8 Aufbau und Anwendung des Qualitätsindikatorenkatalogs

In diesem Kapitel wird die Anwendung des Qualitätsindikatorenkatalogs beschrieben. Im ersten Abschnitt werden zuerst Aufbau und Inhalt des Katalogs beschrieben. Im Anschluss daran werden mögliche Klassifizierungen der Qualitätsindikatoren und deren jeweiliger Einsatzzweck erläutert. Am Ende des Kapitels wird die Anwendung des Katalogs zur Verbesserung der technischen Qualität von Softwaresystemen anhand zweier Beispiele beschrieben.

8.1 Inhalt des Katalogs

Die Grundidee bei der Ermittlung des Code-Quality-Index ist der Vergleich der Anzahl der Problemmustervorkommen innerhalb des betrachteten Softwaresystems mit der entsprechenden Anzahl in einer Reihe von Referenzsystemen (vgl. Kapitel 4). Um diesen Vergleich durchführen zu können, ist es zwingend nötig, dass der Ermittlung der Kennzahlen identische und für die Anwendung genügend präzise Problemmusterdefinitionen zugrunde liegen. Ebenso müssen die mit dem einzelnen Problemmuster implizierten Auswirkungen auf Qualitätseigenschaften identisch sein. Um dies zu garantieren, wurde der im Kapitel 10 enthaltene Katalog von 52 Qualitätsindikatoren erstellt. Jeder Qualitätsindikator beschreibt ein Problemmuster und dessen Implikationen auf die betrachteten Qualitätseigenschaften sowie die wirtschaftlichen Aspekte Kostenzuwachs und Unmittelbarkeit (vgl. Abschnitt 3.1 und Abschnitt 4.3.1).

Alle den Qualitätsindikatoren innewohnenden Problemmuster können mit Hilfe von Werkzeugen automatisch erkannt werden, ohne dass eine manuelle Konfiguration der Erkennungsstrategien notwendig ist.

8.1.1 Beschreibung von Qualitätsindikatoren

Die Beschreibung der Qualitätsindikatoren erfolgt mit Hilfe eines einheitlichen Schemas. Die Struktur und der Inhalt des verwendeten Schemas werden im Folgenden vorgestellt:

- Definition – Die Definition enthält eine kurze und prägnante Beschreibung des zugrunde liegenden Problemmusters. Sie ist maßgebend für die Erkennung des Problems und definiert ggf. Grenzwerte, ab denen ein Muster zum Problemmuster wird. Die Präzision der Definition muss so hoch sein, dass daraus später automatische Erkennungsstrategien abgeleitet werden können; diese hohe Präzision führt u.U. zu einer schwierigen Lesbarkeit, versucht aber, die in Abschnitt 3.2 in der Praxis immer wieder anzutreffenden Risiken von Softwaremetriken zu kontrastieren, indem alle notwendigen Details für die dort aufgeführten Messschritte beschrieben werden.
- Beschreibung – An dieser Stelle erfolgt eine ausführliche Darlegung der Problemstruktur und eine Begründung, welche negativen grundsätzlichen Implikationen mit dem Problemmuster verbunden sind.
- Qualitätseigenschaften – In diesem Abschnitt werden die negativen Auswirkungen des Problemmusters auf die betrachteten Qualitätseigenschaften – Analysierbarkeit, Modifizierbarkeit, Stabilität, Prüfbarkeit, Austauschbarkeit, Zeitverhalten und Verbrauchsverhalten – beschrieben und prozentual gewichtet (in den Stufen 0%, 25%, 50%, 75% und 100%). Eine Gewichtung mit 0% bedeutet dabei, dass dieses Problemmuster keine negativen Auswirkungen auf die jeweilige Qualitätseigenschaft hat, während die 100%-Wichtung mit einer sehr starken Auswirkung gleichzusetzen ist. Letztlich befördern diese Einflussbeschreibungen das Problemmuster zu einem Qualitätsindikator.
- QBL entsprechend Code-Quality-Index – Dieser Abschnitt beginnt mit einem Kiviat-Diagramm, das die Gewichtungen der Qualitätseigenschaften zeigt. Des Weiteren wird die Relevanz des Qualitätsindikators für die wirtschaftlichen Aspekte Kostenzuwachs und Unmittelbarkeit erläutert und prozentual gewichtet. Analog zu der Gewichtung der Qualitätseigenschaften erfolgt auch hier eine Einstufung in 0%, 25%, 50%, 75% und 100%.

  Den Abschluss bildet die Benennung des aus diesen Parametern ermittelten QBL (zur Ermittlung s. Kapitel 4).
- Technische Parameter – In diesem Abschnitt sind die technischen Aspekte Betrachtungsebene, QBL-Quartile und informelle Hinweise für die Erkennung aufgeführt.

  Bei der *Betrachtungsebene* werden die Granularitätsklassen Technologie, Architektur, Design und Code unterschieden (s. Abschnitt 8.2).

  Der Unterabschnitt *QBL-Grenzwerte* beinhaltet die Kennzahlen, die aus der empirischen Datenbasis (s. Abschnitt 3.3.1) für das Problemmuster ermittelt wurden. Diese Werte bilden die Grundlage für die Ermittlung des QBL eines Softwareprojekts. Die Kennzahlen enthalten Angaben zum Maximum, oberen Quartil, Median, unteren Quartil und Minimum. Das Maximum gibt dabei die größte normierte Anzahl[1] an Verletzungen (Instanzen des Problemmusters) in allen Referenzprojekten an. Gleiches gilt analog für die anderen

Werte. Alle Kennzahlen sind getrennt für die Programmiersprachen Java und C++ angeben.

Im Unterabschnitt *Informelle Hinweise für die Erkennung* ist verbal beschrieben, wie das Problemmuster bei einer statischen Codeanalyse erkannt werden kann und wie die Anzahl der Probleminstanzen bzw. die Verletzungsanzahl ermittelt wird.

8.2 Klassifizierung der Qualitätsindikatoren

Die Qualitätsindikatoren können hinsichtlich der Granularität und des QBL klassifiziert werden. Aus der ersten Klassifizierung lassen sich Hinweise auf die in die Problembehebung involvierten Rollen ableiten, während letztere die Grundlage für die Bestimmung des Code-Quality-Index darstellt. Beide Klassifizierungen werden in den folgenden Unterabschnitten näher erläutert.

Grundsätzlich ist auch eine Klassifizierung entsprechend der jeweils relevanten Programmiersprachen denkbar. Da die hier betrachteten Indikatoren jedoch ausschließlich hinsichtlich ihrer Relevanz für die Sprachen Java und C++ untersucht wurden und alle Indikatoren ohne Ausnahme für beide Sprachen relevant sind, ist diese Klassifizierung hier trivial und wird nicht näher besprochen.

8.2.1 ... nach Granularität

Bei der Klassifikation nach Granularitätsklassen werden die Qualitätsindikatoren bzw. die innewohnenden Problemmuster bestimmten Abstraktionsebenen zugeordnet.

Technologie
Architektur
Design
Code

Abb. 8–1 *Abstraktionsebenen von Qualitätsindikatoren*

Die oberste (grobgranularste) Abstraktionsebene bildet die Technologieebene. Dieser Ebene werden Qualitätsindikatoren zugeordnet, die systemweite Merkmale untersuchen. Die Systemgrenze ist hierbei durch die betrachtete Technologie, also Java bzw. C++, definiert. Ein Beispiel für einen Qualitätsindikator der Technologieebene ist der Qualitätsindikator »Duplizierter Code«.

1. Um die Vergleichbarkeit der ermittelten Anzahl an Verletzungen zu erreichen, sind die Werte über LOC normiert (s. Abschnitt 3.3.1).

