert werden müssen, werden mit dem Tags **XXX** markiert [FH10].
Mit ConQAT werden alle Task Tags visualisiert und in Form von Treemaps und Tabellen dargestellt. Dadurch haben die Entwickler einen gesamten Überblick über alle Codestellen, die eine Nachbesserung nötig haben.

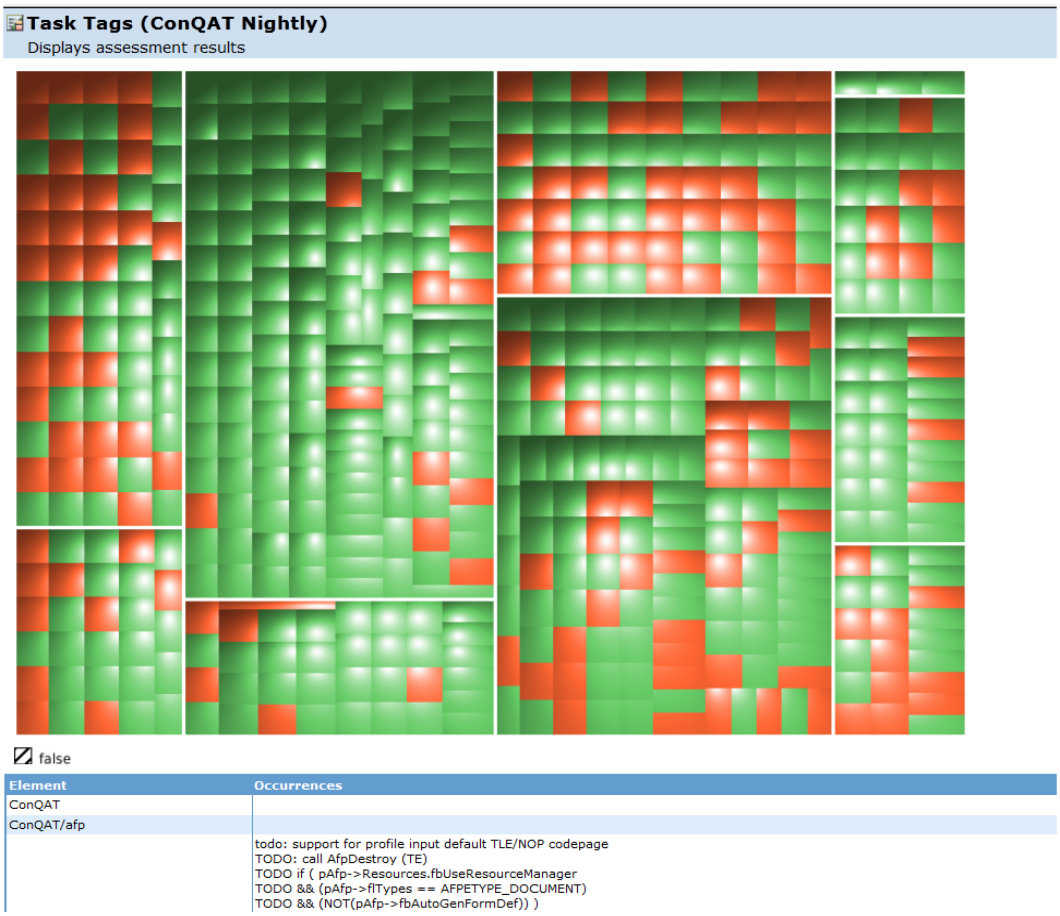


Abbildung 6.4.: Task Tags in der „XYZ“-Software

6.2.8. Trend Analyse

In vielen Fällen ist es sinnvoll nicht nur bestimmte Qualitätsdefizite zu analysieren, sondern auch ihre Entwicklung über die Zeit zu verfolgen. Dafür bietet ConQAT die Möglichkeit mit Hilfe von Trendanalysen die positiven und negativen Entwicklungen bestimmter Qualitätskriterien zu überwachen. Dadurch kann die Qualität der Software für die Entwickler oder für die Projektleitung sichtbar gemacht werden. Zum Persistieren und Laden von früheren Daten für die Trendanalyse wurde das Datenbanksystem HSQLDB verwendet. Das Bild 6.5 zeigt eine einmonatige Trendanalyse der Software des Unternehmens „XYZ“, welche die Kenngröße **Clone Units** analysiert und darstellt.

Abbildung 6.5.: Cloned Units als Trendanalyse

6.2.9. Include Directive

Die „Include Directive“ in der Sprache C dient zum Einbinden externer Quelltext-Dateien. ConQAT analysiert die Kreise (zirkuläre Abhängigkeiten) in den Include-Strukturen der C-Systeme mit Hilfe dem Prozessor **IcludeAnalyzer** und dem Programm **dot** aus dem **GraphViz-Paket**. Aufgrund fehlender Dokumentation wurde erst kurz vor dem Abgabetermin ein ConQAT-Block für die Analyse der Include-Direktiven erstellt. Daher gab es keine ausreichende Zeit, um diesen Block auszuführen. Der Analyseblock wurde, wie auch alle

anderen erstellten Analyseblöcke, als Datei in das Wiki-System des Unternehmens angehängt und dadurch für die Entwickler zur Verfügung gestellt.

