

Diplomarbeit

**Entwicklung eines generischen
Qualitätsmodell-Editors und
Auswertungswerkzeuges**

Autor: Martin Jansen

Vorgelegt der: Fakultät für Mathematik, Informatik
und Naturwissenschaften der Rheinisch-
Westfälischen Technischen Hochschule
Aachen im Januar 2009

Angefertigt am: Lehr- und Forschungsgebiet Informatik 3
Prof. Dr. rer. nat. Horst Lichter

Gutachter: Prof. Dr. rer. nat. Horst Lichter
Prof. Dr.-Ing. Ulrik Schroeder

Betreuer: Dipl.-Inform. Holger Schackmann

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 26.01.2009.

(Martin Jansen)

Inhaltsverzeichnis

1	Einleitung	1
1.1	Aufgabenstellung	2
1.2	Überblick über die Ausarbeitung	3
1.3	Danksagungen	3
2	Grundlagen	5
2.1	Allgemeine Begriffe	5
2.2	Qualität und Qualitätsmodelle	6
2.2.1	Problematik bei der Erfassung von Qualitäten	7
2.2.2	Bidirektionale Qualitätsmodelle	8
2.3	Messungen und Metriken	9
2.4	BugzillaMetrics	10
2.4.1	Änderungsanträge	10
2.4.2	Motivation für BugzillaMetrics	11
2.4.3	Metrik-Spezifikation	11
3	Anforderungen	15
3.1	Einleitung	15
3.1.1	Ziel des Werkzeugs	15
3.1.2	Benutzergruppen	15
3.1.3	Zusammensetzung	17
3.2	Funktionale Anforderungen	18
3.2.1	Qualitätsmodelle	18
3.2.2	Qualitätsmodell-Editor	25
3.2.3	Auswertungswerkzeug	26
3.3	Nichtfunktionale Anforderungen	28
3.3.1	Systemvoraussetzungen	28
3.3.2	Bedienbarkeit	28
3.3.3	Geschwindigkeit	29

3.3.4	Interoperabilität	29
3.3.5	Wartbarkeit	29
4	Softwareentwicklungsprozess	31
4.1	Rahmenbedingungen	31
4.2	Beschreibung des Softwareentwicklungsprozesses	32
4.3	Entwicklungsschritte	36
4.3.1	Ermittlung der Anforderungen	36
4.3.2	Entwurfs- und Implementierungsphase	37
4.3.3	Dokumentation	39
5	Architektur	41
5.1	Allgemeine Architektur	42
5.2	Architektur des Qualitätsmodells-Editors	43
5.2.1	GMF	43
5.2.2	Qualitätsmodelle, -eigenschaften und -indikatoren	46
5.2.3	Wertebereiche von Qualitätsknoten	47
5.2.4	Mess- und Berechnungsvorschriften	48
5.2.5	Frontend	55
5.3	Architektur des Auswertungswerkzeuges	58
5.3.1	Einleitung	59
5.3.2	Das Datenmodell für Auswertungen	60
5.3.3	Messen und Berechnen	72
5.3.4	Messwerkzeuge	74
5.3.5	Algorithmen für Berechnungsvorschriften	78
5.3.6	Frontend	86
5.4	RPC-Schnittstelle für BugzillaMetrics	94
5.4.1	Einleitung	95
5.4.2	Implementierung	96
6	Evaluierung	99
6.1	Qualität der Produkte	99
6.1.1	Funktionalität	99
6.1.2	Zuverlässigkeit	100
6.1.3	Benutzbarkeit	101
6.1.4	Effizient	102
6.1.5	Änderbarkeit	102
6.1.6	Übertragbarkeit	104
6.2	Qualität des Entwicklungsprozesses	105

INHALTSVERZEICHNIS

7 Fazit & Ausblick	107
7.1 Ausblick	108
Literaturverzeichnis	111

Abbildungsverzeichnis

4.1	Integration von Unit Tests und Refactoring in den Entwicklungsablauf	34
4.2	Domänen-Modell des Prototypen	38
5.1	Arbeitsschritte bei MDD mit GMF	45
5.2	Wertebereiche von Qualitätsknoten	48
5.3	Messvorschriften für Qualitätsindikatoren	50
5.4	Architektur der Filterfunktionen	52
5.5	Architektur der komplexen Funktionen	54
5.6	MVC-Architekturmuster in GMF-basierten Anwendungen . .	56
5.7	Struktur der Wizard-Erweiterungen	58
5.8	Generelle Struktur des Auswertungsmodells	61
5.9	Konfiguration einer Auswertung	63
5.10	Konfiguration der Datenquellen	67
5.11	Datenreihen und Werte	69
5.12	Komponente zum Vermessen und Berechnen einer Auswertung	73
5.13	Architektur der Messwerkzeuge	74
5.14	Detailarchitektur des Messwerkzeuges für BugzillaMetrics . .	76
5.15	Grundstruktur der Berechnungs-Komponente	79
5.16	Argumente der Berechnungen	81
5.17	Algorithmen für Berechnungsvorschriften	83
5.18	Das Base Filter Widget von BugzillaMetrics	89
5.19	Das Domänenmodell des Base Filter Widget	91
5.20	Grafische Benutzerschnittstelle für das Base Filter Widget . .	92
5.21	Struktur des XML-RPC-Servers für BugzillaMetrics	97

Kapitel 1

Einleitung

Das Vorhandensein großer Softwareprojekte in einem Unternehmen geht in der Regel einher mit der bewussten Entscheidung für ein wohldefiniertes Prozessmodell zur Steuerung der Entwicklung dieser Projekte. Die Verwendung eines Softwareentwicklungsprozesses ist jedoch kein Selbstzweck, sondern mit dem Ziel verknüpft, qualitativ hochwertige Software in einem definierten Zeitrahmen und mit im Vorfeld planbarem Aufwand zu erstellen. Daher ist eine permanente Überwachung der erzielten Prozessqualität erforderlich, um Verbesserungspotential aufzuzeigen und somit die Erwartungen an den Prozess zu erfüllen. Die Ermittlung der hierfür notwendigen Datenbasis ist oftmals zeit- und kostenintensiv [CVW98]. Dieser Effekt wird durch umfangreiche Projektportfolios noch verstärkt.

Daraus ergibt sich die Motivation, die für die Qualitätssicherung notwendigen Daten aus bereits bestehenden Datenbeständen automatisiert zu ermitteln. Das Open-Source-Softwarepaket *QMetric* [SLLJ09] verfolgt den Ansatz, diese mithilfe flexibel definierbarer Metriken aus Werkzeugen zum Change Request Management oder aus Versionierungssoftware zu ermitteln. Im Kern besteht QMetric aus dem Werkzeug BugzillaMetrics, das einen generischen Auswertungsalgorithmus für diese Metriken bereitstellt, der (trotz des Namens) an verschiedene Datenquellen wie die Bugzilla-Software oder Subversion angebunden werden kann.

Qualitätsmodelle, mit deren Hilfe beschrieben wird, welche Qualitätsmerkmale von einem Entwicklungsprozess erwartet werden, dienen als Hilfsmittel zum Erkennen der Verbesserungspotentiale in Entwicklungsprozessen. Ziel von QMetric ist es, eine Verknüpfung von Qualitätsmodellen mit den

durch Metriken bereitgestellten Daten herzustellen, so dass die Bewertung der Qualität auf Basis der Qualitätsmodelle automatisiert, beliebig oft wiederholbar sowie regelmäßig und damit zeit- und kostengünstig durchgeführt werden kann.

1.1 Aufgabenstellung

Die Aufgabe, mit der sich diese Diplomarbeit beschäftigt, ist die Erstellung einer neuen Anwendung als Teil von QMetric. Mit dieser Anwendung sollen Qualitätsmodelle für die Qualität von Softwareentwicklungsprozessen angelegt sowie bearbeitet werden können und es soll mit ihr möglich sein, die erreichte Qualität für existierende Softwareentwicklungsprojekte zu ermitteln.

Hierzu muss zunächst ein geeignetes Metamodell zur Repräsentation von Qualitätsmodellen sowie eine geeignete Darstellungsform für diese gefunden werden. Das Metamodell muss sodann um geeignete Strukturen ergänzt werden, die eine Verknüpfung mit den durch QMetric bereitgestellten Datenquellen und eine Bewertung der erreichten Qualität erlauben. Auf Basis dieser Arbeiten müssen die Anforderungen an das zu erstellende Werkzeug ermittelt werden. Daran anschließend muss die Architektur der Anwendung skizziert sowie eine Implementierung durchgeführt werden.

Die Diplomarbeit gliedert sich damit in die folgenden Arbeitsschritte:

1. **Einarbeitung in die Theorie** der Qualitätsmodelle, der Softwareprozessqualität sowie Möglichkeiten zu deren Bewertung und Erstellung eines Qualitätsmetamodells
2. **Analyse der Erweiterungen für Qualitätsmetamodelle** zwecks Integration in QMetric
3. **Ermittlung der Anforderungen** an eine Anwendung zum Anlegen und Bearbeiten von Qualitätsmodellen sowie zur Durchführung von Auswertungen auf deren Basis
4. **Erstellung einer Architekturbeschreibung** für diese Anwendung
5. **Implementierung der Anwendung** gemäß der Architektur
6. **Bewertung** der Ergebnisse

1.2 Überblick über die Ausarbeitung

In Kapitel zwei werden grundlegende Begriffe und Konzepte u.a. aus der Theorie der Qualitätsmodelle, auf die in den nachfolgenden Kapiteln Bezug genommen wird, definiert. Das daran anschließende Kapitel drei erläutert die funktionalen und nichtfunktionalen Anforderungen an die zu erstellende Anwendung. Kapitel vier beschreibt den Softwareentwicklungsprozess, der bei der Erstellung der Anwendung verwendet wurde. Die Architektur der Programmstruktur der Anwendung wird in Kapitel fünf dargelegt. Daran an schließt mit Kapitel sechs ein Abschnitt, in dem das Ergebnis der Entwicklung sowie der Entwicklungsprozess selbst bewertet werden. Das letzte Kapitel zieht ein Fazit der Entwicklungsarbeit und gibt einen Ausblick auf zukünftige Erweiterungs- und Verbesserungsmöglichkeiten der Anwendung.

1.3 Danksagungen

Ich danke Prof. Dr. rer. nat. Lichter für die Möglichkeit, meine Diplomarbeit am Lehr- und Forschungsgebiet Informatik 3 erstellen zu können. Außerdem danke ich ihm und Prof. Dr.-Ing. Schroeder für die Begutachtung meiner Arbeit. Großer Dank gebührt Holger Schackmann, der mir als Betreuer zur Seite gestanden hat. Des Weiteren gilt mein Dank Alexander Nyssen und Veit Hoffmann vom Lehr- und Forschungsgebiet Informatik 3 sowie meinem Kommilitonen Henning Schäfer. Sie haben mir in einer frühen Phase der Diplomarbeit wertvolles Feedback gegeben.

Schließlich danke ich meinen Eltern für die Unterstützung, die ich während meines gesamten Studiums durch sie erfahren habe. Ohne sie wäre diese Arbeit nicht möglich gewesen.

Kapitel 2

Grundlagen

In diesem Kapitel werden wichtige grundlegende Begriffe, auf die in den nachfolgenden Kapiteln Bezug genommen wird, definiert und erläutert.

2.1 Allgemeine Begriffe

Definition 1. Software-Projekt

Ein **Software-Projekt** wird in [TY97] definiert als

A temporary activity that is characterized by having a start date, specific objectives and constraints, established responsibilities, a budget and schedule, and a completion date.

Diese Definition wird in [LL07, S. 90] um die Zusätze erweitert, dass kaum ein Projekt über stabile Ziele und Randbedingungen verfügt und dass es mit dem Kunden einen Abnehmer für das Ergebnis des Projektes gibt.

Definition 2. Graph

In [Ste02, S. 52] wird ein **Graph** als Tupel (V, E) definiert, wobei V eine endliche nichtleere Menge von Knoten ist. Die Menge E ist eine Teilmenge der zweielementigen Teilmengen von V , also $E := \{\{x, y\} \mid x, y \in V, x \neq y\}$. Die Elemente der Menge E bezeichnet man als Kanten.

Bei **gerichteten Graphen** sind die Kanten zusätzlich noch orientiert. Das heißt, dass eine Kante also nicht durch eine zweielementige Menge, sondern

durch ein geordnetes Paar dargestellt wird: $E \subseteq V \times V$.

Graphen sind gut geeignet, um hierarchisch gegliederte Strukturen auf grafischem Weg zu beschreiben.

Definition 3. Modell

Bei einem **Modell** handelt es sich um eine Abstraktion eines Originals, bei der bewusst auf nicht relevante Merkmale verzichtet wird. In [Sta73, S. 128ff] wird der allgemeine Modell-Begriff durch drei Merkmale gekennzeichnet:

1. *Abbildungsmerkmal*: Bei Modellen handelt sich stets um Abbildungen oder Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
2. *Verkürzungsmerkmal*: Durch Modelle werden in der Regel nicht alle Attribute des zugrunde liegenden Originals erfasst, sondern nur die, die für den Kontext, in dem das Modell Verwendung findet, relevant erscheinen.
3. *Pragmatisches Merkmal*: Ein Modell ist einem Original nicht per se eindeutig zuzuordnen, sondern es wird von einem modellbenutzenden Subjekt innerhalb einer bestimmten Zeitspanne und unter Einschränkung auf einen bestimmten Zweck eingesetzt. Es ist also bei der Arbeit mit einem Modell nicht nur wesentlich, *wovon* das Modell ein Abbild ist, sondern auch *für wen*, *wann* und *wozu* es verwendet wird.

2.2 Qualität und Qualitätsmodelle

Dieser Abschnitt beschreibt den Begriff der Qualität, Qualitätsmodelle sowie die Motivation und das Konzept bidirektionaler Qualitätsmodelle.

Definition 4. Qualität

Der Begriff **Qualität** wird in der DIN 55350 wie folgt definiert:

Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen.

Von besonderem Interesse im Kontext dieser Arbeit ist die **Softwareprozessqualität**, also die Qualität des Prozesses, der zur Erstellung einer Software herangezogen wird. In [LL07, S. 64] werden für diese Art der Qualität die folgenden Merkmale herausgearbeitet:

- Einhaltung von Kosten- und Termingrenzen
- Minimierung des Aufwands
- Sammeln von Kenntnissen
- Schaffung wiederverwendbarer Komponenten
- Pflege des Betriebsklimas

Eine hohe Prozessqualität in einem Unternehmen garantiert nicht sofort, dass auch alle Produkte, die im Unternehmen entstehen, mit dieser hohen Qualität entwickelt und produziert werden. Vielmehr ist es erforderlich, dass die allgemein vorhandene Prozessqualität für das jeweilige Produkt in eine hohe Projektqualität übertragen wird.