Die nächste Granularitätsklasse repräsentiert die Architekturebene. Dieser Ebene zugeordnete Qualitätsindikatoren beziehen sich auf die technische Architektur der Software, die in der Regel die erste Dekomposition des Gesamtsystems in Teile beschreibt. Hierzu gehören zum Beispiel Qualitätsindikatoren, die Merkmale von Paketen fokussieren, wie »Paketchen«.

Die unter der Architekturebene angeordnete Designebene stellt die nächste Abstraktionsebene dar. Dieser werden Qualitätsindikatoren zugeordnet, die im Zusammenhang mit typisch objektorientierten Aspekten, wie Vererbung oder Polymorphie, stehen und die in der Regel Klassen, deren inneren Aufbau sowie deren Interaktion fokussieren.

Die unterste und feingranularste Ebene bildet die Codeebene. Die dieser Ebene zugeordneten Qualitätsindikatoren betreffen Codemerkmale wie Exception-Handler oder nicht erreichbaren Code.

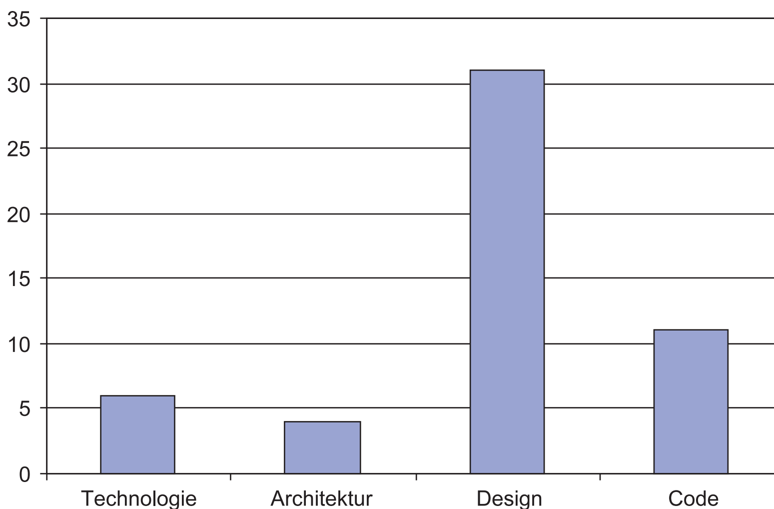

Abb. 8–2 *Verteilung der 52 Qualitätsindikatoren auf die Abstraktionsebenen*

Die beschriebene Klassifizierung ordnet jeden Qualitätsindikator einer Abstraktionsebene zu. Andererseits können jeder Abstraktionsebene aber auch Rollen zugeordnet werden. Mit Hilfe dieser Klassifizierung lassen sich somit jedem Qualitätsindikator die Rollen zuordnen, die schwerpunktmäßig in die Problembehebung involviert sind. So liegt der Fokus der Rolle »Entwickler« hauptsächlich auf den Qualitätsindikatoren der Codeebene, während sich der Architekt eher für die Qualitätsindikatoren der Architektur- und Technologieebene verantwortlich zeigt.

Weiterhin ist die Klassifizierung bei der Auswahl der Analysewerkzeuge dienlich. Um Qualitätsindikatoren der Technologieebene analysieren zu können, muss das Analysewerkzeug ein Systemmodell des gesamten analysierten Systems verwenden; zur Analyse von Qualitätsindikatoren auf Codeebene genügt dage-

gen ein Modell mit geringerem Umfang, z.B. nur der Artefakte innerhalb einer einzelnen Datei.

8.2.2 ... nach QBL

Jeder Qualitätsindikator kann einem QBL zugeordnet werden. Die Zuordnung resultiert aus der Relevanz des zugrunde liegenden Problemmusters für die betrachteten Qualitätseigenschaften sowie aus den Auswirkungen hinsichtlich der Wirtschaftlichkeitsfaktoren Kostenzuwachs und Unmittelbarkeit (vgl. Kapitel 4). Die Verteilung der Qualitätsindikatoren aus dem Katalog auf die relevanten QBL ist in Abbildung 8–3 dargestellt[2]. Eine Auflistung der Zuordnung von Qualitätsindikatoren und QBL kann den Abbildungen im Abschnitt 4.4 entnommen werden.

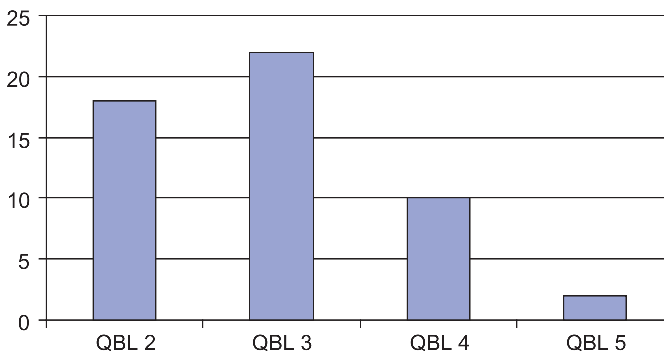

Abb. 8–3 *Verteilung der Qualitätsindikatoren entsprechend ihres QBL*

Wie im Abschnitt 4.4 dargelegt, fokussiert jeder Quality-Benchmark-Level bestimmte Qualitätsaspekte. Der Fokus von QBL 2 liegt z.B. auf den Qualitätseigenschaften Stabilität und Analysierbarkeit. Die Zuordnung von Qualitätsindikatoren und QBL bildet somit eine Grundlage für die Bestimmung des Code-Quality-Index. Diese Klassifizierung stellt eine Form der beiden im Abschnitt 4.3 beschriebenen Zertifizierungsverschärfungen dar.

8.3 Anwendung des Katalogs

Im Folgenden wird die Anwendung des Qualitätsindikatorenkatalogs zur Verbesserung der technischen Qualität von Softwaresystemen anhand eines Qualitätsindikators für die Java-Open-Source-Bibliothek OpenJGraph erläutert. Im darauf

2. Um Fehlinterpretationen vorzubeugen, sei an dieser Stelle nochmals darauf hingewiesen: Ein höherer QBL ist nicht »nur« mit weiteren Qualitätsindikatoren gekoppelt, sondern auch mit einer Verschärfung der Schwellwerte für bereits relevante Indikatoren (vgl. Kapitel 4).

folgenden Abschnitt werden die Ergebnisse einer Code-Quality-Index-Analyse für OpenJGraph und das in C++ geschriebene Werkzeug Doxygen dargelegt und für beide Beispiele eine Bestimmung des Code-Quality-Index vorgenommen.

8.3.1 Analyse des Qualitätsindikators »lange Parameterliste«

Die Erläuterungen zur Anwendung des Qualitätsindikatorenkatalogs erfolgen im Weiteren anhand des Qualitätsindikators »lange Parameterliste«. Dieser Qualitätsindikator ist entsprechend der in Abschnitt 8.2 vorgestellten Klassifizierungen der Granularität Codeebene und dem QBL 2 zugeordnet.

Als Beispielsystem für die Erläuterung dient die Open-Source-Bibliothek »OpenJGraph«. OpenJGraph ist eine Java-Bibliothek zur Darstellung und Manipulation von Graphen [URLOpenJGraph]. Die Bibliothek unterstützt verschiedene Grapharten, z.B. gerichtete und ungerichtete Graphen. Weiterhin enthält sie grundlegende Graphalgorithmen und ermöglicht die Visualisierung von Graphen sowie die interaktive Manipulation der Graphen durch den Nutzer, z.B. das Einfügen und Entfernen von Knoten bzw. Kanten.