6.3. Ergebnisse

Mit Hilfe der durchgeführten ConQAT-Analysen konnten verschiedenen Qualitätsdefizite an mehreren Stellen im Quellcode festgestellt werden. Es wurden Code-Duplikate, Magic Number, zu große C-Dateien, zu lange Funktionen, Kommentare in verschiedenen Sprachen, Funktionen mit zu langen Parameterlisten und eine unzureichende Kommentar-Rate identifiziert. Die Code-Duplikate, die gefunden wurden, waren zwischen 10 und 890 Zeilen lang. Zudem waren die Codeduplizierungen über mehrere Dateien verteilt, manche davon traten in 60 unterschiedlichen Dateien auf. Dabei variierte die „UnitCoverage“ (siehe Punkt 5.1.2.2) Metrik zwischen 8 und 94 %. Es wurden auch sehr große Module identifiziert, die aus 5000 und 42.000 Codezeilen bestehen. Zudem enthielten einige Dateien Funktionen mit bis zu 2500 Zeilen Code und bis zu 30 Parametern. In mehreren Dateien wurden viele komplexe Funktionen mit einer McCabe Komplexität von über 100 gefunden. Eine „McCabe Zahl“ von über 50 gilt bereits als undurchschaubar (siehe Kapitel 2). Viele von den Dateien waren außerdem nicht ausreichend kommentiert. Darunter waren auch Module mit einer Kommentar-Rate von 1% bei einer Code-Länge von 20.000 Zeilen (siehe Bild 6.3). Während der durchgeführten Analyse mit ConQAT wurden Magic Numbers an mehreren Stellen im Sourcecode lokalisiert. Einige der Magic Numbers traten bis zu 30-mal auf und wurden nicht als Konstanten definiert. Des weiteren wurden an mehreren Stellen im Code schlechte und unpräzise Bezeichner festgestellt. Durch die Analyse wurden mehrere Module gefunden, in denen Kommentare in verschieden Sprachen auftraten. Die Trend-Analyse (siehe Bild 6.5) zeigt auch, dass für eine Verbesserung der Situation während der Diplomarbeit keine Maßnahmen getroffen wurden. Die Entwickler begründeten es mit fehlender Zeit für die Refaktorisierung.
Trotz der gefunden Qualitätsdefizite wurde den Entwicklern keine zusätzliche Zeit zur Behebung dieser zur Verfügung gestellt. Daher konnten auch während der Diplomarbeit keine positiven Ergebnisse zu einer Verbesserung der inneren Qualität der Software des Unternehmens festgestellt werden.

6.4. Probleme bei der Umsetzung

Der Einsatz neuer Technologien ist oft mit Schwierigkeiten verbunden. Auch während der Einführung des Werkzeugs ConQAT traten einige Probleme auf. Ursächlich hierfür war insbesondere die fehlende und veraltete ConQAT-Dokumentation, was die Einarbeitung in das Werkzeug in erheblichem Maße erschwert und somit die Einarbeitungsphase unnötig verlängert hat. Ein weiteres Problem war die benötigte Verarbeitungszeit von ConQAT. Zwar dauern die meisten Analysen bei 5 Millionen Codezeilen nur zwischen einer und zehn Minuten, die Clone-Detection nahm aber bis zu 30 Stunden in Anspruch. Die Ausführung der Clone-Detection-Analyse mit dem Eclipse Editor war nicht möglich und brach nach einigen

Minuten erfolglos ab. Nach einer Anpassung der Parameter für die Java Virtual Machine war die Clone-Detection-Analyse mit der konsolenbasierten ConQAT Engine möglich. Die Parameter wurden in der mitgelieferten Batch-Datei wie folgt angepasst.
Der Wert der maximalen Größe des Java-Heap-Speichers (**xmx**) wurde von 512 auf 2048MB erhöht. Zusätzlich wurde auch die maximale Stapelgröße (**xss**) auf 32MB gesetzt. Weil außerdem die Analyse über die Kommandozeile erfolgte, wurde auch der Parameter **server**[1] in der Batch-Datei hinzugefügt und der Parameter **MaxPermSize**[2] [DFH+10] auf 512MB gesetzt.
Trotz allem dauert der Analysevorgang etwa 30 Stunden. Eine weitere Erhöhung der Parameter lies die verfügbare Hardware nicht zu. Daher wurde in ConQAT die zu analysierende Code-Menge auf 900.000 Zeilen herab gesetzt. Diese Code-Menge lies sich deutlich schneller analysieren. Die Analyse dauerte lediglich zehn Minuten.