Kenntnisse über die Prozessqualität eines Herstellers von Software kann bei dessen Kunden in die Entscheidungsfindung für oder gegen eine Auftragsvergabe einfließen.

Die Bildung von Qualitätsmodellen im Kontext von Software geht zurück auf die Arbeiten von Boehm [BBK⁺78] und McCall [CM78] aus den Siebzigerjahren des vergangenen Jahrhunderts.

2.2.1 Problematik bei der Erfassung von Qualitäten

In der Regel beschreiben Qualitätsmodelle für Softwareentwicklungsprozesse subjektive Erwartungen an den Prozess. Ein Beispiel dafür ist die Qualitätsanforderung, die ein gutes Projektklima fordert. Es handelt sich dabei insofern um eine subjektive Erwartung, als dass verschiedene Beteiligte am Entwicklungsprozess unterschiedliche Vorstellungen bezüglich eines für sie angenehmen Projektklimas haben und somit eine fundierte Aussage über das Erreichen dieser Qualitätsanforderungen nicht objektiv möglich ist.

Um eine objektive Bewertung von Qualitätsanforderungen sicherzustellen, werden in [SSM06, S. 35ff] **bidirektionale Qualitätsmodelle** beschrieben,

die eine Verknüpfung subjektiver Qualitätserwartungen und objektiver Bewertungen dieser Qualitäten ermöglichen.

Im nächsten Abschnitt erfolgt nun ein kurzer Überblick über diese Art von Qualitätsmodellen.

2.2.2 Bidirektionale Qualitätsmodelle

Auf der einen Seite der bidirektionalen Qualitätsmodelle setzt sich der Begriff der Qualität zusammen aus unterschiedlichen *Qualitätseigenschaften*, die möglicherweise wiederum aus feingranulareren Qualitätseigenschaften konstruiert werden. Diese Qualitätseigenschaften sind in der Regel nicht ohne weiteres objektiv messbar.

Auf der anderen Seiten der bidirektionalen Qualitätsmodelle bieten *Qualitätsmerkmale* die Möglichkeit, auf objektive Art und Weise zwischen Entitäten (zum Beispiel zwischen zwei Softwareentwicklungsprozessen) zu unterscheiden.

Die Bewertung und Ermittlung des Erfüllungsgrades von Qualitätsmerkmalen kann zum Beispiel mithilfe von Tools wie BugzillaMetrics ([Bugb]) erfolgen. So kann man zum Beispiel die Qualitätsanforderung “Stabilität/Kontinuität”, die als Unterpunkt der oben thematisierten Anforderung “Gutes Projektklima” anzusehen ist, auf die mit BugzillaMetrics objektiv messbaren Qualitätsmerkmale “Häufigkeit P1 Bugs” und “Aufwand P1 Bugs” abbilden.

Aus einem oder mehreren Qualitätsmerkmalen setzen sich *Qualitätsindikatoren* zusammen. Sie werden in [SSM06, S. 54] definiert als

ein Problemmuster, das als beweiskräftiger Anzeiger (Indiz) von einem oder mehreren Qualitätsmerkmalen auf eine (oder mehrere) Qualitätseigenschaften dient. Naturgemäß muss sichergestellt sein, dass der Indikator eine aussagekräftige, u.U. gewichtete Beweiskraft für die untersuchten Qualitätseigenschaften besitzt.

Somit bilden Qualitätsindikatoren das Bindeglied zwischen den subjektiven Qualitätseigenschaften und den objektiv feststellbaren Qualitätsmerkmalen. Eine Qualitätseigenschaft ist somit zwar nicht direkt messbar, kann jedoch durch mathematische Zusammenhänge wie lineare Gleichungen über einer Menge anderer untergeordneter Qualitätseigenschaften und Qualitätsindikatoren beschrieben werden.

2.3 Messungen und Metriken

Dieser Abschnitt beschreibt grundlegende Begriffe, die sich mit Messungen und Metriken, durch die Messungen beschrieben werden, befassen.

Definition 5. Messung

In [FP97] wird eine **Messung** definiert als

the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.

Das Messen liefert also eine Aussage über eine (vorher unbekannte) Größe eines Objektes aus der realen Welt. Der Zweck einer Messung ist in der Regel, die Grundlage für zu treffende Entscheidungen herzuleiten.

Definition 6. (Software-)Metrik

[IEE98] definiert den Begriff der **Softwaremetrik** wie folgt:

Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.

In [Tha94] wird der Begriff der Softwaremetrik aufgeteilt in **Produktmetriken**, das heißt Metriken, die eine Aussage über Eigenschaften wie Code-Qualität oder Preisstruktur einer Software-Einheit liefern, und **Prozessmetriken**. Diese liefern eine Aussage über den Erfüllungsgrad der an einen Softwareentwicklungsprozess gestellten Qualitätsanforderungen.

Definition 7. Quantil

Ein **Quantil** wird in [Geo04, S. 225] wie folgt definiert:

Sei Q ein Wahrscheinlichkeitsmaß auf $(\mathbb{R}, \mathcal{B})$ und $0 < \alpha < 1$. Dann heißt jede Zahl $q \in \mathbb{R}$ mit $Q([q, \infty]) \geq 1 - \alpha$ ein α -Quantil von Q . Ein $\frac{1}{2}$ -Quantil von Q ist gerade der Median, und ein $(1 - \alpha)$ -Quantil heißt auch ein α -Fraktile von Q . Die $\frac{1}{4}$ - und $\frac{3}{4}$ -Quantile heißen auch das untere und obere Quartil.

Anschaulich bedeutet die Aussage, dass ein (Mess-)Wert x im ersten Quartil bzw. unter dem $\frac{1}{4}$ -Quantil angesiedelt ist, dass x unterhalb von 75% aller in der betrachteten Menge enthaltenen Kennzahlen liegt. Quantile erlauben somit eine intuitive und leicht verständliche Interpretation der Lage eines Wertes im Verhältnis zu einer gegebenen empirischen Datenbasis. Außerdem ist beispielsweise der Median weniger anfällig für Ausreißer in der Datenbasis als die reine Betrachtung eines Mittelwertes.

2.4 BugzillaMetrics

Bei **BugzillaMetrics** [Bugb] handelt es sich um ein Werkzeug zur Auswertung von Änderungsanträgen mithilfe von Metriken. Die Metriken werden dabei in einer deklarativen, XML-basierten Spezifikation durch den Benutzer des Werkzeugs beschrieben. Die Metrik-Spezifikationen sind unabhängig vom den Änderungsanträgen zugrunde liegenden Datenbankschema.

2.4.1 Änderungsanträge

Von zentraler Bedeutung für die Arbeit mit BugzillaMetrics ist die Existenz und die Verfügbarkeit von Änderungsanträgen. Daher erfordert dieser Begriff eine genauere Definition:

Definition 8. Änderungsantrag

Ein **Änderungsantrag** (engl. *Change Request, CR*) bezeichnet im Projektmanagement einen formal erfassten Wunsch nach Veränderungen eines Merkmals eines mit dem Projekt assoziierten Produktes oder Gegenstandes. Die Motivation für die Erstellung eines Änderungsantrages ist entweder das Vorhandensein eines Defektes oder der Wunsch nach Erweiterung oder Verbesserung vorhandener Funktionalität.

Änderungsanträge werden in Change Request Management (CRM) Software erfasst und verwaltet. Beispiele für bekannte CRM-Anwendungen sind Bugzilla [Buga] oder Mantis [Man]. In CRM-Anwendungen wird in der Regel eine Reihe von Informationen mit einem Änderungsantrag verknüpft. Dazu gehört neben dem Produkt, auf das sich der Antrag bezieht, auch der Bearbeiter des Antrages, die Zeitpunkte von Erfassung und Fertigstellung, die

Priorität, mit der der Antrag bearbeitet wird, sowie ein Status, der wiedergibt, ob der Antrag offen, in Bearbeitung, abgeschlossen oder beispielsweise wiedereröffnet ist.

Weitere gebräuchliche Bezeichnungen für Änderungsanträge sind *Bug*, *Case* und *Issue*.

2.4.2 Motivation für BugzillaMetrics

Nachfolgend wird auf Version 0.9.4 von BugzillaMetrics Bezug genommen. In dieser Version unterstützt das Werkzeug nur die CRM-Anwendung Bugzilla. Eine Erweiterung um eine Anbindung an CVS und Subversion ist für eine spätere Version ebenso geplant wie die Unterstützung für Mantis.

Die Motivation für das Anwenden von Metriken auf Änderungsanträgen liegt darin begründet, dass die Ergebnisse der Auswertung helfen, die Evolution von Softwareentwicklungsprozessen zu bewerten und Schwachstellen und Verbesserungspotential im gewählten Entwicklungsprozess aufzudecken.

Durch die Möglichkeit, in BugzillaMetrics individuelle Metriken in XML zu spezifizieren, ist das Werkzeug wesentlich flexibler als die Schnittstellen zur Auswertung in bekannten CRM-Anwendungen [Oba07, S. 26ff].

2.4.3 Metrik-Spezifikation

Grundsätzlich besteht eine Metrik für BugzillaMetrics aus *Filtern*, mit denen gezielt nach Eigenschaften der Änderungsanträge selektiert werden kann, und *Ereignissen*, die das Auftreten von historischen Änderungen im Änderungsantrag abbilden. Zu diesen Ereignissen gehören beispielsweise das Anlegen des Eintrags oder das Zuweisen zu einem Entwickler.

Da auf die Struktur der Metrik-Spezifikation in späteren Kapiteln Bezug genommen wird, ist eine Beschreibung hier erforderlich. In Listing 2.1 ist eine Metrik-Spezifikation wiedergegeben, mit der die Anzahl der noch nicht abgeschlossen oder wiedereröffneten Änderungsanträge erfasst wird.

Listing 2.1: Metrik-Spezifikation

```
<metric>
  <baseFilter>
    <value field="product">2</value>
```

```

</baseFilter>
<groupingParameters>
  <none />
</groupingParameters>
<groupEvaluations>
  <calculation name="totals">
    <sum caseValueCalculator="totals" />
  </calculation>
</groupEvaluations>
<caseValueCalculators>
  <countEvents id="totals">
    <event>
      <and>
        <endOfTimeInterval />
        <stateFilter>
          <or>
            <value field="status">NEW</value>
            <value field="status">REOPENED</value>
            <value field="status">ASSIGNED</value>
          </or>
        </stateFilter>
      </and>
    </event>
    <weight>
      <default />
    </weight>
  </countEvents>
</caseValueCalculators>
<evaluationTimePeriod>
  <timePeriod>
    <start>2007-10-14</start>
    <end>2008-10-14</end>
  </timePeriod>
</evaluationTimePeriod>
<timePeriodGranularity>
  <week />
</timePeriodGranularity>
<fixedFields>
  <field>product</field>
</fixedFields>
</metric>

```

Die Spezifikation einer Metrik besteht aus mehreren Elementen, die hier beschrieben werden:

- Der **Base Filter** einer Metrik beschreibt, welche Änderungsanträge bei der Auswertung Berücksichtigung finden. Dadurch ist es möglich,

die Metrik auf einzelne Produkte zu beschränken, wenn die CRM-Anwendung für mehrere Produkte gleichzeitig verwendet wird.

- Mithilfe der **Evaluation Time Period** kann festgelegt werden, aus welchem Zeitraum Änderungsanträge berücksichtigt werden sollen.
- Der Parameter **Time Granularity** bestimmt, nach welchem Intervall (Tag, Kalenderwoche, Monat, Jahr) der durch Evaluation Time Period festgelegte Zeitraum in Zeitfenster aufgeteilt werden soll.
- Der **Grouping Parameter** gibt an, nach welchem Kriterium Änderungsanträge in der Auswertung gruppiert werden. So ist es beispielsweise möglich, alle Änderungsanträge, die sich auf das gleiche Produkt beziehen, zusammengefasst zu betrachten und zu bewerten.
- Auf die gemessenen Rohdaten wird ein vordefinierter **Value Calculator** angewendet, der den einzelnen Änderungsanträgen einen Wert zuweist. Ein Beispiel hierfür ist die Dauer, die ein Eintrag in einem bestimmten Zustand verbracht hat.

BugzillaMetrics 0.9.4 unterstützt die folgenden Value Calculators:

- Der **Count Events** Calculator berechnet für jeden Änderungsantrag einen Wert, indem das Auftreten eines spezifiziertes Ereignisses, das beim Änderungsantrag auftritt, mit einer Gewichtung multipliziert wird.
 - Der **Count Events Until** Calculator ermittelt für jeden Änderungsantrag, wie oft ein Ereignis stattfindet, bis ein anderes Ereignis eintritt.
 - Der **Interval Length** Calculator ermittelt pro Änderungsantrag die Anzahl der Tage, die zwischen dem Auftreten zweier Ereignisse liegen.
 - Der **State Residence Time** Calculator berechnet für jeden Änderungsantrag die Anzahl der Tage, die der Antrag in einem bestimmten Zustand war, bis ein bestimmtes Ereignis eingetreten ist.
- Mithilfe von **Group Evaluations** wird festgelegt, wie die Werte, die von den Value Calculators berechnet worden sind, in der Metrik kombiniert werden.

Kapitel 3

Anforderungen

Dieses Kapitel beschreibt die wichtigen Bestandteile der Anforderungsspezifikation für das zu erstellende Werkzeug. Sie basiert auf den Konzepten und Begrifflichkeiten, die in den vorangegangenen Kapiteln vorgestellt wurden.

3.1 Einleitung

3.1.1 Ziel des Werkzeugs

Das Ziel des Werkzeugs soll die Bereitstellung einer Anwendung sein, mit deren Hilfe Qualitätsmodelle für Softwareentwicklungsprozesse, basierend auf der Idee der bidirektionalen Qualitätsmodelle, erstellt, verwaltet und auf gegebene Software-Projekte angewandt werden können. Die Anwendung muss über grafische und intuitive Benutzerschnittstellen verfügen, die eine geringe Einstiegshürde darstellen.

Die objektive Beantwortung der Frage nach der Qualitätserfüllung erfolgt durch Auswertung der Daten von Änderungsanträgen.