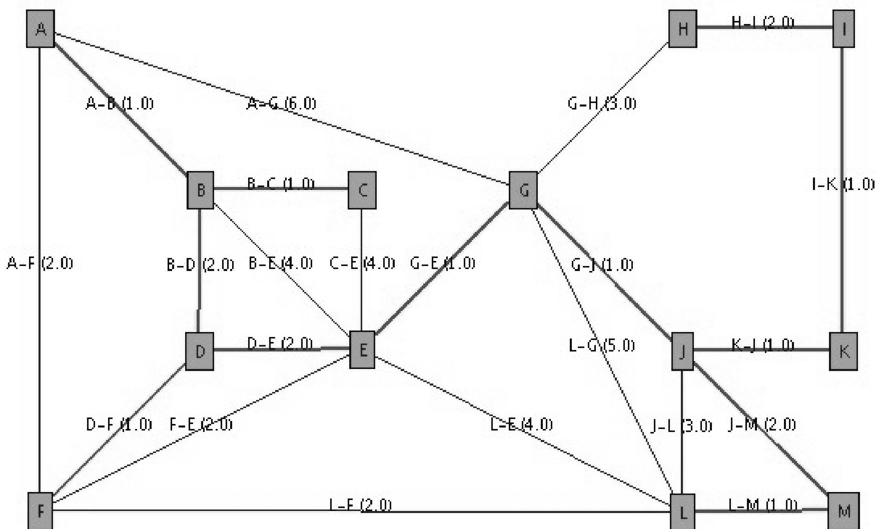

Abb. 8–4 *Screenshot einer OpenJGraph-Applikation*

Entwickelt wurde die Bibliothek über einen Zeitraum von mehreren Jahren. Die letzte veröffentlichte Version 0.9.2 aus dem Jahr 2002 ist Gegenstand der folgenden Analyse.

Das System besitzt 21.157 Quellcodezeilen, 40 Interfaces und 180 Klassen (s. Tab. 8–1). Es handelt sich demnach dabei um ein kleines System, was jedoch die Anwendung des Qualitätsindikatorenkatalogs nicht behindert.

Artefakte	Anzahl
Pakete	13
Dateien	173
Klassen	180
Interfaces	40
Methoden	1.078
Attribute	458
LOC	21.157

Tab. 8–1 *Systemübersicht OpenJGraph*

Die Analyse des Qualitätsindikators erfolgt mit Hilfe von Checkstyle. Checkstyle ist ein Open-Source-Werkzeug zur Überprüfung von Kodierungsrichtlinien für Java-Quellcode [URLCheckstyle]. Neben der Prüfung von Layoutrichtlinien ermöglicht Checkstyle auch das Auffinden von Designproblemen, dupliziertem Code und komplexen Strukturen. Insgesamt beinhaltet Checkstyle über 120 Tests (Checks), die um eigene Tests erweitert werden können. Schwerpunktmäßig fokussiert Checkstyle die Codeebene, ermöglicht aber insbesondere auch die Analyse von Codeduplikaten (Technologieebene).

Die Anpassung von Checkstyle an die zu prüfenden Regeln erfolgt in einer XML-Datei. In dieser werden die durchzuführenden Tests aktiviert und konfiguriert. Nachfolgend ist ein Beispiel für die Anpassung eines Tests angegeben. Gemeldet werden sollen alle Quelltextzeilen, an deren Ende ein oder mehrere Leerzeichen stehen:

```
<!-- No Trailing Spaces -->
<module name="GenericIllegalRegexp">
  <property name="format" value="\s+$"/>
  <property name="message" value="Line has trailing spaces."/>
</module>
```

Konfigurationsdateien sind richtlinienspezifisch und nicht projektspezifisch. Eine Konfigurationsdatei für unternehmensspezifische Richtlinien kann somit – einmal erstellt – auf alle Projekte angewandt werden, die diesen Richtlinien unterliegen. Für eine Überprüfung gemäß den Kodierungsrichtlinien von Sun [URLSun] oder OpenORB [URLOpenORB] liegen der Checkstyle-Distribution bereits entsprechende Konfigurationsdateien bei.

Checkstyle kann entweder in ein Ant-Skript [URLAnt] integriert oder über die Kommandozeile gestartet werden. Die Ergebnisse der Tests können als Text oder in Form einer XML-Datei ausgegeben werden.

Ist *Ant* Bestandteil der Entwicklungsumgebung, so kann die Richtlinienprüfung in den normalen Arbeitsprozess der Entwickler und/oder in den »nightly build« integriert werden. Der Start über die Kommandozeile bietet sich dagegen

für die Integration von Checkstyle in ein Portal (s. Abschnitt 7.2) oder bei der Verwendung einer Build-Umgebung ohne Checkstyle-Unterstützung an.

Mit Checkstyle wurde an dieser Stelle bewusst ein weiteres Werkzeug für eine Code-Quality-Index-Analyse verwendet. Es wird von vielen Java-Entwicklern eingesetzt, lässt sich einfach in bestehende Entwicklungsumgebungen oder Portale (vgl. Kapitel 7) integrieren und deckt in der Standardauslieferung bereits einige die Codeebene adressierende Qualitätsindikatoren korrekt ab. Nicht zuletzt demonstriert die Möglichkeit der Verwendung eines weiteren Werkzeugs die Werkzeugunabhängigkeit des Code-Quality-Index.

Informationen des Qualitätsindikatorenkatalogs

Im Qualitätsindikatorenkatalog ist der Qualitätsindikator »lange Parameterliste« wie folgt definiert: »*Eine lange Parameterliste liegt vor, wenn eine Methode (inkl. Struktoren) mehr als 7 Übergabe-Parameter besitzt. Optionale Parameter werden hierbei mit betrachtet, variable Parameterlisten zählen als einzelner Parameter. Nicht betrachtet werden Methoden, die eine Methode aus einer externen Bibliothek überschreiben oder implementieren.*«

Das zugrunde liegende Problemmuster wird in der dazugehörigen Beschreibung näher erläutert. So sind »*Methoden mit sehr vielen Parametern [..] nur schwer zu verwenden, zu verstehen und sehr änderungsanfällig*«. Wie sich bereits aus dieser Beschreibung entnehmen lässt, hat das Problemmuster Implikationen auf die Qualitätseigenschaften Analysierbarkeit, Modifizierbarkeit, Prüfbarkeit und Änderbarkeit. Die Relevanz bezüglich dieser Qualitätseigenschaften ist dementsprechend im Katalog auch mit 100% angegeben. Weitere, wenn auch geringere Auswirkungen bestehen hinsichtlich der Qualitätseigenschaften Stabilität und Zeitverhalten. In Abbildung 8–5 sind die genannten Relevanzen in einem Kiviat-Diagramm dargestellt.

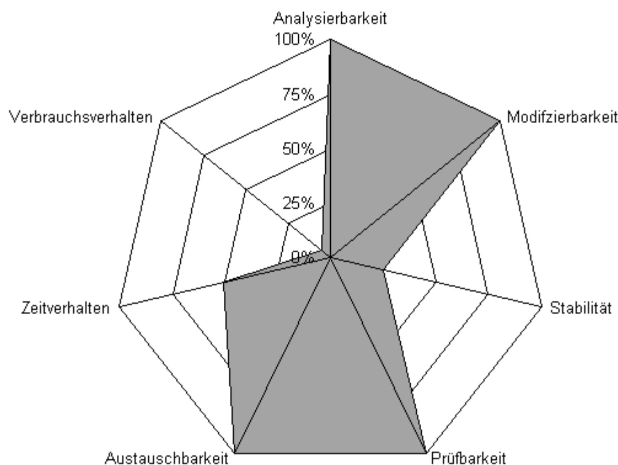

Abb. 8–5 *Relevanz des Problemmusters »lange Parameterliste« bzgl. der Qualitätseigenschaften*

Ergebnisse der Checkstyle-Analyse

Die Anzahl der Parameter der Methoden lässt sich mit dem Checkstyle-eigenen Test *ParameterNumber* überprüfen. Die Parameter des Tests bestimmen die Anzahl der zulässigen Methodenparameter sowie die zu prüfenden Deklarationen (nur Methoden und/oder nur Konstruktoren). Nicht konfiguriert werden kann bei diesem Open-Source-Werkzeug, dass bei der Überprüfung Methoden ignoriert werden sollen, »*die eine Methode aus einer externen Bibliothek überschreiben oder implementieren*«. Die Ergebnismenge kann folglich mehr Verletzungen enthalten, als gemäß der Problemmusterdefinition vorhanden sind. Bei den Falschmeldungen handelt es sich in der Terminologie des diagnostischen Fehlers dann um »falsch positive«.