6.5. Vorschläge

Mit Hilfe des Qualitätsanalysewerkzeugs ConQAT wurde eine Basis für die Zukunft geschaffen, um eine kontinuierliche Qualitätsanalyse durchführen zu können. Zudem wurden auch Vorschläge für die Verbesserung der Qualität der Softwaresysteme des Unternehmens gemacht. Als erstes wurde die Einführung von Kodier- und Dokumentationsrichtlinien vorgeschlagen. Dadurch kann ein Unternehmensstandard geschaffen werden, um die Zusammenarbeit der Entwickler zu vereinfachen und das Zurechtfinden in fremden Code zu erleichtern. Diese Regeln können automatisiert mithilfe des eingeführten Werkzeugs ConQAT überprüft werden. Es wurde auch die Einführung von Reviews und die Erstellung von Checklisten vorgeschlagen. Die Checkliste soll dem Prüfer bei der Durchführung der Reviews helfen und die wichtigsten Qualitätsaspekte benennen. Außerdem muss für eine kontinuierliche Qualitätsanalyse ein unterstützender Prozess eingeführt werden, der eine schnelle Reaktion auf Qualitätsverstöße ermöglicht. Nur durch einen gut durchdachten Prozess lassen sich die Qualitätssicherungsmaßnahmen einplanen und realisieren.
Eines der gravierendsten Defizite der Software waren Code-Duplikate. Die Häufigkeit eines Code-Duplikats lässt sich durch ConQAT erfassen. Daher wurde vorgeschlagen zuerst die häufigsten Code-Duplikate zu eliminieren. Des weiteren wurde vorgeschlagen zuerst die größten Code-Duplikate zu entfernen, da unter den kürzeren oftmals „False Positive“ festgestellt wurden. Mittelfristig sollte die Architektur rekonstruiert werden, um Schwachstellen der Software-Architektur aufzudecken. In Zukunft sollte die Entwicklung neuer Funktionen durch ConQAT auf Einhaltung von Code-Richtlinien untersucht werden. Zudem sollte in kleinen Schritten der bestehende Code an die Richtlinien angepasst werden. Um die Softwarequalität zu verbessern, wurden Metriken, die das Werkzeug ConQAT unterstützt, wie z. B. Line of Codes, Clone LoC, Clone Count, UnitCoverage, Comment Ratio für das Unternehmen vorgeschlagen. Für die Analyse von Bug Patterns (wie z. B. Pufferüberläufe) wurde der

[1] http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html
[2] http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

Einsatz des statischen Analysewerkzeugs PC-Lint angeraten. Dafür wurde ein ConQAT-Analyseblock erstellt, der die Ergebnisse von PC-Lint verarbeiten und visualisieren kann. Der Analyseblock wurde im Wiki der Firma allen C-Entwicklern zugänglich gemacht.

7. Analyse der Auswirkungen nach ConQAT-Einsatz

7.1. Analyse

Die letzte Aufgabe der Diplomarbeit bestand darin, eine zweite Mitarbeiterbefragung bei „XYZ“ AG durchzuführen. Durch diese Befragung sollten eventuelle Verbesserungen nach dem Einsatz von ConQAT analysiert werden. Zudem sollten auch die relevanten Ergebnisse von ConQAT bezüglich die Softwarequalitätsanalyse ermittelt werden.

7.1.1. Konstruktion und Vorgehensweise der Analyse

In der C-Abteilung wurde ConQAT zu Unterstützung der Qualitätsanalyse in einem dreier Team eingesetzt. Aus diesem und aus dem Grund, dass eine mündliche Befragung viel schneller durchzuführen ist, hat man sich entschieden, eine Befragung in Form von Interviews durchzuführen. Die Interviews enthielten neun Fragen und wurden innerhalb von zwei Tagen durchgeführt. Die Fragen bezogen sich hauptsächlich auf ConQAT und die Erkenntnisse der Mitarbeiter während der Arbeit mit ConQAT. Das Interview bestand aus den folgenden Hauptfragen:

1. Was halten Sie persönlich, losgelöst von der Entwicklung bei „XYZ“ AG, von ConQAT?
2. Welche von den Ergebnissen von ConQAT sind hilfreich oder könnten hilfreich für Sie in der Zukunft sein?
3. Haben Sie zusammen mit den Teamkollegen die Ergebnisse von ConQAT angeschaut und diskutiert?

Zum Ende der Befragung wurden alle Ergebnisse zusammengefasst und in der Ausarbeitung dokumentiert.

7.1.2. Ergebnisse der Analyse

Die Mitarbeiter haben angegeben, dass sie das Prinzip des Werkzeugs qualitativ schlechtere Teile im Code zu visualisieren, gut finden. Allerdings bemängelte einer der Befragten, dass man durch Klicken auf die Treemap-Bilder nicht zu dem jeweiligen Code Stück gelangen kann. Speziell für die Erkennung von Copy&Paste Stellen im Code hat der selbe Entwickler angegeben, dass er die ConQAT-Analyse nur über die Kommandozeile und nicht mittels

Eclipse ausgeführt hat. Für viele der Analysen bietet ConQAT mittels Eclipse viele andere Visualisierungsmöglichkeiten. Obwohl sich der Entwickler dessen bewusst war, hat er die Entwicklungsumgebung nicht installiert, daher konnte er auch nicht die zusätzlichen Analysemöglichkeiten von ConQAT ausprobieren. Als hilfreiche ConQAT-Analysen wurden von allen drei Entwicklern die Clone- und die Trendanalyse, die Magic Number Analyse und die Metrik Lines of Code genannt. Es wurde auch von einem Entwickler erwähnt, dass ConQAT bei der Clone Analyse viele False Positive liefert. Ein False Positive ist hierbei ein als Clone markierter Code, welcher von Entwicklern nicht als Clone erachtet wird. Aus diesem Grund war er der Meinung, dass ConQAT nicht für eine C-Code Analyse geeignet ist. Auf die Frage, wie die Ergebnisse der Analyse kommuniziert werden sollen, gaben zwei der Entwickler an, dass sie die Ergebnisse von ConQAT per E-Mail erhalten wollen. Der dritte Entwickler hat angegeben, dass die Ergebnisse einmal monatlich per Review diskutiert werden sollten. Alle Befragten haben angegeben, dass sie keine explizite Zeit zur Verbesserung, der von ConQAT entdeckten Qualitätsdefizite, haben. Es wurde auch angegeben, dass das Team nur ein einziges Mal die Ergebnisse von ConQAT diskutiert hat. Auf die Frage, welche Erkenntnisse aus der Diskussion gewonnen wurden, haben zwei der Entwickler erwähnt, dass ein paar Clone Duplikate im Code gefunden worden sind. Der dritte Entwickler war der Meinung, dass er durch die ConQAT-Analysen keine größeren Erkenntnisse gewinnen konnte. Aller drei Entwickler waren der Meinung, dass ConQAT in Zukunft zur Verbesserung der die Wartbarkeit des Codes beitragen könnte, falls sich die False-Positive-Rate reduziert und noch weitere Optionen für C-Code angeboten werden.