3.1.2 Benutzergruppen

Bei den möglichen Benutzern des Werkzeugs kann zwischen drei Gruppen unterschieden werden: **Qualitätsmanager**, **Projektmanager** und **Manager von Produktportfolios**.

In diesem Abschnitt werden diese Benutzergruppen und die Tätigkeiten, die sie im Werkzeug durchführen, beschrieben.

3.1.2.1 Qualitätsmanager

Die Qualitätsmanager nutzen das Werkzeug dazu, Instanzen des Qualitätsmodells zu definieren, zu erweitern und zu überarbeiten. Sie benötigen daher Lese- und Schreibzugriff auf die Gestaltung der Graphen, mit denen die bidirektionalen Qualitätsmodelle beschrieben werden, und auf deren Eigenschaften. Qualitätsmanager legen nicht nur die Anzahl der Qualitätseigenschaften und Qualitätsindikatoren fest, sondern bestimmen auch, wie die Werte von Qualitätseigenschaften berechnet bzw. wie jene von Qualitätsindikatoren gemessen werden.

3.1.2.2 Projektmanager

Projektmanager sind daran interessiert, für das ihnen unterstellte Projekt die aktuell erreichte Qualität zu messen, um Schwachstellen im Softwareentwicklungsprozess auszumachen oder um zu überprüfen, ob qualitätsverbessernde Maßnahmen zum Erfolg führen.

Daher nutzen Projektmanager die von Qualitätsmanagern definierten Modellinstanzen, indem sie auf deren Basis konkrete Auswertungen konfigurieren und durchführen. Nach erfolgter Auswertung können die Projektmanager auf die gemessenen Werte von Qualitätsindikatoren bzw. die berechneten Werte von Qualitätseigenschaften zugreifen.

Projektmanager nutzen nur die durch die Qualitätsmanager erstellten Instanzen, können diese jedoch nicht selbst verändern oder neue Qualitätsmodelle anlegen.

3.1.2.3 Produktportfolio-Manager

Produktportfolio-Manager tragen Projektverantwortung für eine Menge in der Regel verwandter Projekte, die jeweils von einem Projektmanager betreut werden. Für Produktportfolio-Manager ist es wichtig zu erkennen, bei welchen der von ihnen verantworteten Projekte im Bezug auf die erreichte Prozessqualität Verbesserungsbedarf im Vergleich zu den übrigen Projekten besteht, um sodann geeignete Verbesserungs- und Lenkungsmaßnahmen für diese Projekte auf den Weg zu bringen.

Aus diesem Grund ist es Produktportfolio-Managern möglich, für ihre Projekte eine gemeinsame Auswertung durchzuführen und im Anschluss die

Ergebnisse, aufgeschlüsselt nach Projekt, einzeln abzufragen und zu vergleichen.

Wie die Projektmanager nutzen die Produktportfolio-Manager nur die durch die Qualitätsmanager erstellen Instanzen, können diese jedoch nicht selbst verändern oder neue Modelle erstellen.

3.1.3 Zusammensetzung

Aus den beschriebenen Benutzergruppen ergibt sich für die zu erstellende Anwendung, dass sie im Kern in zwei nach außen sichtbare Hauptkomponenten untergliedert werden muss: Die Komponente des **Qualitätsmodell-Editors** und die **Auswertungswerkzeug**-Komponente.

Qualitätsmodell-Editor:

Der Qualitätsmodell-Editor wird durch die Benutzergruppe der Qualitätsmanager verwendet, um neue Qualitätsmodelle auf Basis des Konzeptes der bidirektionalen Qualitätsmodelle zu erstellen und bestehende Qualitätsmodelle zu bearbeiten.

Auswertungswerkzeug:

Mit dem Auswertungswerkzeug führen die Benutzergruppen der Produktmanager und der Produktportfolio-Manager Qualitätsbewertungen für konkrete Projekte durch, indem sie auf Basis eines der, durch die Qualitätsmanager zur Verfügung gestellten, Qualitätsmodelle Umfang und Zeitraum von Messungen mithilfe von Metrikwerkzeugen konfigurieren und diese Messungen durchführen.

Integration:

Beide Komponenten müssen sich gut in den Arbeitslauf in Softwareunternehmen integrieren. Daher müssen sie beide als Plug-ins für die frei und für eine Vielzahl an Betriebssystemen verfügbare Entwicklungsumgebung Eclipse ([Ecl]) zur Verfügung gestellt werden.

Eclipse ist nicht nur ein weit verbreitetes Werkzeug zur Softwareentwicklung, sondern bietet mit dem *Plug-in Development Environment* ([PDE]) auch eine Infrastruktur zur Entwicklung von Erweiterungen in Form so genannter Plug-ins an. Außerdem verfügt die Eclipse-Plattform mit *GMF* ([GMF])

über ein Rahmenwerk, mit dem sich grafische Editoren gemäß der in Abschnitt 3.2.2 spezifizierten Anforderungen entwickeln lassen.

3.2 Funktionale Anforderungen

Der folgende Abschnitt beschreibt die funktionalen Anforderungen an die Werkzeuge. Gemeinhin wird mit dieser Art der Anforderungen festgelegt, *was* die Werkzeuge leisten sollen.

3.2.1 Qualitätsmodelle

Der Qualitätsmodell-Editor muss in der Lage sein, Qualitätsmodelle zu erfassen und für die spätere Verwendung zu speichern. Dieser Abschnitt beschreibt die Anforderungen an die Qualitätsmodelle und die ihnen zugrunde liegende Datenstruktur.

Die vorhergehende Beschreibung der bidirektionalen Qualitätsmodelle legt nahe, dass deren Repräsentation auf Basis eines antizyklischen Graphen erfolgt. Es sind die folgenden Knotenarten zu unterscheiden:

- Qualitätsindikator
- Qualitätseigenschaft

Das dritte im vorherigen Abschnitt erwähnte Konzept der Qualitätsmerkmale ist konzeptionell im Metrikwerkzeug angesiedelt. Es wird daher in den folgenden Ausführungen nicht weiter betrachtet.

Im Folgenden werden die unterschiedlichen Arten von Knoten genauer beschrieben. Die Kanten des Graphen sind gerichtet. Eine Kante von Knoten A zu Knoten B bedeutet, dass die Bewertung von A in die Berechnung von B einfließt.

3.2.1.1 Qualitätseigenschafts-Knoten

Diese Knoten haben eingehende Kanten von anderen Qualitätseigenschafts-Knoten, wenn sie durch feingranularere Qualitätseigenschaften weiter untergliedert werden können. Andernfalls verfügen sie über eine oder mehrere eingehende Kanten von Qualitätsindikator-Knoten.

Berechnungsvorschriften

Da sich Qualitätseigenschaften aus Qualitätsindikatoren und anderen Qualitätseigenschaften zusammensetzen, muss es möglich sein, im Qualitätsmodell festzulegen, wie sich eine Eigenschaft aus den mit ihr assoziierten Eigenschaften und Indikatoren berechnet. Diese Aufgabe übernimmt pro Qualitätseigenschaft eine mit ihr assoziierte Berechnungsvorschrift, die in einer geeigneten Datenstruktur pro Qualitätseigenschaft gespeichert wird.

Grundsätzlich muss zwischen zwei Arten von Berechnungsvorschriften unterschieden werden: **Komplexe Funktionen** und **Filterfunktionen**. Im Folgenden werden beide Arten beschrieben.

Komplexe Funktionen

Bei komplexen Funktionen handelt es sich um mehrstellige Abbildungen der Form

$$q : \mathbb{R} \times \cdots \times \mathbb{R} \rightarrow \mathbb{R}; x_1, \dots, x_n \mapsto y \quad (3.1)$$

wobei $x_1, \dots, x_n, y \in \mathbb{R}$ gilt und die x_i die berechneten bzw. gemessenen Bewertungen der assoziierten Qualitätseigenschaften bzw. Qualitätsindikatoren darstellen. Es ist darüber hinaus möglich, die einzelnen x_i vor dem Einsetzen in q mit einstelligen Filterfunktionen, die im anschließenden Abschnitt beschrieben werden, zu bearbeiten. In diesem Fall haben die Abbildungen die Form

$$q(f_1(x_1), \dots, f_n(x_n)) = y \quad (3.2)$$

für n Filterfunktionen f_1, \dots, f_n .

Sowohl Qualitätseigenschaften mit nur einer assoziierten Qualitätseigenschaft bzw. nur einem assoziierten Qualitätsindikator als auch auf Qualitätseigenschaften mit mehreren assoziierten Eigenschaften bzw. Indikatoren können mit komplexen Funktionen verknüpft werden.

In der ersten Ausbaustufe des Werkzeugs müssen zwei komplexe Funktionen angeboten werden: **Lineare Gleichungen** und **Quantil-Berechnungen**.

1. Lineare Gleichungen

Eine Verarbeitung der berechneten bzw. gemessenen Bewertungen assoziierter Qualitätseigenschaften bzw. assoziierter Qualitätsindikatoren mithilfe von linearen Gleichungen bietet sich, basierend auf [CM78, S. 137], dann an, wenn die assoziierten Qualitätseigenschaften und Qualitätsindikatoren mit unterschiedlicher Stärke auf die Qualitätseigenschaft, mit der sie assoziiert sind, einwirken.

In diesem Fall muss folgende Gleichung verwendet werden:

$$q(x_1, \dots, x_n) = c_1x_1 + c_2x_2 + c_3x_3 + \dots + c_nx_n \quad (3.3)$$

wobei q die Bewertung der Gesamtqualität und c_i die Koeffizienten für die Gewichtungen der Einzelbewertungen e_i der mit der Qualitätseigenschaft assoziierten Qualitätseigenschaften bzw. Qualitätsindikatoren sind.

2. Quantil-Berechnungen

Die Betrachtung von Quantil-Einordnungen erlaubt in vielen Fällen eine einfache und anschauliche Interpretation von gemessenen oder berechneten Qualitätswerten.

Die komplexe Funktion zur Quantil-Berechnung ermöglicht es, für eine Qualitätseigenschaft, die nicht per se (siehe unten für eine Erklärung dieses Sachverhalts) über eine empirische Datenbasis (siehe Erläuterung hierzu auf Seite 24) und somit über die Möglichkeit zur direkten Bestimmung einer Quantil-Einordnung, wie es bei entsprechenden Qualitätsindikatoren möglich wäre, verfügt, trotzdem eine solche Einordnung sinnvoll zu berechnen.

Mit jeder Quantil-Berechnungsfunktion verknüpft ist eine weitere komplexe Funktion, wie zum Beispiel eine lineare Gleichung. Die eigentliche Berechnung umfasst drei Schritte:

- (a) Im ersten Schritt wird eine lokale empirische Datenbasis für die Qualitätseigenschaft aus den empirischen Datenbasen der assoziierten Qualitätseigenschaften bzw. Qualitätsindikatoren berechnet.

Sei dazu c die mit der Quantil-Berechnung verknüpfte komplexe Funktion. Die Menge $(ed_{i,0}, \dots, ed_{i,m})$ enthalte die Werte der empirischen Datenbasis für die i -te assoziierte Qualitätseigenschaft

bzw. den i -ten assoziierten Qualitätsindikator. Insgesamt gebe es n assoziierte Eigenschaften und Indikatoren. Die Anzahl der Elemente m muss für alle Datenbasen gleich sein.

Es wird dann die lokale empirische Datenbasis ed auf folgendem Wege berechnet:

$$ed = (c(ed_{1,0}, \dots, ed_{n,0}), \dots, c(ed_{1,m}, \dots, ed_{n,m})) \quad (3.4)$$

- (b) Im zweiten Schritt wird der absolute Wert der Qualitätseigenschaft mithilfe der verknüpften komplexen Funktion ermittelt. Seien dazu qp_1, \dots, qp_n die berechneten bzw. gemessenen Werte der assoziierten Qualitätseigenschaften bzw. Qualitätsindikatoren. Es ergibt sich dann als absoluter Wert qp :

$$qp = c(qp_1, \dots, qp_n) \quad (3.5)$$

- (c) Abschließend wird für qp die Quantil-Position in der Menge ed bestimmt. Diese Position wird als Wert der entsprechenden Qualitätseigenschaft verwendet.

Weitere komplexe Funktionen sind denkbar. So sollte es in künftigen Ausbaustufen der Anwendung beispielsweise möglich sein, das Minimum, das Maximum oder den (gewichteten) Mittelwert der Werte der assoziierten Eigenschaften oder Indikatoren als Bewertung für eine Qualitätseigenschaft zu verwenden.

Filterfunktionen

Bei Filterfunktionen handelt es sich um einstellige Funktionen, die einen beliebigen Wert auf einen anderen Wert abbilden:

$$f : \mathbb{R} \rightarrow \mathbb{R}; x \mapsto y \quad (3.6)$$

mit $x, y \in \mathbb{R}$.

Filterfunktionen können innerhalb von komplexen Funktionen zur Vorverarbeitung einzelner Bestandteile der Funktionen verwendet werden.

Im Werkzeug werden die folgenden Filterfunktionen unterstützt:

1. **Identische Abbildung:** Eine Qualitätseigenschaft, die diese Filterfunktion verwendet, gibt die für die assoziierte Qualitätseigenschaft berechnete absolute Bewertung bzw. die für den assoziierte Qualitätsindikator gemessene Bewertung ohne weitere Umformungen zurück.
2. **Quantil-Filter:** Wird diese Filterfunktion von einer Qualitätseigenschaft verwendet, gibt die Eigenschaft die Einordnung des Wertes der assoziierten Eigenschaft bzw. des assoziierten Indikators in das entsprechende Quantil zurück. Die Quantil-Thematik wird in Abschnitt 3.2.1.2 weiter erörtert.
3. **Skalierungs-Filter:** Diese Filterfunktion nimmt eine Skalierung der ursprünglich berechneten bzw. gemessenen Bewertung auf einen bestimmten Bereich, zum Beispiel zwischen 0 und 1, vor. Man bedient sich dabei der Formel

$$x_{norm} = \left[(x - min) \cdot \frac{max_{norm} - min_{norm}}{max - min} \right] + min_{norm}, \quad (3.7)$$

wobei x die berechnete bzw. gemessene Bewertung der assoziierten Qualitätseigenschaft bzw. das assoziierten Qualitätsindikators ist und $(min, max) \in (\mathbb{R}, \mathbb{R})$ den Wertebereich, in dem x liegt, sowie $(min_{norm}, max_{norm}) \in (\mathbb{R}, \mathbb{R})$ den Wertebereich, in den x skaliert werden soll, beschreiben. Die Variable x_{norm} ist die skalierte/normierte Bewertung.