Nachfolgend ist eine Checkstyle-Konfigurationsdatei für die Prüfung des Problemmusters »lange Parameterliste« angegeben:

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
   "-//Puppy Crawl//DTD Check Configuration 1.1//EN"
   "http://www.puppycrawl.com/dtds/configuration_1_1.dtd">
<module name="Checker">
  <module name="TreeWalker">
    <!-- Avoid long parameter lists -->
    <module name="ParameterNumber">
      <!-- max 7 parameter allowed -->
      <property name="max" value="7"/>
      <!-- check methods and constructors -->
      <property name="tokens" value="METHOD_DEF, CTOR_DEF"/>
    </module>
  </module>
</module>
```

Checkstyle ermittelt im OpenJGraph-Quellcode zwei Verletzungen für den *ParameterNumber*-Test:

```
Starting audit...
D:\ojg\salvo\jesus\graph\visual\layout\AbstractGridLayout.java:681:17:
Mehr als 7 Parameter.
D:\ojg\salvo\jesus\graph\visual\layout\AbstractGridLayout.java:792:17:
Mehr als 7 Parameter.
Audit done.
```

Es handelt sich dabei um zwei Varianten der Methode `alternateMinimalEdgePlacement`. Die nachfolgend dargestellten Quellcodeausschnitte sind unverändert den Orginaldateien entnommen. Bereits der Umfang der Methodenköpfe (mehrere Zeilen) weist dabei die erschwerte Erfassbarkeit als ein Problem von *langen Parameterlisten* aus.

```
681 private Point alternateMinimalEdgePlacement(
682   VisualVertex anchorVertex, VisualVertex adjacentVertex,
683   Point anchorPoint, Point adjacentPoint, List finalplaced,
684   Grid grid, int preferredx, int preferredy )
...
792 private Point alternateMinimalEdgePlacement
793 (
794   VisualVertex anchorVertex, VisualVertex adjacentVertex,
795   Point anchorPoint, Point adjacentPoint, List finalplaced,
796   Grid grid, int preferredx, int preferredy, int xtestpoints[],
      int ytestpoints[] )
```

Die genaue Parameteranzahl der beiden gefundenen Methoden ist 8 bzw. 10. Checkstyle prüft bei der Ermittlung der Methoden jedoch nicht, ob es sich um »neue« oder bereits in externen Bibliotheken deklarierte und hier »nur« implementierte/überschriebene Methoden handelt. Da letztere laut Definition des Qualitätsindikators von der Betrachtung ausgeschlossen sind, muss dieses bei der Anwendung von Checkstyle manuell überprüft werden. Wie oben ersichtlich, handelt es sich in beiden Fällen um *private* Methoden. Da in Java-Programmen *private* Methoden keine Methoden von Oberklassen/Schnittstellen überschreiben bzw. implementieren können, folgt, dass es sich um echte Fehler handelt.

Im Folgenden werden die Problemstellen näher untersucht, um darauf aufbauend eine konkrete Problembehebung vorzuschlagen.

Qualitätsoptimierung

Das Problemmuster »lange Parameterliste« (vgl. Abschnitt 10.25) kann auf vielfältige Weise behoben werden (s. auch [Fowl99]):

- Entfernen nicht verwendeter (»toter«) Parameter.
  Je länger die Parameterliste, desto höher ist die Wahrscheinlichkeit, dass über die Zeit ein Parameter obsolet wird, innerhalb der Implementierung folglich nicht mehr verwendet wird. Dieser Parameterballast kann – nachdem die Verwendung ausgeschlossen werden konnte – sehr einfach beseitigt werden.
- Ersetzen von Parametern durch Methodenaufrufe.
  Anstatt einen Wert oder ein Objekt zu übergeben, wird die Methode so modifiziert, dass sie sich die notwendigen Daten (über entsprechende Methodenaufrufe) selber beschaffen kann. Dies ist insbesondere dann von Vorteil, wenn einzelne Werte innerhalb der Methode nur unter bestimmten Bedingungen benötigt werden. Bei der Bereitstellung über Methodenparameter müssen diese Werte dagegen immer (vor dem Methodenaufruf) ermittelt werden.
- Übergeben einer Klasse anstatt der darin enthaltenen Daten.
  Oftmals werden an Methoden mehrere Daten ein und desselben Objekts übergeben. Bei diesem Lösungsansatz wird dagegen das Objekt selbst übergeben und die Methode erfragt die benötigten Daten bei Bedarf selbst vom

Objekt. Auch hier lassen sich durch die »Ermittlung bei Bedarf« (lazy loading) Performance-Gewinne erzielen.

- Zusammenfassen von Parametern zu Parameterobjekten.
  Werden Parameter immer wieder zusammen verwendet, so können sie zu einer eigenen Klasse oder einem eigenen Typ zusammengefasst werden.
- Aufteilen der Methode.
  Sofern in den Ausführungspfaden der Methoden disjunkte Parameter(-mengen) verwendet werden, kann die Methode in mehrere Methoden aufgeteilt werden.

Alle vorgeschlagenen Lösungsstrategien sind zunächst nur unter der Bedingung realisierbar, dass es sich bei der Probleminstanz nicht um eine Methode handelt, die eine Methode einer Oberklasse bzw. Schnittstelle überschreibt oder implementiert. Existiert eine solche Deklaration in einer Oberklasse/Schnittstelle des Systems[3], so ist die Behebung wesentlich schwieriger. In einem solchen Fall stellen sowohl die Deklaration als auch alle zugehörigen Redefinitionen/Implementierungen Instanzen des Problemmusters dar. Aufgrund des potenziellen polymorphen Aufrufs der Methoden können die Parameterlisten der Methoden jedoch nicht unabhängig voneinander geändert werden. Stattdessen muss die Modifikation der Parameterliste in der Oberklasse/Schnittstelle erfolgen und in allen implementierenden bzw. überschreibenden Methoden nachgezogen werden. Das gilt analog, wenn die Probleminstanz selbst eine Methode einer Oberklasse/Schnittstelle ist und andere Methoden diese überschreiben/implementieren.

Grundsätzlich setzen alle genannten Lösungsstrategien voraus, dass der Quellcode zur Problembehebung näher untersucht werden muss; die Auswahl der konkreten Lösungsstrategie oder gar eine Kombination kann nur in den seltensten Fällen vollständig automatisiert erfolgen.

Für das OpenJGraph-Beispiel lassen sich zunächst aufgrund der Sichtbarkeit »*private*« der beiden Methoden folgende Implikationen ableiten:

1. Es ist sichergestellt, dass die Methoden keine Oberklassenmethoden überschreiben bzw. Schnittstellenmethoden implementieren[4].
2. Es kann ausgeschlossen werden, dass die Methoden in Unterklassen überschrieben werden.
3. Die Modifikation der Methodensignaturen ist vereinfacht, da private Methoden nur innerhalb der Klasse aufgerufen werden können und somit alle Änderungen nur lokale Anpassungen (innerhalb der umgebenen Klasse) nach sich ziehen.[5]

---

3. Externe Deklarationen von Methoden und die zugehörigen implementierenden bzw. überschreibenden Methoden sind laut Problemmusterdefinition von der Betrachtung ausgeschlossen.
4. Diese Folgerung gilt nur für Java. In C++ ist es durchaus möglich,
   1. Methoden beim Überschreiben zu »privatisieren« und
   2. derartig überschriebene Methoden polymorph aufzurufen.

Die weitere Analyse des Quellcodes ergibt, dass das konkrete Codebeispiel durch eine Kombination der oben aufgeführten Überarbeitungen deutlich verbessert werden kann.

1. In beiden Methoden werden die ersten zwei Parameter (`anchorVertex` und `adjacentVertex`) und in der zweiten Methode zusätzlich der Parameter `adjacentPoint` nicht verwendet. Diese Parameter können daher gefahrlos aus der Parameterliste gestrichen werden (Überarbeitung: Entfernen nicht verwendeter Parameter). Das bedingt eine Anpassung aller fünf Aufrufstellen der beiden Methoden. Darüber hinaus muss selbstverständlich auch die Dokumentation entsprechend modifiziert werden (hier vor allen Dingen die Javadoc-Kommentierung der Methoden).
2. Weiterhin lässt sich aus der Dokumentation der Methoden entnehmen, dass die beiden Parameter `preferredx` und `preferredy` die X- und Y-Koordinaten eines Punktes beschreiben. Die Parameterliste der Methoden lässt sich daher durch die Verwendung eines Objekts der Klasse `java.awt.Point` anstatt der beiden genannten Parameter weiter reduzieren.

Die beschriebenen Modifikationen resultieren in den modifizierten Methodensignaturen:

```
681 private Point alternateMinimalEdgePlacement(
682   Point anchorPoint, Point adjacentPoint,
683   List finalplaced, Grid grid,
684   Point preferredPoint)
...
792 private Point alternateMinimalEdgePlacement(
793   Point anchorPoint,
794   List finalplaced, Grid grid,
795   Point preferredPoint,
796   int xtestpoints[], int ytestpoints[])
```

Mit 5 bzw. 6 Methodenparametern erfüllen beide Methoden jetzt zumindest die Forderungen hinsichtlich der maximalen Parameteranzahl.

Bei der Anwendung zusätzlicher Refactoring-Maßnahmen ließe sich die zweite Methode noch weiter vereinfachen: So modifiziert die Methode zunächst in Abhängigkeit von den Koordinaten der Punkte `anchorPoint` und `preferredPoint` die Werte der Arrays `xtestpoints` und `ytestpoints`. Dieser Teil könnte aus der Methode extrahiert werden, z.B. in eine neue Methode `modifyTestPoints`. Nachfolgend ist der ursprüngliche Aufruf der Methode `alternateMinimalEdgePlacement` dargestellt:

5. Mit Hilfe von Reflections besteht gleichwohl die Möglichkeit, private Methoden von außerhalb aufzurufen (s. [Burt03]). Diese Zugriffsarten können jedoch mit Hilfe der statischen Analyse in der Regel nicht erkannt werden und sind daher auch nicht im Fokus des Problemmusters.

```
alternatePlacement = alternateMinimalEdgePlacement(
                         anchorPoint, finalplaced, grid, preferredPoint,
                         new int[] {-1,1,-2,2,-2,2,-1,1},
                         new int[] {1,1,0,0,-2,-2,-3,-3});
```

Nach der Extraktion der Methode `modifiyTestPoints` und der Streichung des Parameters `anchorPoint` – dieser wird nur in `modifyTestPoints` verwendet – sieht der Aufruf der Methoden wie folgt aus:

```
xtestpoints = new int[] {-1,1,-2,2,-2,2,-1,1};
ytestpoints = new int[] {1,1,0,0,-2,-2,-3,-3};
modifyTestPoints(anchorPoint, preferredPoint, xtestpoints, ytestpoints);