Während den Interviews hatte ich den Eindruck, dass sich die Entwickler zu wenig mit dem Werkzeug ConQAT beschäftigt hatten, um einen fundierten Überblick über seine Leistungsfähigkeit zu gewinnen. Ein Grund dafür ist sicherlich die zu geringe Zeit, die den Entwicklern für die Qualitätssicherung zugesprochen wird. Außerdem fehlt ein unterstützender Prozess in der C-Abteilung für die Qualitätssicherung. Dadurch ist der sinnvolle Einsatz eines Softwarequalitätsanalysewerkzeugs unmöglich. Des weiteren konnten die Entwickler nicht alle Analyse- und Visualisierungsmöglichkeiten von ConQAT in vollem Umfang nutzen, da nicht alle Entwickler Eclipse installiert haben.

7.2. Zusammenfassung und Evaluierung der gesamten Analyseergebnisse

Abschließend folgt eine Einschätzung über die bisherige Qualitätssicherung und die Akzeptanz des Qualitätsanalysewerkzeugs ConQAT im Unternehmen.
Im Rahmen dieser Diplomarbeit wurden zwei Analysen durchgeführt. Das Hauptziel der ersten Analyse bestand darin, den aktuellen Zustand des Qualitätssicherungsprozesses des Unternehmens „XYZ“ zu ermitteln. Als Ergebnis der Analyse wurde festgestellt, dass in diesem Unternehmen kein Qualitätssicherungsprozess vorhanden ist. Es hat sich herausgestellt, dass die C-Abteilung die Qualität ihrer Software nicht misst und wenig Wert auf die innere Qualität der Software legt. Zudem besteht die Qualitätssicherung ausschließlich aus Tests, deren Realisierung sich aber aufgrund der schwammigen Spezifikation enorm erschwert.

Außerdem werden in dem Unternehmen keine Qualitätsziele während des Softwareentwicklungsprozesses festgelegt und verfolgt.
Zur Verbesserung dieser Situation wurde mit Hilfe des Analysewerkzeugs ConQAT, eine Basis zur kontinuierlichen Qualitätsanalyse der inneren Qualitätsmerkmale (Wartbarkeit, Weiterentwicklungsfähigkeit) geschaffen. Die durchgeführten Analysen mit ConQAT haben gezeigt, dass in dem Sourcecode der C-Abteilung viele Qualitätsdefizite vorhanden sind, welche die Wartbarkeit der Software enorm erschweren. Durch die zweite Analyse konnte festgestellt werden, dass für die Behebung dieser Defizite keine Zeit zu Verfügung steht. Das liegt auch daran, dass sich die Entwickler in C-Bereich während der Entwicklung ihrer Software an keinem Vorgehensmodell orientieren. Dadurch lassen sich Qualitätssicherungsmaßnahmen sehr schwer einplanen und realisieren. Die zweite Analyse zeigte auch, dass das Team, bei dem ConQAT eingesetzt wurde, nur ein einziges Mal die Analyseergebnisse des Werkzeugs diskutiert hatte. An einigen Stellen konnten auch „False Positives“ lokalisiert werden. Wegen der fehlenden Zeit und dem Auftreten von „False Positives“ fand das Werkzeug nur in geringem Maße eine Akzeptanz bei den Entwicklern. Des Weiteren hat sich herausgestellt, dass ohne dem Vorhandensein eines unterstützenden Prozesses, ein sinnvoller Einsatz eines Softwarequalitätsanalysewerkzeugs unmöglich ist. Aus diesem Grund konnten nach dem Einsatz von ConQAT keine positiven Ergebnisse in Bezug auf die Verbesserung der Softwarequalität festgestellt werden.

8. Zusammenfassung und Ausblick

In Rahmen dieser Diplomarbeit wurde beim Industriepartner „XYZ“ AG das Qualitätsbewertungswerkzeug ConQAT eingesetzt, um fehlende oder verteilte qualitätsrelevante Informationen zu sammeln und diese in einer aggregierten und verdichteten Form darstellen zu können. Dadurch wurde eine Möglichkeit zur Steuerung und Planung von täglichen Aktivitäten der Qualitätssicherung im Unternehmen geschaffen. Vor dem Einsatz dieses Werkzeugs wurde eine Analyse der Qualitätssicherung bei „XYZ“ AG durchgeführt, um zu erfahren, welche Maßnahmen bzgl. der Softwarequalität in dem Unternehmen unternommen werden. Diese Analyse fand in Form von mündlichen und schriftlichen Befragungen statt. Nach der Analyse wurde ConQAT in einem dreier Team im C-Bereich eingesetzt. Hierfür wurden mehrere ConQAT-Analyseregeln konzipiert, die zur Ermittlung der Schwachstellen innerhalb des „XYZ“-Quellcodes nötig waren. Es wurde gezeigt, dass ConQAT Qualitätsdefizite und Probleme innerhalb des Sourcecodes der C-Abteilung erkennt und es den Softwareentwicklern dadurch ermöglicht, diese Schwachstellen so früh wie möglich zu finden und auszubessern. Dem Unternehmen wurden Vorbesserungsmaßnahmen vorgeschlagen. Dennoch wird den Entwicklern im Unternehmen leider keine explizite Zeit für die Verbesserung der Schwachstellen im Quellcode zur Verfügung gestellt, sodass keiner der gefundenen Qualitätsdefizite während der Durchführung der Diplomarbeit behoben wurde. Zum Schluss der Arbeit wurde eine zweite Analyse durchgeführt, um die positiven und die negativen Erkenntnisse nach dem Einsatz von ConQAT zu erfassen und die eventuellen Verbesserungen der aktuellen Qualitätssicherung ermitteln zu können.