Wird die Bewertung einer Qualitätseigenschaft skaliert, so werden min und max durch $q(min_1, \dots, min_n)$ bzw. $q(max_1, \dots, max_n)$ ermittelt, wobei min_i bzw. max_i die untere bzw. obere Grenze des Wertebereichs des i -ten mit der Qualitätseigenschaft verbundenen Indikators sind.

Vorraussetzung für die Verwendung eines Skalierungs-Filters ist, dass die assoziierte Qualitätseigenschaft bzw. der assoziierte Qualitätsindikator über einen endlichen Wertebereich verfügt.

4. **Abbildung auf benutzerdefinierte Wertebereiche:** Hierbei definiert der Benutzer eine Abbildung von Wertebereichen auf skalare Werte. Für die berechnete bzw. gemessene Bewertung der assoziierten Qualitätseigenschaft bzw. des assoziierten Qualitätsindikators wird dann der passende Wertebereich ermittelt und der diesem Wertebereich zugeordnete skalare Wert wird als Bewertung verwendet.

Vorraussetzung für sinnvolle Ergebnisse dieser Filterfunktion ist, dass die benutzerdefinierten Wertebereiche den gesamten Zahlenraum von $-\infty$ bis $+\infty$ abdecken.

5. **Schwellenwert-Filter:** Dieser Filter weist allen Werten, die unterhalb eines definierten Schwellenwertes t liegen, den Wert a zu, während alle Werte, die größer als t sind, auf den Wert b abgebildet werden:

$$x_{filtered} = \begin{cases} a : x \leq t \\ a : x > t \end{cases} \quad (3.8)$$

Weitere Eigenschaften

Um die Bedeutung von Knoten zu erläutern, ist es möglich, eine Beschreibung als Freitext mit ihnen zu assoziieren. Jeder Knoten kann optional mit einer oberen und einer unteren Schranke versehen werden, die gemeinsam den Wertebereich des Qualitätswertes, der bei einer Messung auf Basis des betreffenden Qualitätsmodell mit dem Knoten assoziiert ist, einschränken. Die obere wie auch die untere Schranke können dabei auch den Wert ∞ bzw. $-\infty$ annehmen, was bedeutet, dass der Wertebereich in den entsprechenden Richtung unbeschränkt ist.

Auch bei dem Wurzelknoten des Graphen handelt es sich um Qualitätseigenschafts-Knoten. Ein weitere Spezialisierung ist hier nicht erforderlich. Außerdem ist denkbar, dass ein graphische Darstellung mehr als einen Wurzelknoten enthält.

3.2.1.2 Qualitätsindikator-Knoten

Wie Qualitätseigenschafts-Knoten verfügen sie ebenfalls über alle in Abschnitt 3.2.1.1 spezifizierten Eigenschaften. Qualitätsindikator-Knoten können allerdings nicht als Wurzelknoten vorkommen, sondern haben im finalen Modell stets ausgehende Kanten zu Qualitätseigenschafts-Knoten. Zur Konstruktionszeit ist es allerdings denkbar, dass isolierte Qualitätsindikator-Knoten im Graphen vorhanden sind, die dann zu einem späteren Zeitpunkt mit einem Qualitätseigenschafts-Knoten verbunden werden.

In Qualitätsindikator-Knoten wird definiert, welches Metrikwerkzeug als Auswertungskomponente im Rahmen einer Messung unter Verwendung des

aktuellen Qualitätsmodells zum Einsatz kommt, um die den mit diesem Indikator verknüpften Wert zu bestimmen. Wie im Abschnitt 2.2.2 dargelegt, kann es sich dabei um vielfältige Werkzeuge handeln. Aus diesem Grund müssen Qualitätsindikator-Knoten auf flexible Art und Weise mit verschiedenen Auswertungskomponenten verknüpft werden können.

Ein Beispiel für eine Auswertungskomponente ist ein Interface zu BugzillaMetrics. Hierbei wird im Knoten eine Konfiguration für dieses Werkzeug hinterlegt, bei der bereits der *CaseValueCalculator* und die *GroupEvaluation* als XML-Definition vorliegen, weitere Parameter wie der *Basefilter*, der zu verwendende Zeitraum, die Granularität und die Zugriffsdaten zur zu verwendenden BugzillaMetrics-Installation noch fehlen. Diese Informationen sind spezifisch pro durchzuführender Messung und werden somit erst im Rahmen dieser festgelegt. Die Datenstruktur für die Aufnahme der Konfiguration muss allerdings generisch und flexibel genug sein, um beliebige andere Auswertungskomponenten bedienen zu können.

Optional verfügt ein Qualitätsindikator-Knoten über eine empirische Datenbasis, in der historische Werte für die betrachtete Metrik erfasst werden:

Die empirische Datenbasis

Um bei einer Messung für einen Qualitätsindikator über Vergleichswerte zur Einordnung der gemessenen Werte zu verfügen, ist es möglich, im Rahmen der eigentlich Messung zusätzlich eine empirische Datenbasis zu ermitteln. In ihr werden historische Werte für die in einem Qualitätsindikator betrachtete Metrik erfasst. Die Art und der Umfang der Information, die für die Elemente der Datenbasis vorliegen, hängt vom Metrikwerkzeug ab, das zusammen mit der Datenbasis bei einem Qualitätsindikator-Knoten Verwendung findet.

Jedes Element der Datenbasis ist entweder mit einem Zeitraum oder einem Zeitpunkt verknüpft, der eine zeitliche Einordnung der entsprechenden Werte erlaubt. Im Fall von BugzillaMetrics, das zeitraumorientiert arbeitet, repräsentiert ein Element der Datenbasis daher die Bewertung von einem gewissen Datum bis zu einem anderen Datum, während zum Beispiel das in [SSM06] beschriebene Code-Quality-Management Snapshot-orientiert ist und somit mit Bewertungen zu einem gewissen Zeitpunkt arbeitet.

Im Rahmen der Konfiguration der empirischen Datenbasis vor einer Auswertung muss es möglich sein, Filter auf der Datenbasis zu definieren, um sich bei der Auswertung der entsprechenden Metrik auf eine Teilmenge der verfügbaren historischen Daten zu beschränken. Gefiltert werden kann die Datenbasis sowohl nach Zeiträumen als auch nach Eigenschaften der einzelnen enthaltenen Elemente wie dem Produkt oder der Version, der das Element zugeordnet ist.

Auf Basis der Werte aus dem empirischen Datenbasis muss es möglich sein, den für den Qualitätsindikator gemessenen Wert in eine n -Quantil-Darstellung einzuordnen, was eine intuitive und leicht verständliche Interpretation für die Einordnung des Wertes ermöglicht. Liegt der gemessene Werte beispielsweise im ersten Quartil ($\frac{1}{4}$ -Quantil), so ist er besser (vorausgesetzt, kleine Werte sind bei der betrachteten Metrik besser als große) als mindestens 75% der historischen Werte aus der Datenbasis. Um die verschiedenen Qualitätsindikatoren eines Qualitätsmodells vergleichbar zu machen, muss n im gesamten Modell gleich sein.

Bei Verwendung von BugzillaMetrics ist nicht nur der für die Messung verwendete Zeitraum von Bedeutung, sondern auch die zeitliche Auflösung, nach der die Ergebnisse gruppiert werden. Möglich sind beispielsweise Tage, Wochen, Monate oder Jahre. Beide Werte sind von Bedeutung bei der Bestimmung der Vergleichswerte aus der empirischen Datenbasis; sie müssen daher in entsprechenden Datenstrukturen bei den Qualitätsindikatoren vorgehalten werden.

3.2.2 Qualitätsmodell-Editor

Der Qualitätsmodell-Editor muss bereitgestellt werden als Plug-in für die Eclipse-Anwendung. Mit ihm muss es möglich sein, bidirektionale Qualitätsmodelle für Softwareentwicklungsprozesse (siehe Abschnitt 3.2.1) neu anzulegen und nachträglich zu bearbeiten.

Dazu muss der Editor dem Benutzer eine **Zeichenoberfläche** zur Verfügung stellen, auf der die grafische Repräsentationen von Qualitätsmodellen in Form antizyklischer Graphen bearbeitet werden kann. Es müssen dazu Knoten-Symbole für Qualitätsindikatoren und -eigenschaften sowie ein Werkzeug zum Anlegen von Kanten zwischen zwei Knoten zur Verfügung stehen. Die Eigenschaften von Qualitätsknoten wie die zu verwendende Mess- oder Be-

rechnungsvorschrift, die anzuwenden Filterfunktionen und die Wertebereiche müssen nach Selektion eines Knotens über entsprechende Dialoge konfiguriert werden können. Auf der Zeichenfläche müssen die Knoten und Kanten auf beliebige Art und Weise anzuordnen sein. Darüber hinaus muss das Werkzeug eine Funktion bereitstellen, die es erlaubt, die Knoten nach einem festen Schema anzuordnen.

Der Qualitätsmodell-Editor muss insofern **Netzwerkfähigkeit** aufweisen, als dass bei der Konfiguration der Messvorschriften für Qualitätsindikatoren, die BugzillaMetrics verwenden, auf die Liste der und öffentlich freigegebenen vordefinierten Metriken einer über Netzwerk erreichbaren BugzillaMetrics-Installation zugegriffen werden kann. Gleichzeitig muss es trotzdem möglich sein, Metriken manuell durch Angabe der entsprechenden XML-Konfigurationsdaten zu definieren.

Die Forderung nach Netzwerkfähigkeit macht gegebenenfalls eine Erweiterung von BugzillaMetrics um eine Schnittstelle notwendig, über die externe Anwendungen wie das vorliegende Werkzeug Daten abfragen können. Diese Schnittstelle sollte ausreichend generisch implementiert werden, dass sie zukünftig auch für weitere Anwendungen nutzbar ist.

Zusätzlich muss es möglich sein, die grafische Darstellung von Qualitätsmodellen auszudrucken.

3.2.3 Auswertungswerkzeug

Das Auswertungswerkzeug, ebenfalls in Form eines Plug-ins für Eclipse, dient dazu, über eine grafische Benutzerschnittstelle Qualitätsbewertungen für Projekte auf Basis existierender Qualitätsmodelle durchzuführen. Diese Qualitätsmodelle wurden zuvor mithilfe des Qualitätsmodell-Editors angelegt und liegen in Form von Dateien auf dem System vor, auf dem die Auswertung durchgeführt werden soll.

Dazu wählt der Benutzer beim Anlegen einer neuen Auswertung im Rahmen eines Wizard-Dialogs das Qualitätsmodell aus und legt fest, auf welchem Produkt bzw. welchen Produkten und Versionen die Messung durchgeführt werden soll, und in welchem Zeitraum dies geschieht. Außerdem muss festgelegt werden können, nach welchen Kriterien die empirischen Vergleichsdaten für die Auswertung erfasst werden.

Nachdem die Messungen und Berechnungen für alle Knoten des Qualitätsmodells erfolgt sind, kann der Nutzer die Datenreihen auf verschiedene Art und Weise weiterverarbeiten:

- Auflistung der Werte in einer tabellarischen Darstellung. Hierbei müssen die einzelnen gemessenen bzw. berechneten Werte für einen Qualitätsindikator bzw. eine Qualitätseigenschaft sortiert nach dem Zeitraum, auf den sie sich beziehen, in Tabellenform dargestellt werden.
- Darstellung der Werte in einem XY-Diagramm. Bei dieser Art der Darstellung muss eine Datenreihe für einen Qualitätsknoten in grafischer Form dargestellt werden, indem die Werte in einem XY-Diagramm aufgetragen werden. Die X-Achse ist eine Zeitachse, auf welcher der jeweilige Bezugszeitraum für einen Wert aufgetragen ist. Die Y-Achse enthält die entsprechenden gemessenen bzw. berechneten Werte.

Diese Art der Darstellung lässt eine einfachere Interpretation der zeitlichen Entwicklung innerhalb einer Datenreihe zu, als dies bei der rein tabellarischen Darstellung möglich ist.

- Export der Werte. Neben der Darstellung einer Datenreihe in tabellarischer oder grafischer Form muss diese zwecks Weiterverarbeitung auch im CSV-Format (*Comma-separated values*¹) exportiert werden können. Dabei können die CSV-Daten entweder in die Zwischenablage kopiert oder in einer Datei gespeichert werden.

Der Vorteil der Verwendung von CSV ist, dass es sich dabei um ein einfaches, gleichzeitig jedoch für die vorliegenden Daten ausreichend mächtiges, Format handelt, mit dem ein Datenaustausch mit Tabellenkalkulationsprogrammen wie Microsoft Excel² und OpenOffice.org Calc³ oder ein Import in ein relationales Datenbanksystem wie MySQL⁴ erfolgen kann.

Darüber hinaus muss es mithilfe des Auswertungswerkzeuges möglich sein, alle Informationen (Messeinstellungen, ermittelte Datenreihen usw.) einer Auswertungsinstanz in einer Datei zu speichern und zu einem späteren Zeitpunkt im Werkzeug wieder zu öffnen, ohne dabei alle Einstellungen, Messungen und Berechnungen erneut vorzunehmen.

¹CSV wird definiert in RFC 4180: <http://tools.ietf.org/html/rfc4180>

²<http://office.microsoft.com/de-de/excel/default.aspx>

³<http://de.openoffice.org/product/calc.html>

⁴<http://mysql.com/>

Auch das Auswertungswerkzeug muss über **Netzwerkfähigkeit** verfügen, da bei der Verwendung von BugzillaMetrics an Qualitätsindikatoren diese Software auf einem anderen Rechner vorgehalten wird. Die Kommunikation muss über die in Abschnitt 3.2.2 geforderte Schnittstelle erfolgen.

3.3 Nichtfunktionale Anforderungen

Der folgende Abschnitt beschreibt die nichtfunktionalen Anforderungen an die zu erstellenden Werkzeuge. Es wird also festgelegt, welche Eigenschaft sie aufweisen sollen. Soweit nicht anders angegeben, beziehen sich alle Anforderungen sowohl auf den Editor zum Erstellen von Qualitätsmodellen als auch auf das Auswertungswerkzeug.

3.3.1 Systemvoraussetzungen

Das Werkzeug muss die Java Plattform in Version 5.0 voraussetzen und in Version 3.4 (*Ganymede*) der Eclipse-Entwicklungsumgebung installierbar sein. Es darf keine plattformspezifischen Funktionalitäten voraussetzen, sondern muss überall dort einsetzbar sein, wo Eclipse lauffähig ist.