alternatePlacement = alternateMinimalEdgePlacement(
                         finalplaced, grid, preferredPoint,
                         xtestpoints, ytestpoints);
```

Mit den beschriebenen Modifikationen ließe sich somit die Parameterliste der zweiten `alternateMinimalEdgePlacement` Methode von Anfangs 10 auf 5 Parameter verkürzen.

Einstufung nach QBL

Abschließend soll ein Vergleich des OpenJGraph-Systems mit dem Industriestandard bezogen auf den Qualitätsindikator »lange Parameterliste« erfolgen.

Für den Qualitätsindikator »lange Parameterliste« ergeben sich die in Abbildung 8–6 dargestellten Kennwerte (Verletzungen pro 1.000 LOC).

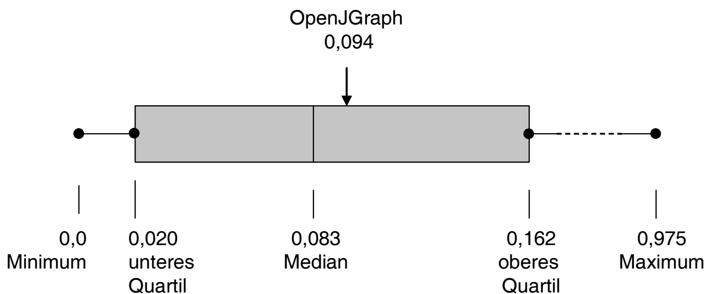

Abb. 8–6 *Boxplot der Kennwerte für den Qualitätsindikator »lange Parameterliste«. Die Zahlen entsprechen der Anzahl der Verletzungen pro 1.000 LOC.*

Im Vergleich mit dem QBL-Repository ist OpenJGraph (ohne die oben beschriebenen Modifikationen) daher beim Qualitätsindikator »lange Parameterliste« schlechter als 50% der Referenzprojekte. Das Projekt besitzt damit, ausschließlich bezogen auf »lange Parameterliste« den QBL 2. Nach der Durchführung der oben beschriebenen Verbesserungen, in deren Folge keine Verletzungen mehr vorliegen, würde das System jedoch bzgl. dieses Qualitätsindikators QBL 5 erreichen.

8.3.2 Code-Quality-Index-Analyse

Bei der Code-Quality-Index-Analyse werden alle oder auch nur ausgewählte Qualitätsindikatoren betrachtet. Diese Art der Verwendung des Qualitätsindikatorenkatalogs kann z.B. für eine Standortbestimmung (Vergleich mit dem Industriestandard) benutzt werden. Im Gegensatz dazu werden bei der Bestimmung des QBL grundsätzlich nur die im betrachteten QBL relevanten Indikatoren untersucht.

Unabhängig von der Anzahl der betrachteten Qualitätsindikatoren ist jedoch das Vorhandensein eines Systemmodells. Bevor das Systemmodell erstellt und die eigentliche Analyse durchgeführt werden kann, sind zwei vorbereitende Schritte notwendig. Zum einen muss das Quellcodeverzeichnis untersucht und ggf. bereinigt werden. Ziel dieser Untersuchung ist es, all jene Teile zu entfernen, die das Analyseergebnis verfälschen könnten. Dazu gehören u.a. (zur Begründung s. Abschnitt 4.2):

- Dateien, die Testcode enthalten.
- Dateien, die ausschließlich und vollständig generierten Code enthalten. Das bloße Enthaltensein von generiertem Code in einer Quellcodedatei ist kein ausreichendes Kriterium für den Ausschluss einer Datei von der Analyse. So fügt z.B. die IDE aus dem Microsofts Visual Studio generierten Code in GUI-Klassen ein.
- Dateien, die Beispielcode beinhalten.
- Dateien, die zu Dokumentationszwecken beigefügt wurden, wie veralteter Code[6] oder fremder Quellcode (z.B. von Open-Source-Bibliotheken).
- Verzeichnisse, die nach der Streichung oben genannter Dateien leer sind.

Im zweiten Schritt müssen die im System verwendeten Bibliotheken identifiziert und lokalisiert werden. Insbesondere sind dabei jene Bibliotheken von Interesse, deren Schnittstellen bzw. Klassen im zu untersuchenden System implementiert respektive durch Unterklassen spezialisiert werden. Die Kenntnis dieser Vererbungs-/Implementierungsbeziehungen ist zum Beispiel für die Analyse des Problemmusters »lange Parameterliste« notwendig (s.o.).

Die Verzeichnisstruktur des Beispielsystems OpenJGraph ist in Abbildung 8–7 dargestellt. Entsprechend der oben aufgeführten Ausschlussliste sind die Unterverzeichnisse *examples* und *test* potenziell zu streichende Quellcodeteile. In beiden Fällen kann dies durch eine Prüfung der enthaltenen Quellcodedateien bestätigt werden.

6. Ausdrücklich sind damit nur solche Dateien gemeint, von denen *zuverlässig bekannt* ist, dass sie veraltet sind, nicht benutzt werden und lediglich der Dokumentation dienen.

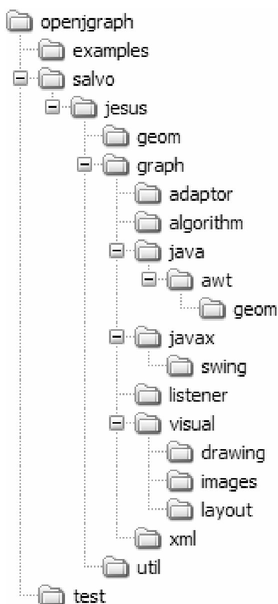

Abb. 8–7 *Verzeichnisstruktur der OpenJGraph-Quellen*

Informationen über die in der OpenJGraph-Bibliothek verwendeten externen Bibliotheken können den dem Quellcode beiliegenden Compile-Skript entnommen werden. Folgende externe Bibliotheken werden laut diesem von OpenJGraph verwendet:

- log4j-core.jar
- log4j.jar
- xerces.jar
- junit.jar

Nach der Modifikation des Quellcodeverzeichnisses – Entfernen der Verzeichnisse `examples` und `test` – sowie der Lokalisierung und Bereitstellung der externen Bibliotheken kann das Projekt analysiert und das zugehörige Systemmodell erstellt werden.

Nachdem das Systemmodell erstellt wurde, können die Analysen der Qualitätsindikatoren erfolgen. Je nach Zielsetzung werden entweder für eine Standortbestimmung alle Indikatoren analysiert (Code-Quality-Index-Analyse), oder es werden alle im gewünschten (zu erreichenden) QBL relevanten Indikatoren analysiert (QBL-Bestimmung).

Im Folgenden sind für die beiden Beispielsysteme OpenJGraph und Doxygen jeweils die Ergebnisse einer vollständigen Code-Quality-Index-Analyse (Betrachtung aller Indikatoren) sowie eine QBL-Bestimmung aufgeführt.

Ergebnisse: OpenJGraph

In Tabelle 8-2 sind zunächst die absoluten Anzahlen der Verletzungen aller Qualitätsindiktoren für das System OpenJGraph aufgeführt. Bereits diese Zahlen können bei einer Trendanalyse (Vergleich der Anzahlen mit vorherigen oder nachfolgenden Versionen von OpenJGraph) Aufschluss über die qualitative Entwicklung des Systems geben.

Qualitätsindikator	Anzahl der Verletzungen
Allgemeine Parameter	39
Attributüberdeckung	2
Ausgeschlagenes Erbe (Implementierung)	21
Ausgeschlagenes Erbe (Schnittstelle)	26
Datenkapselaufbruch	0
Duplizierter Code	0
Falsche Namenslänge	2
Generationskonflikt	0
Gottdatei	0
Gottklasse (Attribut)	0
Gottklasse (Methode)	0
Gottmethode	0
Gottpaket	1
Halbherzige Operationen	11
Heimliche Verwandtschaft	16
Identitätsspaltung	0
Importchaos	390
Importlüge	65
Informelle Dokumentation	369
Interface-Bypass	78
Klässchen	29
Klasseninzest	6
Konstantenregen	0
Labyrinthmethode	6
Lange Parameterliste	2
Maskierende Datei	0
Nachlässige Kommentierung	5.137
Namensfehler	33

Qualitätsindikator	Anzahl der Verletzungen
Objektplacebo (Attribut)	109
Objektplacebo (Methode)	17
Paketchen	3
Pakethierarchieaufbruch	3
Polymorphieplacebo	0
Potenzielle Privatsphäre (Attribut)	23
Potenzielle Privatsphäre (Methode)	21
Pränatale Kommunikation	23
Risikocode	10
Signaturähnliche Klassen	1
Simulierte Polymorphie	14
Späte Abstraktion	2
Tote Attribute	2
Tote Implementierung	0
Tote Methode	5
Überbuchte Datei	7
Unfertiger Code	7
Unvollständige Vererbung (Attribut)	18
Unvollständige Vererbung (Methode)	4
Verbotene Dateiliebe	16
Verbotene Klassenliebe	18
Verbotene Methodenliebe	0
Verbotene Paketliebe	7
Versteckte Konstantheit	12

Tab. 8–2 *Ergebnisse der Problemmusteranalyse für OpenJGraph 0.92*

Im Fokus der Code-Quality-Index-Analyse steht jedoch der Vergleich mit dem Industriestandard. Dafür werden diese Anzahlen mit LOC normiert und mit dem QBL-Repository verglichen.

Das Ergebnis dieses Vergleichs ist im Detail in Abbildung 8–8 dargestellt, während Abbildung 8–9 den Vergleich zusammenfasst. OpenJGraph ist somit bei 17 QualitätsindikaQualitätsindikator:toren besser als 75% der Referenzprojekte, aber bei 15 anderen auch schlechter als 75%. Insbesondere auf letztere sollte daher in einer Refactoringphase im Interesse einer besseren Wartbarkeit der Fokus gelegt werden.

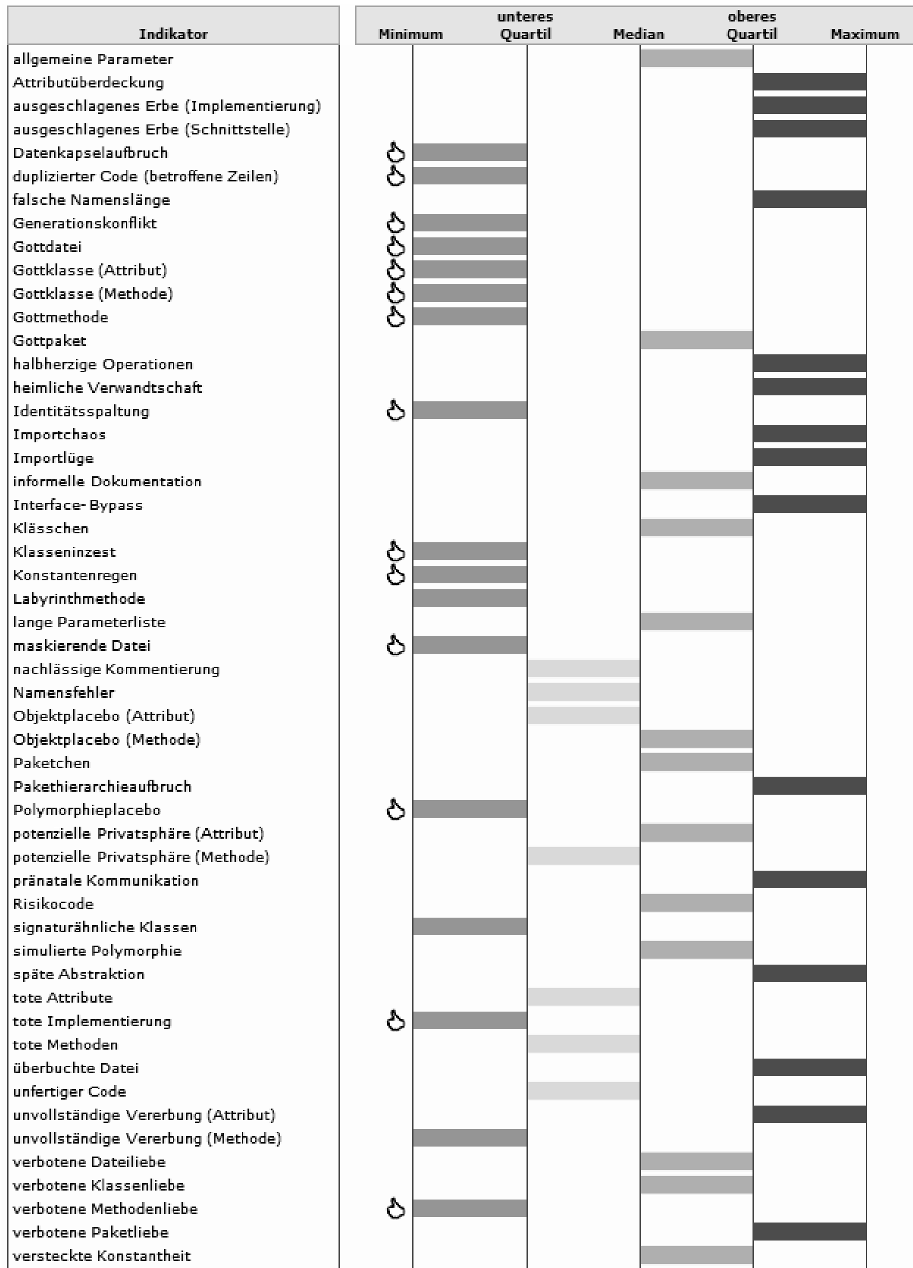

Abb. 8–8 *Vergleich von OpenJGraph mit dem QBL-Repository*

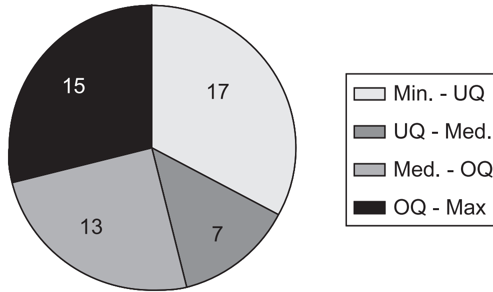

Abb. 8–9 *Zusammenfassung des Vergleichs OpenJGraph und QBL-Repository*

Für die Bestimmung des konkreten QBL wird die Betrachtung auf die im jeweiligen QBL relevanten Qualitätsindikatoren eingeschränkt. Für die QBL-2-Indikatoren ist ein solcher Auszug aus dem Vergleich in Abbildung 8–10 dargestellt. Dementsprechend ist OpenJGraph folgendermaßen einzuordnen:

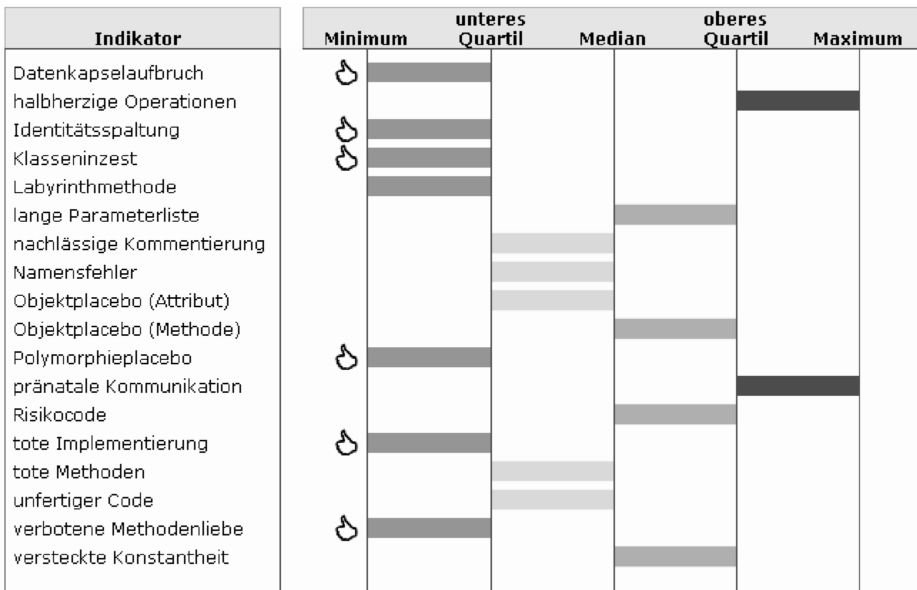

Abb. 8–10 *Ergebnisse des Vergleichs von OpenJGraph mit dem QBL-Repository, nur QBL-2-Indikatoren (Stand 02/2006)*

OpenJGraph ist

- bei den Indikatoren »halbherzige Operationen« und »pränatale Kommunikation« schlechter als 75% der Referenzprojekte,