Ausblick

Die sogenannten Qualitätscockpits ermöglichen eine aggregierte Sicht auf den aktuellen Qualitätszustand eines Softwaresystems in Form von Trendbetrachtungen, Treemap-Grafiken oder Charts. Dadurch können sich die Softwareentwickler schnell einen Überblick über die ausschlaggebenden Qualitätsfaktoren zu einem Softwaresystem oder -projekt verschaffen.
Um einen tatsächlichen Erfolg durch eine Qualitätsanalyse gewinnen zu können, sollte diese kontinuierlich über einen langen Zeitraum zum Zwecke der Qualitätsverbesserung und -kontrolle durchgeführt werden. Zusätzlich könnte durch einen gewissen Automatisierungsgrad der Analyseaufwand enorm minimiert werden. Das Qualitätscockpit könnte dabei stündlich, täglich oder monatlich automatisch aktualisiert und vom Entwickler betrachtet werden.
Ein anderer wesentlicher Punkt ist, dass in einem Unternehmen ein gewisser unterstützender Prozess für die kontinuierliche Qualitätsanalyse vorhanden sein sollte, damit der Verlauf

für die Beseitigung der Qualitätsdefizite explizit festgelegt werden kann. Die Entwickler erhalten dadurch mehr Zeit für Refaktorierung und Nachbesserungen ihres Codes. Zudem sollten auch manuelle Inspektionen durchgeführt werden, welche der Ergänzung der werkzeugunterstützten Qualitätsanalyse dienen. Nur dadurch lässt sich ein verzerrungsfreies Bild der Softwaresystemqualität zeichnen und der Erfolg der kontinuierlichen Qualitätsanalyse garantieren.

A. Anhang - Fragebogen

1. Wie viele Jahre Erfahrung haben Sie in der IT – Industrie?

- ○ weniger als 5 Jahre
- ○ 5 bis 10 Jahre
- ○ 10 bis 20 Jahre
- ○ mehr als 20 Jahre

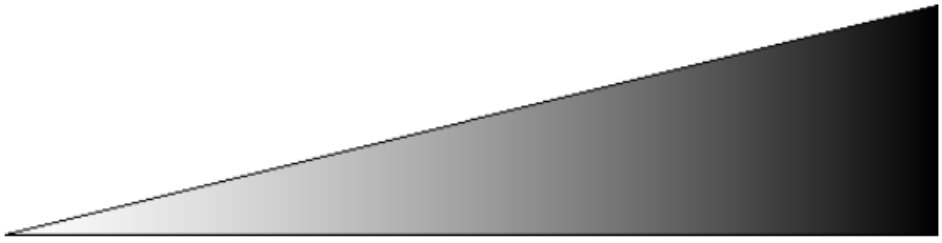
2. Welche Programmiersprachen benutzten Sie hauptsächlich während der Arbeit?

- ○ JAVA
- ○ C
- ○ Andere

3. Welchen Größenklassen sind die durchgeführten Teilprojekte zuzuordnen?

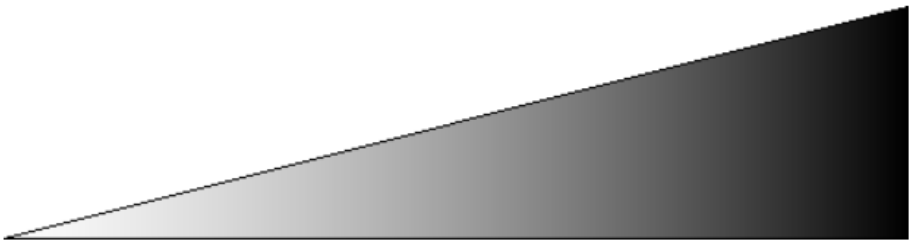
- ☐ kleiner 3 Personenmonate
- ☐ 3 bis 6 Personenmonate
- ☐ 6 Personenmonate bis 1 Personenjahr
- ☐ 1 bis 5 Personenjahre
- ☐ größer 5 Personenjahre

4. Wie wichtig ist die Software-Qualität für Sie als Entwickler, Projektleiter oder Manager ?



○ völlig egal ○ weniger wichtig ○ wichtig ○ sehr wichtig

5. Wie wichtig ist die Software-Qualitätsanalyse für Sie als Entwickler, Projektleiter oder Manager ?



○ völlig egal ○ weniger wichtig ○ wichtig ○ sehr wichtig

6. Scrum wird als Vorgehensmodell zur Softwareentwicklung bei „XYZ“ AG eingesetzt. Nennen Sie bitte Ihre Anregungen und Verbesserungsvorschläge bezüglich der Software-Qualität und der Software-Qualitätsanalyse in Scrum – Umfeld!