3.3.2 Bedienbarkeit

Das Werkzeug muss intuitiv und einfach bedienbar sein. Ein Aspekt der Intuitivität ist die Möglichkeit zur Gestaltung der Qualitätsmodelle mithilfe eines grafischen Editors durch Anlegen von Knoten und Kanten auf einer Zeichenfläche.

Darüber hinaus muss die Erfassung umfangreicherer Informationen wie die Definition einer Berechnungsvorschrift an einem Knoten, der eine Qualitätseigenschaft repräsentiert, übersichtlich und für den Benutzer gut durchschaubar erfolgen. Die Verwendung von *Wizards* hierzu hat nicht nur den Vorteil, dass es sich dabei um einen integralen Bestandteil und ein häufig genutztes Hilfsmittel in der Eclipse-Plattform handelt. Zusätzlich hat das Design-Muster der Wizards nach [JLMS03] den Vorteil, dass der Benutzer durch einen Wizard durch komplexe Aktionen geführt wird und dadurch einen Lerneffekt erfährt. Außerdem handelt es sich bei Wizards um für die Benutzer bekannte und vertraute Elemente der Benutzerschnittstelle.

3.3.3 Geschwindigkeit

Das Werkzeug muss schnell auf Benutzereingaben reagieren und Berechnungsschritte ausführen. Durch die Abhängigkeit der Auswertungskomponente von externen Metrikwerkzeugen kann es trotzdem zu Verzögerungen und Wartezeiten bei Auswertungen kommen, sofern diese aus dem Verhalten der angebundenen Werkzeuge herrühren. So ist beispielsweise in der Anforderungs-Spezifikation ([Gra07]) für das BugzillaMetrics-Werkzeug gefordert, dass es innerhalb von maximal fünf Minuten ein Ergebnis zu liefern hat.

3.3.4 Interoperabilität

Das Werkzeug muss zwecks Förderung der Interoperabilität auf offene Standards bei der Datenhaltung setzen. Daher ist es erforderlich, dass sowohl Qualitätsmodelle als auch Ergebnisse von Auswertungen im XML-Format gespeichert werden.

3.3.5 Wartbarkeit

Es ist damit zu rechnen, dass es in Zukunft erforderlich wird, neben BugzillaMetrics weitere Messwerkzeuge zu unterstützen. Deren Integration in das Werkzeug muss mit vertretbarem Aufwand durchführbar sein.

Darüber hinaus muss das Werkzeug erweiterbar sein um neue Berechnungsvorschriften und Filterfunktionen sowie alternative Darstellungsarten für die Ergebnisse einer Auswertung.

Kapitel 4

Softwareentwicklungsprozess

In diesem Kapitel wird der Softwareentwicklungsprozess, mit dessen Hilfe und Unterstützung die Anwendung geplant und entwickelt wurde, vorgestellt. Außerdem werden die Beweggründe für dessen Einsatz erläutert. Darüber hinaus wird die Erstellung eines explorativen Prototyps zu Beginn der Implementierungsphase thematisiert.

In [IEE90] wird der Softwareentwicklungsprozess wie folgt definiert:

The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use.

[LL07, S. 88f] unterscheidet darüber hinaus zwischen einem auf einem Prozess fußenden *Projekt* als Folge ausgeführter Schritte sowie einem *Prozessmodell*, d.h. einer abstrakten Folge von Schritten, die beliebig vielen Projekten zu Grunde liegt.

4.1 Rahmenbedingungen

Die Entscheidung für oder gegen einen Softwareentwicklungsprozess wird maßgeblich durch die Rahmenbedingungen, unter denen die Softwareentwicklung ablaufen wird, beeinflusst. Im vorliegenden Fall wurden die folgenden Rahmenbedingungen identifiziert:

- Es handelt sich bei der Anwendung um eine Neuentwicklung. Vorherige Versionen oder Legacy-Systeme sind nicht zu berücksichtigen.
- Die Anwendung muss zu einem fixen, nicht verschiebbaren Zeitpunkt fertiggestellt sein.
- Zu Beginn der Entwicklungsphase ist nicht abschließend absehbar, welche konkrete Form die Auswertungskomponente annehmen wird.
- Die Entwicklung findet durch einen einzelnen Entwickler statt.
- Der Entwickler verfügt im Vorfeld der thematischen Einarbeitung über keine tiefergehenden Kenntnisse im Umfeld von Qualitätsmodellen und deren Bewertung.
- Der Entwickler verfügt im Vorfeld über keine Erfahrung im Umgang mit einigen der zu verwendenden Technologien wie GMF und der Eclipse Plug-in API.
- Die zu erstellende Anwendung wird in vorhandene Infrastruktur (Eclipse) integriert und verwendet externe Komponenten (Messwerkzeuge, BugzillaMetrics). Sowohl die vorhandene Infrastruktur als auch die extern anzubindenden Komponenten können als stabil angesehen werden. Es ist jedoch damit zu rechnen, dass zur Anbindung externer Komponenten über die eigentliche Anwendung hinaus weitere Schnittstellen und Adapter realisiert werden müssen.

4.2 Beschreibung des Softwareentwicklungsprozesses

Aus den im vorherigen Abschnitt skizzierten Rahmenbedingungen ergeben sich mehrere Konsequenzen für den zu verwendenden Softwareentwicklungsprozess.

Aufgrund der Tatsache, dass der Entwickler weder mit der Domäne der Qualitätsmodelle und deren Modellierung noch mit einigen der verwendeten Technologien vertraut ist, bietet sich die **Verwendung eines agilen Softwareentwicklungsprozesses mit iterativem Vorgehen** an. Dadurch wird auch berücksichtigt, dass die endgültigen Anforderungen an die Auswertungskomponente zu Beginn der Entwicklungsphase nicht endgültig klar

sind und somit im Rahmen einiger Iterationen weitere und genauere Anforderungen ermittelt werden können.

Ein agiler Softwareentwicklungsprozess, der stark auf ein iteratives Vorgehen setzt, ist *Extreme Programming (XP)* [BF01].

Wie im Folgenden dargelegt wird, wird im hier beschriebenen Fall nicht Extreme Programming als Ganzes übernommen, sondern es werden nur einzelne Praktiken und Konzepte, die hier anwendbar erscheinen, verwendet.

Extreme Programming

Das Extreme Programming beschreibt unter anderem eine Reihe an Praktiken, welche die Zusammenarbeit in Entwicklerteams koordinieren und verbessern sollen. Dazu zählen beispielsweise das *Pair-Programming* oder die unter dem Schlagwort *Kollektives Eigentum* beschriebene Idee, dass man nur als Team erfolgreich sein kann. Da die vorliegende Anwendung nur durch einen einzelnen Entwickler erstellt wird, bleiben diese Praktiken hier unberücksichtigt.

Nichtsdestotrotz kommt mit Subversion [Sub] eine Versionierungssoftware zum Einsatz, die ursprünglich für die Arbeit im Team konzipiert wurde. Ihre Verwendung bietet sich auch für sehr kleine Teams oder einzelne Entwickler an, da es auch in diesem Fall erforderlich sein kann, länger zurückliegende Änderungen am Programmcode nachzuvollziehen oder rückgängig zu machen.

Zwei weitere Paradigma von Extreme Programming, die zum Einsatz kommen und in den nächsten Absätzen beschrieben werden, sind das *Refactoring* und der Einsatz von *Unit Tests*:

Refactoring

Ein Paradigma von Extreme Programming ist die Idee, dass Refactorings essentieller Bestandteil der Anwendungsentwicklung sind. Es handelt sich dabei nach [LL07, S. 555ff] um eine spezielle Art der Restrukturierung von Software, die einer definierten Vorgehensweise folgt und präventiv eingesetzt wird.

Bei der Anwendung von Extreme Programming wird explizit davon ausgegangen, dass Code nicht von Anfang an perfekt ist und im weiteren Verlauf der Entwicklung an sich ändernde Gegebenheiten und neue Komponenten angepasst werden muss.

Eng verzahnt mit dem Refactoring ist die Verwendung von Unit Tests, da nur so nach Refactoringschritten sichergestellt werden kann, dass das Refactoring das beabsichtigte Verhalten der Anwendung nicht beeinträchtigt hat.

Abbildung 4.1 und der folgende Abschnitt beschreiben die Art und Weise der Verwendung von Unit Tests (speziell im Kontext von Refactoring-Maßnahmen) im vorliegenden Fall.

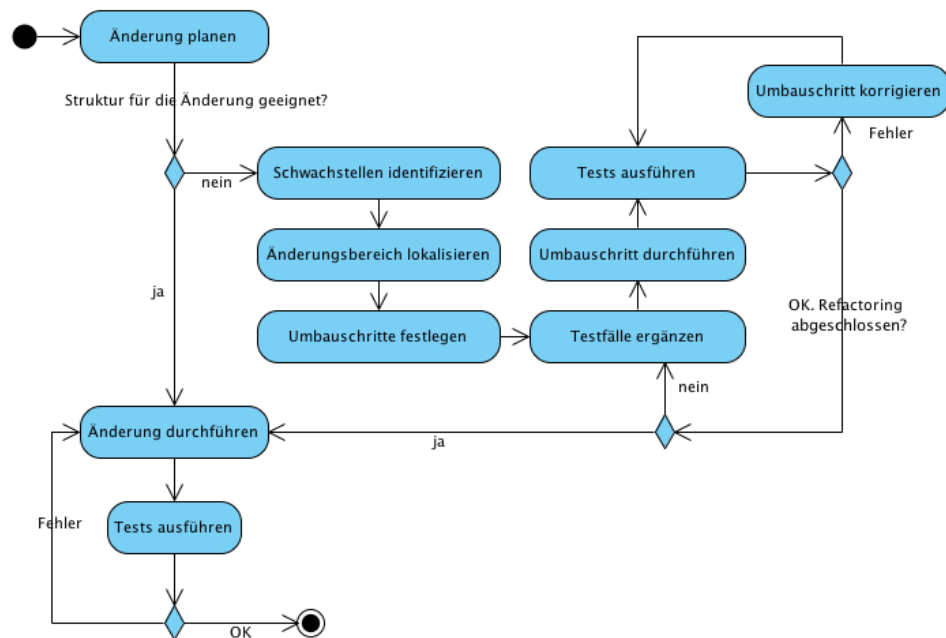


Abbildung 4.1: Integration von Unit Tests und Refactoring in den Entwicklungsablauf. Basiert auf Abb. 23-4 in [LL07, S. 556].

Unit Tests

Ein großer Teil des Programmcodes, der die Anwendung ausmacht, wird mithilfe des GMF-Rahmenwerks [GMF] und modellgetriebener Entwicklung automatisch generiert und bedarf somit keiner Unit Tests, da er per Definition korrekt ist. Andere nichtgenerierte Komponenten der Anwendung wie die Berechnungs- und Messlogik im Auswertungswerkzeug werden jedoch mit Unit Tests versehen. Nach jeder Änderung an der Codebasis sowie nach Refactorings werden diese Tests ausgeführt.

Über die einfache Verwendung von Unit Tests hinaus geht die Praxis des *Test-first development* [Som04, S. 402ff], die im vorliegenden Fall ebenfalls Anwendung erfuhr: Statt zuerst den Programmcode und erst danach die Testfälle zu schreiben, wird beim Test-first development zuerst ein Testfall oder, je nach Komplexität der zu erstellenden Komponente, mehrere Testfälle, geschrieben und auf dieser Basis der Programmcode erstellt.

Der Vorteil hiervon ist, dass dem Entwickler die Spezifikation der zu erstellenden Komponenten vollständig klar sein muss. Außerdem verfügt das Projekt hierdurch für jede Komponente über einen oder mehrere Unit Tests, die zu späteren Zeitpunkten nach Änderungen und Refactorings ausgeführt werden können. Ohne Verwendung von Test-first development droht laut [Som04] das Problem des sogenannten *test-lag*, bei dem die Entwicklung von Programmcode schneller fortschreitet als die Entwicklung der Tests und es somit mittelfristig darauf hinausläuft, dass einzelnen Komponenten nicht mit Tests versehen werden.

Im vorliegenden Fall fand Test-first development Anwendung bei der Implementierung der Berechnungs- und Messlogik in der Auswertungskomponente. Hier wurde für jeden Baustein der Implementierung zuerst ein Testgeschirr angelegt und danach erst die Komponente entwickelt. Neben der Tatsache, dass dadurch eine Anweisungsüberdeckung [LL07, S. 481] von fast 100% erreicht wurde, hatte dies den Vorteil, dass zum Zeitpunkt der Implementierung kein Benutzerinterface zum Testen der Logik notwendig war, da dies vollständig über Unit Tests erfolgen konnte.

Darüber hinaus strebt Extreme Programming *einfaches Design* der Software an. Dem wird hier Rechnung getragen, in dem die Anwendung wo nötig so durch Refactoring und Umstrukturieren angepasst wird, dass sie von möglichst einfacher Struktur und übersichtlich ist.

4.3 Entwicklungsschritte

Der Softwareentwicklungsprozess ist aufgeteilt in drei wesentliche Schritte:

1. **Ermittlung der Anforderungen**
2. **Entwurfs- und Implementierungsphase**
3. **Dokumentation**

Diese drei Schritte werden nachfolgend beschrieben.

4.3.1 Ermittlung der Anforderungen

In der Phase des Softwareentwicklungsprozesses, die sich mit der Ermittlung der Anforderungen befasst, wird in mehreren Iterationen jeweils eine Anforderungsspezifikation erstellt.

In einer ersten Iteration wurde das Konzept der bidirektionalen Qualitätsmodelle nach [SSM06] auf die Anwendbarkeit im vorliegenden Fall hin untersucht. Relevante Konzepte und Begrifflichkeiten wurden übernommen und auf die aktuelle Situation übertragen. Es ergab sich daraus eine grobe Anforderungsspezifikation, in der mit der Unterscheidung zwischen Qualitätseigenschaften und -indikatoren und der grafischen Darstellung und Modellierung der Qualitätsmodelle bereits zwei Kernanforderungen vorgesehen waren.

Basierend auf den Ergebnissen dieser ersten Iteration wurde daraufhin untersucht, welche Anforderungen sich aus der Tatsache ergeben, dass die zu erstellenden Qualitätsmodelle benutzt werden, um mithilfe von externen Auswertungswerkzeugen Messungen und Bewertungen der erreichten Qualität vorzunehmen. Außerdem wurde der Frage nachgegangen, wie sich das Konzept der Berechnungsvorschriften von Qualitätseigenschaften auf die Anforderungen auswirkt.