- bei den Indikatoren »lange Parameterliste«, »Objektplacebo (Methode)«, »Risikocode« und »versteckte Konstantheit« besser als 25%, aber schlechter als die besten 75% der Referenzprojekte,

- bei den Indikatoren »nachlässige Kommentierung«, »Namensfehler«, »Objektplacebo (Attribut)«, »tote Methoden« und »unfertiger Code« besser als 50%, aber schlechter als die besten 25% der Referenzprojekte.
- Bei den restlichen aufgeführten Indikatoren gehört OpenJGraph zu den besten 25% der Referenzprojekte bzw. hat bei den mit der »Hand« markierten Indikatoren die minimale Anzahl von Verletzungen. Bei letzteren wären damit sogar die Kriterien für QBL 5 erfüllt.

Zusammenfassend besitzt OpenJGraph in der analysierten Version QBL 1, d.h. die niedrigste Stufe für übersetzbare Systeme. Zum Erreichen von QBL 2 müssten die Verletzungen der Qualitätsindikatoren »halbherzige Operationen« und »pränatale Kommunikation« zumindest bzgl. ihrer Häufigkeit deutlich reduziert werden.

Ergebnisse: Doxygen

Analog zur Code-Quality-Index-Analyse und der Bestimmung des Code-Quality-Index von OpenJGraph werden im Folgenden die Ergebnisse für Doxygen präsentiert. Doxygen ist ein Open-Source-Dokumentationssystem und vor allem im C++-Umfeld verbreitet [URLDoxygen].

Die für die Analyse verwendete Version 1.4.6 von Doxygen ist in C++ implementiert. Das System besitzt 201.319 Quellcodezeilen und 313 Klassen (s. Tab. 8–3). Mit 32501 Kommentarzeilen ist der Kommentaranteil bei Doxygen im Vergleich zu anderen Systemen sehr gering – dieses zeigt sich im Folgenden auch an den entsprechenden Qualitätsindikatoren »nachlässige Kommentierung« und »informelle Dokumentation«.

Artefakte	Anzahl
Verzeichnisse	1
Dateien	185
Klassen	313
Methoden	9263
Attribute	1.156
LOC	201.319
Kommentarzeilen	32.501

Tab. 8–3 *Systemübersicht Doxygen*

Die vollständige Code-Quality-Index-Analyse und der Vergleich mit dem QBL-Repository ergibt für Doxygen folgendes Ergebnis (s. auch Abb. 8–11 und Abb. 8–12):

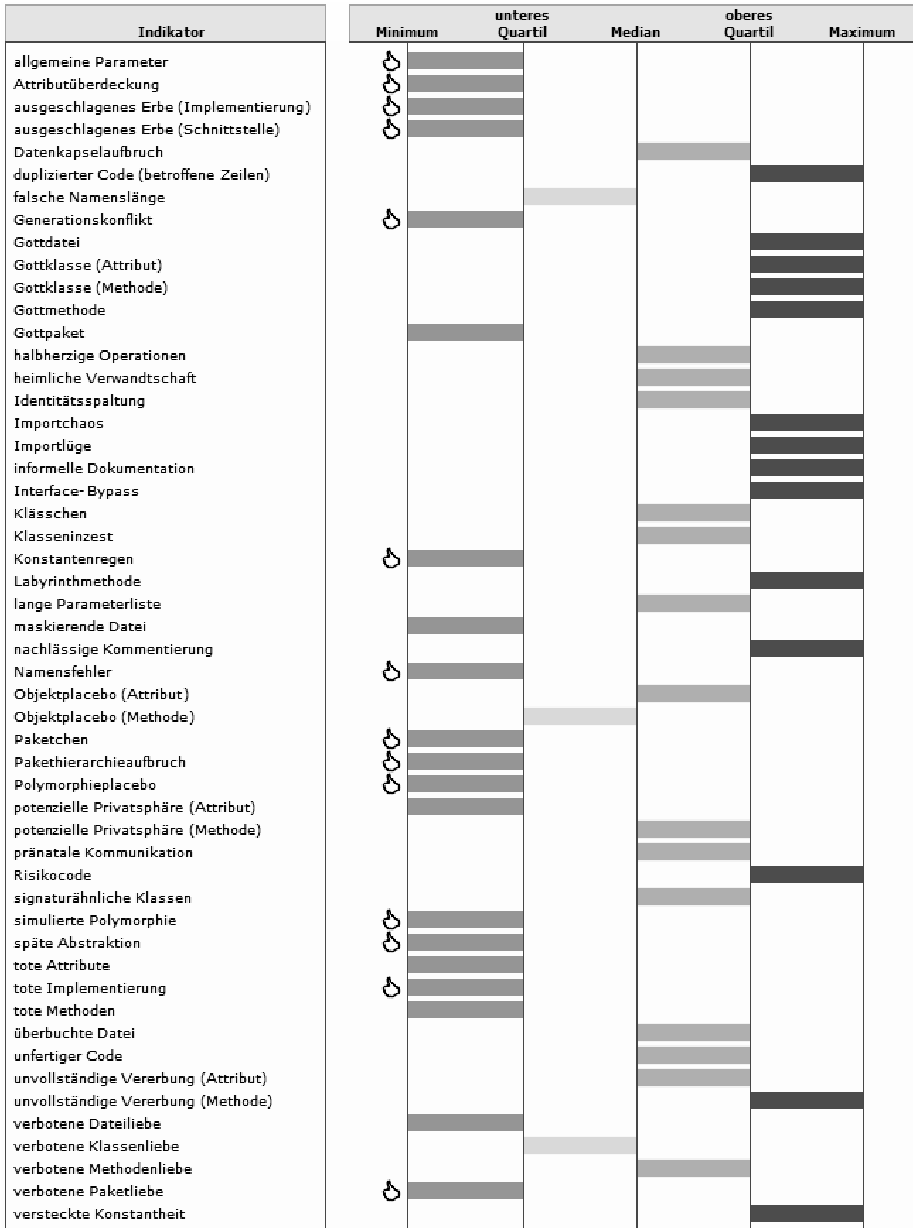

Abb. 8–11 *Vergleich von Doxygen mit dem QBL-Repository*

- Bei 20 Qualitätsindikatoren ist Doxygen besser als 75% der Referenzprojekte. Bei 14 von diesen Indikatoren (mit der »Hand« markiert) ist die Verletzungsanzahl sogar gleich der minimalen Verletzungsanzahl aller Referenzprojekte.
- Bei 3 Qualitätsindikatoren liegt die normierte Verletzungsanzahl von Doxygen im Vergleich zum Repository zwischen dem unteren Quartil und dem Median. Das System ist somit bei diesen Indikatoren besser als 50% der Referenzprojekte.
- Immer noch besser als 25% der Referenzprojekte ist das System bei weiteren 15 Qualitätsindikatoren.
- Schlechter als 75% der Referenzprojekte ist Doxygen schließlich bei 14 Qualitätsindikatoren.

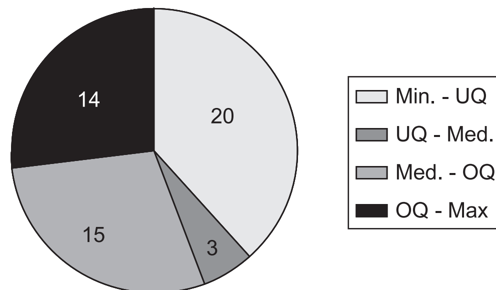

Abb. 8–12 *Zusammenfassung des Vergleichs Doxygen und QBL-Repository*

Zum Abschluss der Betrachtungen von Doxygen soll schließlich noch die Bestimmung des QBL erfolgen. Dafür wird die Menge der betrachteten Qualitätsindikatoren zunächst auf die für QBL 2 relevanten 18 Indikatoren eingeschränkt (s. Abschnitt 4.4.2). Für das Erreichen der Qualitätsstufe QBL 2 darf Doxygen bei keinem dieser 18 Qualitätsindikatoren schlechter als 75% der Referenzprojekte sein. Dieses Kriterium wird jedoch von den Qualitätsindikatoren »Labyrinthmethode«, »nachlässige Kommentierung«, »Risikocode« und »versteckte Konstantheit« nicht erfüllt (vgl. Abb. 8–13), so dass das System nur QBL 1 erreicht. Eine Reduzierung der Verletzungen bei diesen 4 Qualitätsindikatoren wäre eine Grundvoraussetzung für das Erreichen von QBL 2. Insbesondere aufgrund des geringen Kommentaranteils – der umso mehr verwundert, wenn die hinter Doxygen stehende Funktionalität betrachtet wird – wäre dies jedoch mit einem nicht unerheblichen Aufwand verbunden, der sich aber andererseits im Interesse einer guten Wartbarkeit und der zu erwartenden zukünftigen Modifikation und Erweiterungen gerade auch im Open-Source-Bereich auszahlen würde.

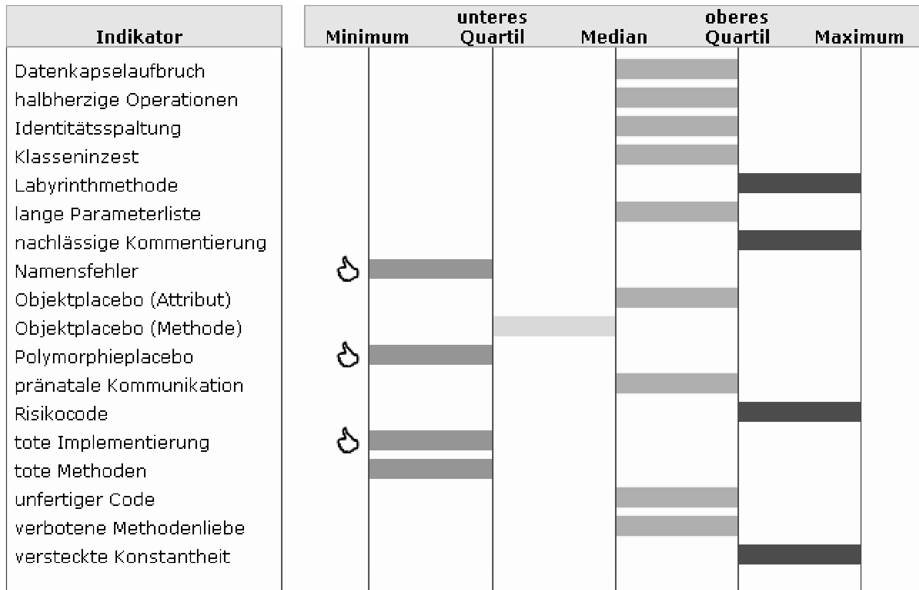

Abb. 8–13 *Status von Doxygen bzgl. der für QBL 2 relevanten Qualitätsindikatoren*