7. Das Wort Metrik entstammt dem Griechischen und bedeutet Kunst des Messens. In dem Software Engineering macht man damit quantifizierten Aussagen über Produkte oder Entwicklungsprozesse.
Kennen Sie Software-Qualitätsmetriken? Wenn ja, benennen Sie diese bitte!

○ ja
○ nein
○ weiß nicht

8. Welche dieser Metriken sind ihrer Meinung nach am nützlichsten für die Software-Qualitätsanalyse bei „XYZ“ AG?

9. Mit welchen Methoden des Software – Engineering werden bei „XYZ“ AG die Software – Qualitätssicherungsprozesse verfolgt? (falls andere bitte angeben)

- ☐ Systematisches Testen
- ☐ Einsetzen von Software- Metriken
- ☐ Einhaltung von Programmierkonventionen
- ☐ Statische Codeanalyse
- ☐ Pair Programming
- ☐ Dokumentationsrichtlinien
- ☐ Andere
- ☐ Keine der Genannten

10. Denken Sie, dass die eingesetzten Methoden des Software – Engineering bei „XYZ“ AG zu Verbesserung der Software-Qualität beitragen?

- ○ ja
- ○ nein
- ○ weiß nicht
- ○ sonstiges

11. Welche Tools für Software Qualitätsanalyse kennen Sie ?

12. Wie zufrieden sind Sie mit dem eingesetzten Tool Sonar? (Die Frage richtet sich an die JAVA – Entwickler/innen)

- ○ sehr zufrieden
- ○ zufrieden
- ○ eher unzufrieden
- ○ sehr unzufrieden

13. Wann und wie oft nutzen Sie Sonar? (Die Frage richtet sich an die JAVA – Entwickler/innen)

14. Welche Ausgaben oder Aussagen von Sonar sind für Sie selbst oder als Team relevant und werden für die Verbesserung der Software-Qualität bzw. des Qualitätssicherungsprozesses verwendet? (Die Frage richtet sich an die JAVA – Entwickler/innen)

15. Wie gut lässt sich die frühzeitige Erkennung von Fehlern mit den eingesetzten Tools ermöglichen? (Die Frage richtet sich an die JAVA – Entwickler/innen)

- ○ sehr gut
- ○ gut
- ○ schlecht
- ○ mangelhaft

16. Erleichtert sich die Wartung durch die eingesetzten Tools für Software Qualitätsanalyse/Statische Codeanalyse? (Die Frage richtet sich an die JAVA – Entwickler/innen)

- ○ ja
- ○ nein
- ○ weiß nicht

17. Welche Ausgaben hätten Sie sich von solch einem Tool gewünscht bzw. was könnte Ihnen hilfreich sein, um Ihre Arbeit bezüglich die Qualität und Wartung zu erleichtern? (Die Frage richtet sich an die C und JAVA – Entwickler/innen)

18. Glauben Sie, dass die Einführung eines neuen Werkzeuges zur Verbesserung der Software-Qualität beitragen wird? (falls nein : bitte begründen)

- ja
- nein
- weiß nicht

19. Ist die aktuelle Software – Qualitätsanalyse bei „XYZ“ AG hilfreich?

- ja
- nein
- weiß nicht

20. Wo sehen Sie Verbesserungsmöglichkeiten oder den größten Handlungsbedarf in dem Software – Qualitätssicherungsprozess(z. B. Test, Development, Dokumentation) der „XYZ“ AG ?