Die sich aus der zweiten Iteration ergebende Anforderungsspezifikation wurde Dritten, die mit der Planung und Entwicklung von Software vertraut sind, dem Entwicklungsprojekt jedoch nicht angehörten, vorgestellt und mit ihnen im Rahmen eines **Structured Walkthrough** ([LL07, S. 274]) diskutiert.

Die Ergebnisse dieser Form des technischen Reviews flossen in die dritte Iteration der Anforderungsspezifikation ein. Darüber hinaus wurden in dieser Phase weitere Anforderungen zur Anbindung empirischer Datenbanken erfasst. Die so entstandene Anforderungs-Spezifikation diente als Grundlage für die Implementierungsphase. Ihre relevanten Teile finden sich im Kapitel 3 wieder.

Nachdem im Rahmen der dritten Iteration das weitere Vorgehen im Rahmen des Entwicklungsprozesses absehbar war, wurde ein **Netzplan** [LL07, S. 107f] erstellt, der das weitere Vorgehen und die Fertigstellungszeitpunkte der Meilensteine beschrieb.

4.3.2 Entwurfs- und Implementierungsphase

In jeder Iteration der Entwurfs- und Implementierungsphase wurde eine neue Komponente entwickelt. Insgesamt erstreckte sich die Entwicklung über drei Iterationen mit den folgenden Zielen:

- Iteration 1: Erstellung eines Prototyps der Qualitätsmodell-Editor-Komponente
- Iteration 2: Erstellung der Qualitätsmodell-Editor-Komponente
- Iteration 3: Erstellung der Auswertungswerkzeug-Komponente

In Iteration 2 und Iteration 3 wurden viele weitere interne Iterationen im Sinne der von Extreme Programming vorgeschlagenen *permanenten Integration* durchgeführt.

In den folgenden Abschnitten werden die einzelnen Hauptiterationen detaillierter beschrieben.

Prototyp der Qualitätsmodell-Editor-Komponente

Zu Beginn der Entwicklungsphase wurde ein Prototyp des Qualitätsmodell-Editors erstellt. Das vorrangige Ziel war die Minimierung von Projektrisiken, die aufgrund unbekannter Technologien drohten.

Die am Projekt Beteiligten verfügten vor Aufnahme der Entwicklungsarbeiten über keine Erfahrung in der Arbeit mit den EMF- und GMF-Rahmenwerken, die kritische und wichtige Komponenten der zu erstellenden An-

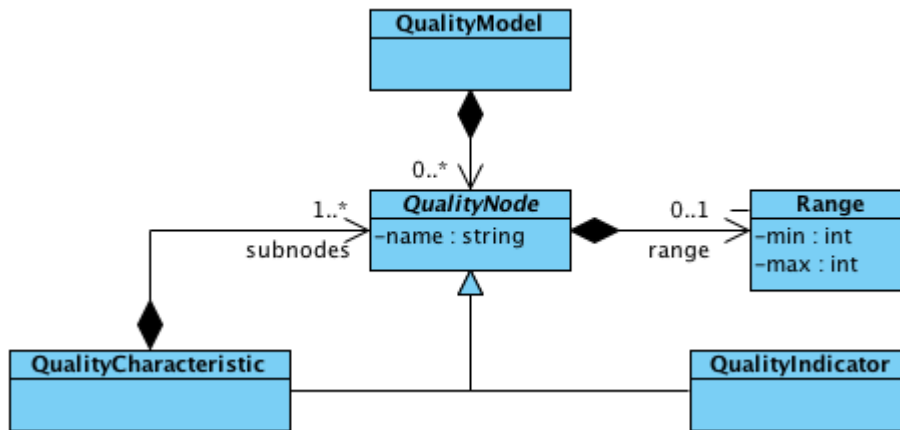


Abbildung 4.2: Domänen-Modell des Prototypen

wendung ausmachen. Um frühzeitig erste Erfahrungen mit diesen Rahmenwerken und der Erstellung von Plug-ins für die Eclipse-Plattform im Allgemeinen zu sammeln sowie Schwachstellen bzw. offenen Fragen aufzudecken, wurde ein Prototyp in Form eines Plug-in erstellt, der einige zentrale Komponenten des Domänen-Modells zur Repräsentation von Qualitätsmodellen implementiert. Der Ausschnitt dieser zentralen Komponenten ist in Abbildung 4.2 wiedergegeben.

Der Hauptfokus bei der Erstellung lag dabei auf der Modellierung des Entwurfsmusters Kompositum ([GHJV95]) zwischen den Klassen `QualityNode` und `QualityCharacteristic` und in der Erstellung eines einfachen Wizard-Dialogs, der über die Kontextmenüs von Knoten, die Instanzen von `QualityCharacteristic` repräsentieren, gestartet werden kann. Darüber hinaus wurde im Prototypen die Assoziation von `Range`-Instanzen mit Instanzen der Klasse `QualityCharacteristic` programmatisch umgesetzt. Dem Lerneffekt war dies insofern zuträglich, als dass es sich bei `Range` um eine Klasse handelt, die zwar zum Domänen-Modell gehört, jedoch nicht auf ein grafisches Element im Editor abgebildet wird und somit über das EMF-API erzeugt und eingebunden werden muss.

Um diese Schritte umzusetzen, war ein frühes Einarbeiten in EMF und GMF ebenso erforderlich wie das Auseinandersetzen mit dem generierten Quellcode des grafischen GMF-Editors, um den Wizard einzubinden und an geeignete

ter Stelle eine Instanz der **Range**-Klasse zu erzeugen und mit einem Knoten zu assoziieren. Gleichzeitig hat der Prototyp trotzdem einen starken Bezug zur zu erstellenden Anwendung, wodurch bereits zu dem Zeitpunkt, da der Prototyp erstellt wurde, konkretes Wissen und Ideen für die Umsetzung des Qualitätsmodell-Editors gesammelt werden konnten.

Erstellung der Qualitätsmodell-Editor-Komponente

Aufbauend auf den Erkenntnissen, die bei der Entwicklung des Wegwerfprototypen gesammelt worden waren, wurde der Qualitätsmodell-Editor implementiert. Es bot sich an, mit dieser Komponente zu beginnen, da somit ein Werkzeug zur Verfügung stand, das mit Qualitätsmodellen Testeingabedaten für das Auswertungswerkzeug erzeugen konnte.

Erstellung der Auswertungswerkzeug-Komponente

Im Anschluss an die Fertigstellung einer ersten Version des Qualitätsmodell-Editors wurde mit der Entwicklung des Auswertungswerkzeuges begonnen.

In diesem Zusammenhang wurde darüber hinaus BugzillaMetrics um eine XML-RPC-Schnittstelle erweitert, von der dann nicht nur das Auswertungswerkzeug sondern auch die nachfolgenden Versionen des Qualitätsmodell-Editors Gebrauch machten.

4.3.3 Dokumentation

Die Dokumentation der Anforderungen und Architektur sowie eine Evaluierung der resultierenden Anwendungen erfolgte projektbegleitend. Eine benutzerbezogene Dokumentation in Form eines *User Guide* wurde nach Abschluss der Entwicklung erstellt.

Kapitel 5

Architektur

Dieses Kapitel beschreibt die Architektur und das Design der zu erstellenden Anwendung.

Nach [LL07, S. 381] beschreibt die Architektur einer Software deren Komponenten und ihre Beziehung, vergleichbar zur tragenden Struktur eines Gebäudes.

Dieses Kapitel gliedert sich in die folgenden Teile:

1. **Allgemeine Architektur.** Der erste Teil beschreibt die allgemeine, grobe Architektur und die Idee der Zerlegung in zwei Hauptkomponenten
2. **Architektur des Qualitätsmodell-Editors.** Teil zwei dieses Kapitels beschreibt die Architektur und das Design des Qualitätsmodell-Editors.
3. **Architektur der Auswertungskomponente.** Der dritte Teil des Kapitels beschreibt mit der Architektur der Auswertungskomponente die zweite Hauptkomponente.
4. **Architektur der XML-RPC-Schnittstelle für BugzillaMetrics.** Der vierte und letzte Abschnitt beschreibt die Architektur der XML-RPC-Schnittstelle, um die BugzillaMetrics erweitert wurde, damit die Auswertungskomponente auf entfernten BugzillaMetrics-Installationen arbeiten kann.

5.1 Allgemeine Architektur

Gemäß der im Rahmen der Anforderungsanalyse ermittelten Anforderungen muss die zu erstellende Anwendung in Form eines oder mehrerer Plug-Ins in die Entwicklungsumgebung Eclipse integriert werden. Sie soll es ermöglichen, auf grafischem Weg Qualitätsmodelle zu definieren und zu bearbeiten sowie Messungen und Auswertungen auf Basis dieser Qualitätsmodelle durchzuführen.

Da die Anwendung mit dem Qualitätsmodell-Editor und der Auswertungskomponenten aus zwei Hauptkomponenten besteht, liegt es nahe, für jede der Komponenten ein eigenes Plug-In zu entwickeln.

Für diese Entscheidung gibt es mehrere Gründe:

- Das Erstellen und Bearbeiten von Qualitätsmodellen setzt ein Vorhandensein der Auswertungsfunktionalität nicht voraus. Gleiches gilt im Umkehrschluss für die Auswertungskomponente, bei der nur erforderlich ist, dass ein Qualitätsmodell vorliegt, nicht jedoch, dass es auch mit dem gleichen Werkzeug bearbeitet werden kann.
- Die beiden Komponenten werden durch verschiedene Benutzerrollen verwendet: Der Qualitätsmodell-Editor wird durch Qualitätsmanager verwendet, um Modelle zu erstellen und zu bearbeiten, während die Auswertungskomponenten durch Projektmanager benutzt wird, die auf Basis vorgegebener Qualitätsmodelle Messungen und Auswertungen für die ihnen betrauten Projekte und Produkte durchführen.

Durch die Zerlegung in zwei Hauptkomponenten wird eine Trennung von Zuständigkeiten erreicht, durch die sich die jeweiligen Benutzerrollen unabhängig voneinander und (an räumlich und technisch getrennten Systemen) voll auf die ihnen aufgetragenen Aufgaben konzentrieren können.

Das Design und die Architektur beider Hauptkomponenten werden im weiteren Verlauf dieses Kapitels erläutert.

5.2 Architektur des Qualitätsmodells-Editors

Die Architektur des Qualitätsmodell-Editors setzt die in der Anforderungsanalyse beschriebenen Konzepte der bidirektionalen Qualitätsmodelle sowie der Berechnungsvorschriften um. Außerdem berücksichtigt sie, dass Instanzen des Qualitätsmodells in einem graphbasierten Editor bearbeitet werden können, indem sie ein entsprechendes Frontend bereitstellt.

5.2.1 GMF

Die Grundlage des Qualitätsmodell-Editors bildet das Rahmenwerk GMF [GMF], das es ermöglicht, grafische Editoren für formale Modelle als Plug-Ins für die Eclipse-Plattform zu entwickeln. GMF kapselt dabei mit EMF [EMF], [BSM⁺03] und GEF [GEF] zwei weitere Rahmenwerke aus dem Eclipse-Projekt:

- Bei EMF handelt es sich um ein Rahmenwerk zur Generierung von Programmcode auf Basis von Modellen. Es handelt sich somit dabei um ein Werkzeug zur modellgetriebenen Softwareentwicklung (Model-Driven Software Development, MDD), bei der aus formalen Modellen automatisiert Code erzeugt wird. Im Fall von EMF dient dieser Code zur Arbeit mit und auf den Modellen.

Die strukturierten Modelle werden im Fall von EMF als Instanzen des Ecore-Metamodells [BSM⁺03, S. 14] formuliert, das sich stark an die Elemente und die Semantik der Klassendiagramme in der UML 2.0 [Fow03] anlehnt. Dadurch ist es möglich, die UML-Diagramme, die im Rahmen der Entwurfsphase erstellt werden, um das Design der Anwendung auf einem hohen Level für die am Projekt beteiligten zu beschreiben, mittels Model-to-Model-Transformationen in Ecore-Instanzen zu übertragen. Mithilfe von Model-to-Code-Transformationen wird aus den Ecore-Instanzen dann lauffähiger Code erzeugt.

Der durch EMF generierte Programmcode kann Instanzen des Modells anlegen, diese manipulieren, gegen Regeln validieren, auf Änderungen überwachen und zwecks Speicherung und Austausch mit anderen Systemen in verschiedene Formate wie zum Beispiel das XML-basierte XMI-Format serialisieren.

- GEF (*Graphical Editing Framework*) ist ein Rahmenwerk, um grafische Anwendungen auf existierenden Datenmodellen für die Eclipse-

Plattform zu erstellen.

GEF stellt nicht nur die Funktionalität für das Erzeugen graphischer Objekte innerhalb der Eclipse-Plattform zur Verfügung, sondern bietet auch eine Implementierung des MVC-Musters für Anwendungen und Plug-Ins, die es verwenden. Hierauf wird detailliert im Abschnitt 5.2.5 eingegangen.

Das Rahmenwerk unterscheidet sich von Bibliotheken wie SWT (*Standard Widget Toolkit*), mit dem die grafischen Bedienelemente wie Buttons und Textfelder in Eclipse gestaltet werden, dadurch, dass es keine standardisierten Element (die so genannten Widgets) zur Verfügung stellt, sondern es dem Entwickler erlaubt, individuelle Zeichenformen als grafische Elemente des zu erstellenden Editors zu definieren.

Das GMF-Rahmenwerk kombiniert beide Ansätze, um grafische Diagramm-Editoren für Ecore-basierte formale Modelle zu erstellen.

Eine modellgetriebene Entwicklung unter Verwendung von GMF folgt dabei stets einem festgelegten Arbeitsablauf, der im Folgenden beschrieben und in Abbildung 5.1 wiedergegeben ist.

1. **Erstellung eines Domänen-Modells.** Das als Instanz des Ecore-Metamodells erstellte Domänen-Modell beschreibt die Bestandteile und Strukturen des dem Editor zugrunde liegenden Modells. Üblicherweise gibt das Modell die Begriffswelt einer bestimmten Domäne wieder; im vorliegenden Fall handelt es sich bei dieser Domäne um bidirektionale Qualitätsmodelle und deren Auswertung.

Durch die starke Anlehnung an UML ist es bei vielen UML-Werkzeugen möglich, UML-Klassendiagramme direkt in das Ecore-Format zu transformieren. Alternativ können Ecore-Modelle auch in Eclipse durch entsprechende Werkzeuge erstellt oder durch mit Annotationen versehenen Java-Code beschrieben werden.

2. **Erstellung der grafischen Definition.** Die grafische Definition ist ein Modell, das Informationen darüber enthält, welche grafischen Elemente im GEF-basierten Editor verwendet werden und welches Aussehen sie haben. Zu den grafischen Elementen im vorliegenden Fall zählen

beispielsweise die Knoten, die im Editor die Qualitätsindikatoren und die Qualitätseigenschaften repräsentieren.

3. **Erstellung der Tooling-Definition.** Die sog. Tooling-Definition beschreibt, welche Werkzeuge zum Erstellen neuer grafischer Elemente im GEF-basierten Editor verwendet werden können. Außerdem kann darüber Einfluss genommen werden auf das Aussehen von Menüs und Werkzeugleisten des Editors.
4. **Erstellung des Mapping-Modells.** Das Mapping-Modell kombiniert das Domänen-Modell, die grafische Definition und die Tooling-Definition. In ihm wird festgelegt, welche grafischen Elemente im Editor welchen Domänen-Elementen im Ecore-Modell zugeordnet werden und mit welchen Werkzeugen aus der Tooling-Definition diese angelegt werden.

Darüber hinaus ist es im Mapping-Modell möglich, auf Basis der Object Constraint Language (OCL) Einschränkungen und Validierungsregeln zu formulieren, die für die zu erstellenden Domänenmodelle

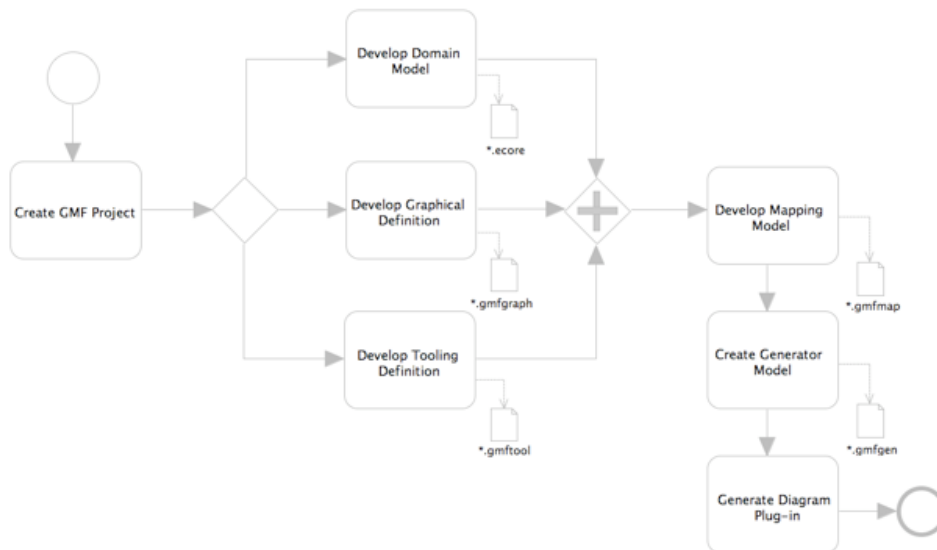


Abbildung 5.1: Überblick über die Arbeitsschritte bei modellgetriebener Softwareentwicklung mit GMF. (Aus: [GMF])

gelten müssen. Die Validierung mittels dieser Regeln kann aus dem GMF-basierten Editor heraus gestartet werden und ist sauber in das Benutzerinterface von Eclipse integriert.

5.2.2 Qualitätsmodelle, -eigenschaften und -indikatoren

Das Qualitätsmodell in seiner Gesamtheit wird in der Architektur durch die Klasse `QualityModel` repräsentiert. In der Eigenschaft `entryNodes` werden die Qualitätseigenschaften und die Qualitätsindikatoren, aus denen das Qualitätsmodell besteht, referenziert.

Qualitätseigenschaften entsprechen dem Domänen-Element `QualityCharacteristic`, während Qualitätsindikatoren durch die Klasse `QualityIndicator` repräsentiert werden. Beide verfügen über eine gemeinsame Oberklasse `QualityNode`, in der die Gemeinsamkeiten beider Klassen modelliert werden: Sowohl Qualitätseigenschaften wie auch Qualitätsindikatoren verfügen über eine Bezeichnung, die im Attribut `name` erfasst wird, sowie über eine Beschriftung im Attribut `description`. Das Attribut `guid` steht für *Globally Unique Identifier* und enthält eine 128 Bit lange, zufällig generierte ID für jeden Knoten.

Exemplarisch für den Rest der Architektur sei hier auf die Bedeutung der Kompositions-Assoziation zwischen `QualityModel` und `QualityNode` verwiesen. In der UML dient sie als Notation zur Beschreibung der Beziehung zwischen einem Ganzen und dessen Teilen. Entscheidend ist in diesem Fall, dass die Existenz der Teile von der Existenz des Ganzen abhängt.

Darüber hinaus hat ihre Verwendung auch direkte Auswirkungen bei der Transformation in ein formales Ecore-Modell. Bei Referenzen, die in UML über Komposition-Assoziationen modelliert werden, handelt es sich nach der Ecore-Transformation um so genannte *Containments*. Für solche wird durch EMF sichergestellt, dass die referenzierten Teile der Teile-Ganzes-Beziehung in der gleichen Ressource (im Standardfall sind Ressourcen im vorliegenden Fall XMI-Dateien) persisiert werden wie das übergeordnete Objekt, hier also die Instanz von `QualityModel`. Der Grund für die Verwendung der Kompositions-Assoziation ist, dass bei der Implementierung ein explizites Hinzufügen von `QualityNode`-Instanzen zu einer Ressource nicht notwendig ist, da sie bereits implizit der gleichen Ressource wie der `QualityModel`-Instanz, in deren `entryNodes`-Attribut sie referenziert werden, zugeordnet sind.

Zur Umsetzung der Tatsache, dass Qualitätsmodelle als azyklische Graphen interpretierbar sind, wird das Kompositum-Entwurfsmuster [GHJV95] verwendet, um Kanten von beliebigen Qualitätseigenschaften oder Qualitätsindikatoren zu Qualitätseigenschaften zu realisieren. Referenzen auf über eine Kante angebundene Instanzen von `QualityNode` werden dabei in der jeweiligen Qualitätseigenschaft im Attribut `subnodes` erfasst. Eine Navigation ist daher nur von Qualitätseigenschaften zu deren zugeordneten Knoten möglich, nicht jedoch in der anderen Richtung. Im Rahmen der Umsetzung wurde dies auch als nicht erforderlich identifiziert. Darüber hinaus wäre eine umgekehrte Navigation durch die Tatsache, dass es sich bei Qualitätsmodellen um Graphen und nicht Bäume handelt und somit mehr als einen übergeordneter Knoten existieren kann, nicht intuitiv möglich.

Der Grund für die Verwendung des Kompositum-Musters ist, dass im Modell keine eigene Klasse zur Repräsentation von Kanten notwendig ist. Dies ist nicht erforderlich, da mit einer Kante keine weitergehenden Informationen verknüpft sind und sie somit reines Hilfsmittel zur Darstellung von Kanten gewesen wären. GMF unterstützt sowohl die Modellierung von Kanten mit eigenen Klassen als auch mit Referenzen. Man spricht dabei von *Type based links* bzw. *Reference based links*. Darüber hinaus wird durch die Art und Weise der Modellierung des Muster bereits eingeschränkt, dass nur Qualitätseigenschaften über eingehende Kanten verfügen können.

5.2.3 Wertebereiche von Qualitätsknoten

Jeder Qualitätsindikator und jede Qualitätseigenschaft kann mit einem Wertebereich versehen werden, der bestimmt, welche oberen und unteren Grenzen die Werte haben, die gemessen bzw. berechnet werden.

Um dies in der Architektur zu realisieren, verfügt `QualityNode` über eine Assoziation zur Klasse `Range`, die einen Wertebereich repräsentiert. Da die oberen und unteren Schranken sowohl endlich als auch nichtendlich sein können, wird deren Repräsentation in eine eigene Klassenstruktur mit einem Interface `Bound` und zwei konkreten Klassen `BoundRegular` für endliche Schranken und `BoundInfinite` für nichtendliche Schranken ausgelagert. Ob es sich bei einer Instanz von `BoundInfinite` um eine Schranke mit dem Wert $-\infty$ oder mit dem Wert $+\infty$ handelt, wird nicht weitergehend modelliert, sondern über den Kontext der Verwendung der Instanz entschieden.

Ein weiterer Vorteil dieses Architekturansatzes ist, dass eine Klasse zur Repräsentation von Wertebereichen mit garantiert endlichen Schranken einfach realisiert werden kann. Siehe dazu die Beschreibung der Filterfunktionen für Bewertungsvorschriften.

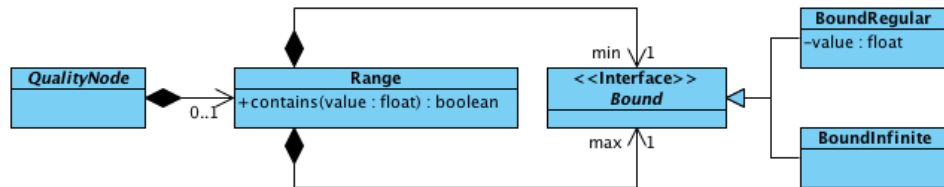


Abbildung 5.2: Wertebereiche von Qualitätsknoten

Die Struktur von Wertebereichen für Qualitätsknoten ist ebenfalls in Abbildung 5.2 wiedergegeben.

Darüber hinaus ist es bei Qualitätsindikator-Knoten möglich, festzulegen, ob große oder kleine Werte der betrachteten Metrik besser sind. Hierzu dient das Attribut `betterValue`, das als Wert eine Instanz von abstrakten Basistypen `BetterValue` annimmt. Die drei davon abgeleiteten Klassen `BetterValueSmallest`, `BetterValueLargest` und `BetterValueNotDefined` (hierbei handelt es sich um den Standardwert) repräsentieren die möglichen konkreten Werte. Der Grund für dieses Design, dass in zukünftigen Versionen des Qualitätsmodell-Editors weitere Arten von besseren Werten einfach hinzugefügt werden können.

5.2.4 Mess- und Berechnungsvorschriften

Mess- und Berechnungsvorschriften geben an, auf welche Art und Weise die Werte an einem Qualitätsknoten im Rahmen einer Auswertung ermittelt werden.

5.2.4.1 Gemessene und berechnete Werte

Bei den Mess- und Berechnungsvorschriften wird unterschieden zwischen den Vorschriften bei Qualitätsindikatoren, die beschreiben, wie Werte mithilfe eines Auswertungswerkzeuges wie `BugzillaMetrics` *gemessen* werden,

und den Vorschriften bei Qualitätseigenschaften, durch die festgelegt wird, auf welche Art und Weise deren Werte *berechnet* werden.

In der Architektur wird dieser Tatsache entgegengekommen, in dem alle Vorschriften eine gemeinsame Basisklasse `ValueSpecification` haben. Für Messvorschriften bei Qualitätsindikatoren gibt es darüber hinaus eine von `ValueSpecification` abgeleitete abstrakte Klasse `ValueSpecificationRetrievedValue`, während Berechnungsvorschriften für Qualitätseigenschaften von `ValueSpecificationCalculatedValue` abgeleitet werden.

Es ist nicht nur möglich, vom entsprechenden Qualitätsknoten zu dessen Mess- bzw. Berechnungsvorschrift zu navigieren, sondern es ist ebenfalls möglich, von einer `ValueSpecification`-Instanz zurück zu der Instanz von `QualityIndicator` bzw. `QualityCharacteristic` zu navigieren, die sie referenziert.

5.2.4.2 Messvorschriften für Qualitätsindikatoren

Alle Messvorschriften, die für Instanzen von `QualityIndicator` definieren, auf welche Art und Weise die Werte gemessen werden, sind Spezialisierungen der abstrakten Basisklasse `ValueSpecificationRetrievedValue`.

Der Grund für diese Architektur ist eine einfache Erweiterbarkeit um neue Mess-Interfaces durch Anlegen einer neuen Spezialisierung von `ValueSpecificationRetrievedValue`.

In der aktuellen Ausbaustufe der Anwendung sind zwei Messvorschriften für Qualitätsindikatoren in der Architektur vorgesehen und implementiert. Diese sind in Abbildung 5.3 wiedergegeben und werden im Folgenden beschrieben.

- Die Klasse `ValueSpecificationRetrievedValueBugzillaMetrics` definiert, dass mithilfe des `BugzillaMetrics`-Werkzeuges Werte für den betreffenden Qualitätsindikator gemessen werden.

Die Klasse verfügt über drei Attribute, welche einen Teil der Informationen über die von `BugzillaMetrics` zu verwendende Metrik spezifizieren:

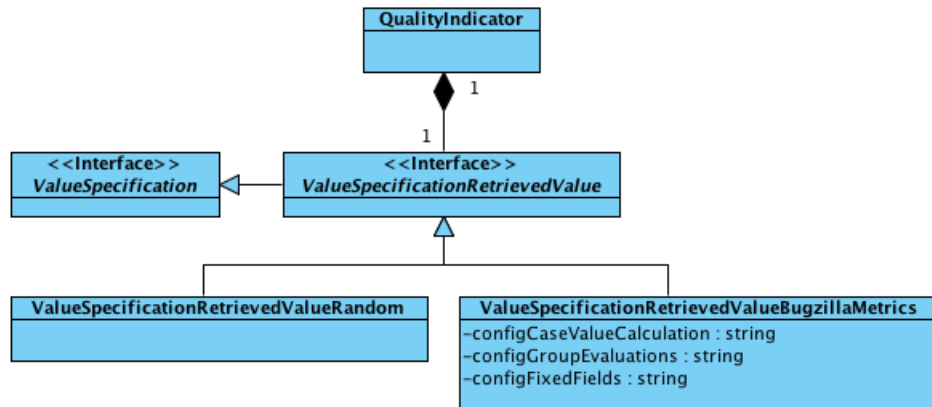


Abbildung 5.3: Messvorschriften für Qualitätsindikatoren

- `configCaseValueCalculations` enthält die XML-Darstellung der *Case Value Calculations*.
- `configGroupEvaluations` enthält die XML-Darstellung der *Group Evaluations*, die beschreibt, wie die Ergebnisse der Case Value Calculation kombiniert bzw. gruppiert werden.
- `configFixedFields` enthält die XML-Darstellung der *Fixed Fields*.