21. Wo ihre Meinung nach sollten Verbesserungen in Hinblick auf Software-Qualitätsanalyse eingeführt werden (Warum?)

22. Was tun Sie selbst, um den Software – Qualitätsprozess zu verbessern?

23. Was tun Sie, wenn der Qualitätssicherungsprozess nicht mehr weiter hilft?

B. Anhang - Interviews

B.1. Interview - Qualitätssicherungsprozess

1. Wie zufrieden sind Sie mit der Qualitätssicherung im Unternehmen?

2. Welche Methoden des Software-Engineerings werden in Ihrem Unternehmen eingesetzt?

3. Wird die Qualität der Software gemessen? Falls ja, welche Metriken werden eingesetzt?

4. Werden Qualitätsziele bei der Softwareentwicklung verfolgt?

5. Was unternehmen Sie selbst, um qualitativ hochwertige Software zu entwickeln?

6. Kennen Sie die Architektur der von Ihnen entwickelten Software?

7. Sind Sie mit der Wartung von Software vertraut?

8. Welche Probleme tauchen bei der Wartung Ihrer Software typischerweise auf?

9. Wo sehen Sie den größten Handlungsbedarf in der Qualitätssicherung?

B.2. Interview - Werkzeugevaluierung

1. Was halten Sie persönlich, losgelöst von der Entwicklung in Ihrem Unternehmen, von ConQAT?

2. Welche von den Ergebnissen von ConQAT sind hilfreich oder könnten hilfreich für sie in der Zukunft sein?

3. Wie sollten die Ergebnisse von ConQAT, ihrer Meinung nach, kommuniziert werden?

4. Haben Sie explizit Zeit die Qualitätsdefizite, die von ConQAT entdeckt wurden, zu verbessern?

5. Haben Sie zusammen mit dem Teamkollegen die Ergebnisse von ConQAT angeschaut und diskutiert?

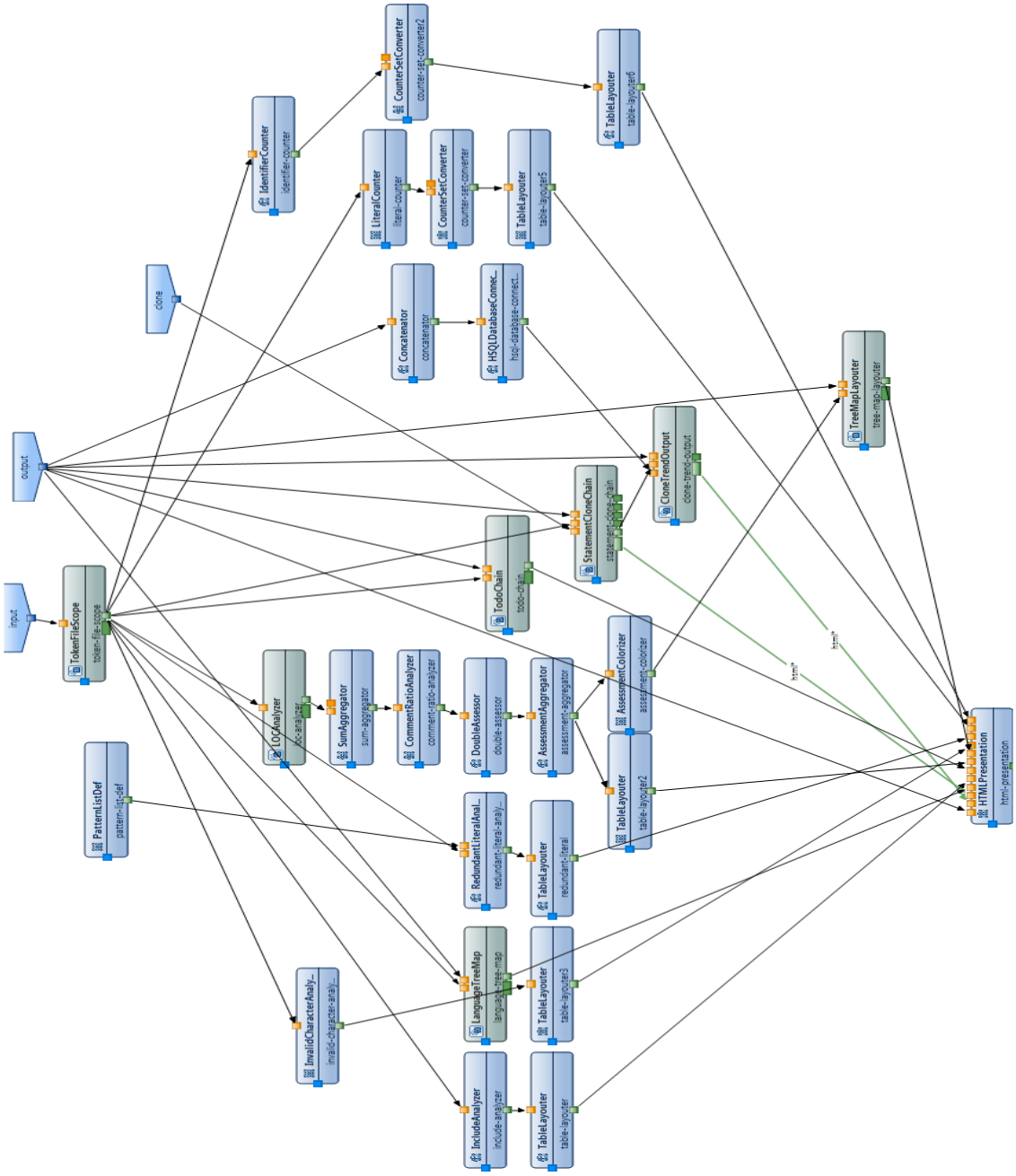
6. Falls die Ergebnisse nicht analysiert worden sind: Wieso nicht?

7. Falls die Ergebnisse analysiert worden sind, welche Erkenntnisse konnten gewonnen werden?

8. Kann ConQAT in der Zukunft hilfreich sein, für die Wartbarkeit der Software?

9. Wie könnte ConQAT am besten in den Entwicklungsprozess integriert werden?

C. Anhang - Analyseblock



Literaturverzeichnis

[91201] I. 9126-1. *Software Engineering Product quality Part1: Quality model*. ISO/IEC, 2001. 11

[Bak95] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In L. Wills, P. Newcomb, E. Chikofsky, Herausgeber, *In Proceedings of the Working Conference on Reverse Engineering*, S. 86–95. IEEE Press, Los Alamitos, California, 1995. 52

[Bal08] H. Balzert. *Lehrbuch der Softwaretechnik -Softwaremanagement*. Spektrum Akademischer Verlag, 2008. 9

[BBK+78] B. Boehm, J. Brown, H. Kaspar, M. Lipow, G. Macleod, M. Merrit. *Characteristics of Software Quality*. North-Holland, 1978. 12

[BCR94] V. Basili, G. Caldiera, D. Rombach. *The Goal Question Metric Approach: Encyclopedia of Software Engineering*. John Wiley & Sons, New York, 1994. 18

[Brc11] R. Brcina. *Zielorientierte Erkennung und Behebung von Qualitätsdefiziten in Software-Systemen am Beispiel der Weiterentwicklungsfähigkeit.* Dissertation, Ilmenau University of Technology, 2011. 11

[BSB08] C. Bommer, M. Spindler, V. Barr. *Softwarewartung: Grundlagen,Management und Wartungstechniken*. dpunkt.verlag, 2008. 10 11

[CM78] J. Cavano, J. McCall. A framework for the measurement of software quality. *In Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, 6:133–139, 1978. 12

[Dei09] F. Deißenböck. *Continuous Quality Control of long lived Software Systems*. Dissertation, Technische Universität München, 2009. 6, 13 42

[Dei10] F. Deissenboeck. *Kontinuierliches Qualitäts-Controling langlebige Softwaresysteme*. Institut für Informatik, TU München, 2010. 1, 2, 38, 41 47

[DFH+10] F. Deißenböck, M. Feilkas, L. Heinemann, B. Humme, E. Jürgen. *ConQAT Book, v2.7*. Technische Universität München, Institut für Informatik, Software & Systems Engineering, 2010. 47, 53 70

[DH11] F. Deissenboeck, B. Hummel. Kontinuierliches Qualitäts-Controlling: Mittel gegen den Qualitätsverfall in der Softwarewartung. *OBJEKTspektrum*, 5:34–38, 2011. 1, 38, 40, 41 44

[DHJ08] F. Deissenboeck, B. Hummel, E. Jürgens. ConQAT - Ein Toolkit zur kontinuierlichen Qualitätsbewertung. In K. Herrmann, B. Brügge, Herausgeber, *Software Engineering*, Band 121, S. 55. 2008. 37

[DJH+08] F. Deissenboeck, E. Jürgens, B. Hummel, S. Wagner, B. M. y Parareda, M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008. 49

[DP06a] F. Deißenböck, M. Pizk. Projekt PQL-Qualitätsmodel. Technischer Bericht, Technische Universität München - Institut für Informatik, 2006. 65

[DP06b] F. Deissenboeck, M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006. 65

[DS06] F. Deissenboeck, T. Seifert. Kontinuierliche Qualitätsüberwachung mit CONQAT. In *GI Jahrestagung (2)'06*, S. 118–125. 2006. 44, 47 49

[FBBO02] M. Fowler, K. Beck, J. Brant, W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, Amsterdam, 2002. 19 64

[FH10] P. Fischer, P. Hofer. *Lexikon der Informatik*. Springer Verlag, Berlin, 2010. 67

[GGIW12] M. Gleirscher, D. Golubitskiy, M. Irlbeck, S. Wagner. On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises. In *SWQD*, S. 14–38. 2012. 64

[Gol11] J. Goll. *Methoden und Architekturen der Softwaretechnik*. Vieweg+Teubner Verlag, Springer Fachmedien Wiesbaden GmbH, 2011. 12, 14, 15 17

[JDHW09] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. Do Code Clones Matter? In *Proc. 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 2009. Available at http://www4.in.tum.de/ wagnerst/publ/icse09.pdf. 52

[Jür12] E. Jürgen. Copy & Paste & Bug. *VKSI Magazin: Qualitätssicherung in der Software-Entwicklung*, 1:13, 2012. 52

[Lig02] P. Liggesmeyer. *Software-Qualität:: Testen, Analysieren und Verifizieren von Software*. Spektrum-Verlag, Heidelberg, Germany, 2002. 8, 13, 17, 18 19

[LL10] J. Ludewig, H. Lichter. *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag Heidelberg, 2010. 1, 5, 6, 7, 10, 11, 14 18

[LR04] M. Lipper, S. Roock. How to Successfully Execute Complex Restructurings. In *Refactorings in Large Software Projects*. 2004. 21

[McC05] S. McConnell. *Code Complete: Deutsche Ausgabe der Second Edition*. Microsoft Press Deutschland, 2005. 10

[MW77] J. McCall, G. Walters. *Factors in Software Quality. The National Technical Information Service (NTIS)*. Springfield, VA, USA, 1977. 12

[RB09] M. Riebisch, S. Bode. Software-Evolvability. *Informatik-Spektrum*, 4:339–343, 2009. 11

[Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer Verlag New York, Inc.,New York,NY, USA, 1997. 11

[Sch07a] K. Schneider. *Abenteuer Software Qualität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. dpunkt.verlag GmbH, 2007. 5, 8, 14, 15, 16, 17 18

[Sch07b] S. Schulze. *Klonerkennung und -klassifizierung zur Unterstützung des Refactoring in Softwaresystemen*. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2007. 52

[Som01] I. Sommerville. *Software Engineering*. 6. Auflage Addison-Wesley, Pearson Studium, Boston, MA, USA, 2001. 1

[SSB10] H. M. Sneed, R. Seidl, M. Baumgartner. *Software in Zahlen: Die Vermessung von Applikationen*. Carl Hanser Verlag München, 2010. 7, 14, 15 17

[Tha00] G. E. Thaller. *Software-Metriken: einsetzen-bewerten-messen*. Verlag Technik, 2000. 15

[Voh10] M. Vohl. Analyse von Bezeichnern. Technischer Bericht, Universität Bonn, 2010. 65

[Wal11] E. Wallmüller. *Software Quality Engineering: Ein Leitfaden für bessere Software-Qualität*. Carl Hanser Verlag München, 2011. 6 8

[WDFJ08] S. Wagner, F. Deißenböck, M. Feilkas, E. Jürgens. Software-Qualitätsmodelle in der Praxis: Erfahrungen mit aktivitätenbasierten Modellen. In *Workshop-Band Software-Qualitätsmodellierung und -bewertung (SQMB '08)*. Technische Universität München, 2008. 12 41

Alle URLs wurden zuletzt am 30.07.2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Mehmed Metuh)