Eine Erläuterung zu diesen Konfigurationsfeldern findet sich im Kapitel Grundlagen in Abschnitt 2.4.3. Die Konfiguration für BugzillaMetrics wird zur Laufzeit im Rahmen der Erstellung einer Auswertung um weitere Parameter ergänzt.

- Die Klasse `ValueSpecificationRetrievedValueRandom` wurde vornehmlich in der Entwicklungsphase zu Testzwecken verwendet, als die Anbindung an BugzillaMetrics noch nicht realisiert war.

Wird die durch diese Klasse induzierte Messvorschrift verwendet, so sind die Messwerte des entsprechenden Qualitätsindikators Zufallszahlen, die im für den Indikator festgelegten Wertebereich liegen.

Durch diese Hilfs-Messvorschrift war bereits früh eine Auswertung von Qualitätsmodellen basierend auf Zufallszahlen möglich, ohne dass die aufwändige BugzillaMetrics-Anbindung benötigt wurde.

5.2.4.3 Berechnungsvorschriften für Qualitätseigenschaften

Bei den Berechnungsvorschriften, die für Qualitätseigenschaften festlegen, wie deren Werte im Rahmen einer Auswertung berechnet werden, handelt es sich um Klassen, die das Interface `ValueSpecificationCalculatedValue` implementieren.

Dabei erfolgt eine weitere Unterteilung in Unterklassen von `FilterFunction` und `ValueSpecificationComplexFunction` je nachdem, ob es sich bei der Berechnungsvorschrift um eine Filterfunktion oder eine komplexe Funktion im Sinne von Abschnitt 3.2.1.1 der Anforderungsanalyse handelt.

Filterfunktionen

Filterfunktionen werden als Filter in Argumenten komplexer Funktionen referenziert. Es handelt sich bei Filterfunktionen im Domänenmodell um Implementierungen der abstrakten Klasse `FilterFunction`, die wiederum das Interface `ValueSpecificationCalculatedValue` implementiert. Der Grund für dieses Design ist, dass so die Anwendung einfach um neue Filterfunktion erweitert werden kann.

Die Architektur der Filterfunktionen ist in Abbildung 5.4 wiedergegeben.

Alle Filterfunktionen beziehen sich auf eine Berechnungsvorschrift, deren Wert durch sie gefiltert wird. Diese Assoziation wird durch das Attribut `argument` von `FilterFunction` ausgedrückt.

Gemäß der Anforderungsanalyse wird zwischen fünf Filterfunktionen unterschieden:

- **Identische Abbildungen** werden durch die Klasse `FilterFunctionAbsolute` implementiert.
- **Quantil-Filter** werden durch die Klasse `FilterFunctionQuantile` implementiert.
- **Skalierungs-Filter** werden durch die Klasse `FilterFunctionNormalize` implementiert und entsprechen einer Skalierung der Form

$$x_{norm} = \left[(x - min) \cdot \frac{max_{norm} - min_{norm}}{max - min} \right] + min_{norm}, \quad (5.1)$$

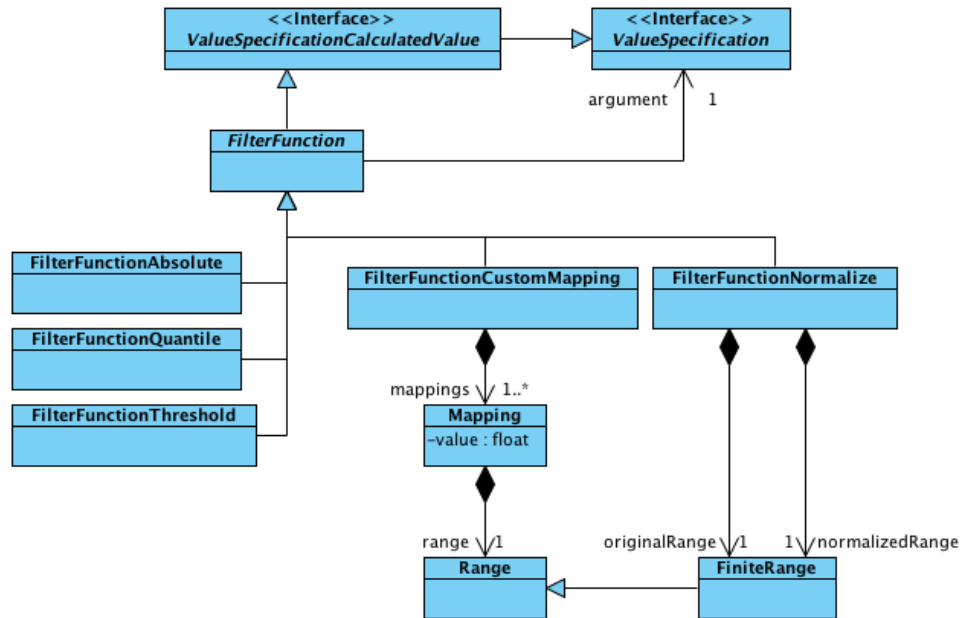


Abbildung 5.4: Architektur der Filterfunktionen

Die Werte von (min, max) und (min_{norm}, max_{norm}) werden durch zwei Referenzen auf Instanzen von `FiniteRange` festgelegt. Dabei handelt es sich um eine Spezialisierung der Wertebereichsklasse `Range`, welche nur endliche obere und untere Schranken zulässt. Mit unendlichen Werten für die obere oder untere Schranke ist eine Skalierung nicht möglich.

Der Standardfall sieht vor, dass es sich bei `originalRange` um den Wertebereich handelt, der für den zu filternden Qualitätsknoten konfiguriert ist. Daher sind im Frontend die entsprechenden Eingabefelder damit vorbelegt.

- Der Filter zur **Abbildung auf benutzerdefinierte Wertebereiche** wird realisiert durch die Klasse `FilterFunctionCustomMapping`.

Instanzen dieser Klasse werden parametrisiert durch eine Menge von

Instanzen der Klasse `Mapping`. Es handelt sich bei `Mappings` um Container, die einen Wertebereich vom Typ `Range` auf einen skalaren numerischen Wert `value` abbilden. Eine Instanz von `FilterFunctionCustomMapping` ist pro benutzerdefiniertem Wertebereich mit jeweils einer Instanz von `Mapping` assoziiert.

Eine direkte Verwendung des `Hashtable`- oder `HashMap`-Datentyps aus dem Java Development Kit (JDK) zur Realisierung der Abbildung von Wertebereichen auf skalare Werte ist in diesem Fall nicht möglich, da EMF hierfür keine Möglichkeit zur Serialisierung bietet.

- Der **Schwellenwert-Filter** wird durch die Klasse `FilterFunctionThreshold` repräsentiert. Er kapselt drei Datenfelder, die in Form von Fließkommazahlen den Schwellenwert, den Wert, auf den alle Eingabe, die kleiner als der Schwellenwert sind, abgebildet werden sowie den Wert, auf den Eingaben, die größer als der Schwellenwert sind, abgebildet werden, enthalten.

Komplexe Funktionen

Die Verwendung einer komplexen Funktion als Berechnungsvorschrift einer Qualitätseigenschaft induziert, dass der Wert dieser Qualitätseigenschaft aus mehreren Werten der untergeordneten Qualitätsknoten berechnet wird.

Die Architektur der komplexen Funktionen ist in Abbildung 5.5 wiedergegeben.

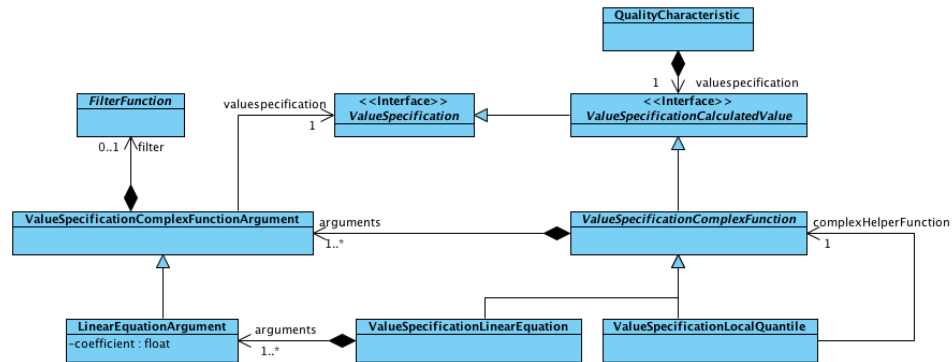


Abbildung 5.5: Architektur der komplexen Funktionen

Alle komplexen Funktionen sind Spezialisierungen der abstrakten Klasse `ValueSpecificationComplexFunction`.

Gemäß der Anforderungsanalyse handelt es sich bei komplexen Funktionen um mehrstellige Abbildungen, d.h. aus mehreren Eingabewerten wird ein Ausgabewert berechnet. Dieser Anforderung wird mit dem Konzept der Argument-Klassen begegnet:

Ein Argument einer komplexen Funktion besteht grundsätzlich aus der Berechnungsvorschrift einer untergeordneten Qualitätseigenschaft oder eines untergeordneten Qualitätsindikators sowie einer optionalen Filterfunktion, die vor der Weiterverarbeitung den Wert des entsprechenden untergeordneten Qualitätsknotens filtert.

Die Umsetzung dieser Idee in der Architektur erfolgt mithilfe der Klasse `ValueSpecificationComplexFunctionArgument` sowie deren Unterklassen. Eine Instanz dieser Klasse referenziert über das Attribut `valuespecification` die Berechnungsvorschrift des untergeordneten Qualitätsknoten, der durch sie in der komplexen Funktion repräsentiert wird. Das Attribut `filter` verweist auf eine Instanz von `FilterFunction`, welche den optionalen Filter definiert.

Je nach Art der komplexen Funktion müssen die Argumente weitere, funktionsspezifische Attribute enthalten. In diesem Fall wird `ValueSpecificationComplexFunctionArgument` durch entsprechende Unterklassen erweitert. Ein Beispiel dafür ist `LinearEquationArgument`, die im Folgenden im

Zusammenhang mit linearen Gleichungen als komplexe Funktion beschrieben wird.

Derzeit sind die folgenden komplexen Funktion in der Architektur vorgesehen:

- Die Klasse `ValueSpecificationLinearEquation` definiert die Bewertungsvorschrift für eine Qualitätseigenschaft als **lineare Gleichung** der Form

$$q(x_1, \dots, x_n) = c_1x_1 + c_2x_2 + c_3x_3 + \dots + c_nx_n. \quad (5.2)$$

Dazu ist es erforderlich, für jedes Argument x_1, \dots, x_n die Koeffizienten c_1, \dots, c_n zu erfassen und zu speichern. Hierzu verwendet die Klasse eine angepasste Argument-Klasse `LinearEquationArgument`, die neben der Berechnungsvorschrift und dem Filter auch den Wert des entsprechenden Koeffizienten enthält.

- **Quantil-Berechnungen** werden in Instanzen des Qualitätsmodells definiert durch die Klasse `ValueSpecificationLocalQuantile`.

Wie in der Anforderungsanalyse festgehalten, benötigt diese komplexe Funktion eine weitere komplexe Funktion als Hilfsfunktion zur Berechnung der lokalen empirischen Datenbasis sowie des absoluten Wertes der Qualitätseigenschaft. Hierzu dient die Referenz `complexHelperFunction`, welche auf die zu verwendende Hilfsfunktion verweist.

Diese komplexe Funktion benötigt keine angepasste Argument-Klasse, da die Referenz auf die komplexe Hilfsfunktion nicht pro Argument festgelegt wird.

5.2.5 Frontend

Die Architektur der Frontends des Qualitätsmodell-Editors wird maßgeblich durch die Verwendung von GEF im GMF-Rahmenwerk beeinflusst.

GEF induziert für die auf ihm basierenden Anwendungen das **Architekturmuster** nach [LL07, S. 382] **Model/View/Controller (MVC)**. Es

handelt sich dabei um ein weit verbreitetes Muster zur Konstruktion interaktiver Anwendungen.

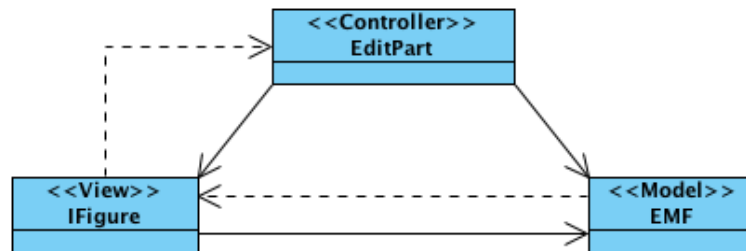


Abbildung 5.6: MVC-Architekturmuster in GMF-basierten Anwendungen

- Beim **Model** handelt es sich dabei um eine Instanz des Domänen-Modells, die im Editor erstellt bzw. bearbeitet werden soll.
- Für alle Elemente des Domänen-Modells, die auf grafischem Wege im Editor bearbeitet werden sollen, muss ein **View** bereitgestellt werden, der das Interface **IFigure** der `draw2d`-Bibliothek, die Bestandteil der Eclipse-Plattform ist, implementiert.
- Ebenso muss für jedes Domänen-Element, das grafisch editierbar ist, ein **Controller** implementiert werden, der das Interface **EditPart** implementiert. Im Fall von GMF ist es in der Regel ausreichend, direkt das davon abgeleitete Interface **GraphicalEditPart**, das für Controller von grafischen Elementen konzipiert ist, zu implementieren.

Die `EditParts` übernehmen nicht nur die Aufgabe, Änderungen aus dem Model an die entsprechenden Views zu delegieren und Änderungen an einem View an das Model weiterzureichen, sondern bieten auch die Möglichkeit, im Rahmen des Observer-Entwurfsmuster eigene, nicht automatisch generierte Klassen als Beobachter für Änderungen im View oder im Model zu registrieren.

Außerdem fungieren die `EditParts` als Fassade zum Zugriff auf Model und View in individuellen Erweiterungen der grafischen Editoren durch zum Beispiel Konfigurationswizards (siehe unten).

Neben dem MVC-Architekturmuster und den oben genannten Entwurfsmustern werden weitere Muster in GEF verwendet:

- Das Verhalten der EditParts wird festgelegt durch Unterklassen des Interface `EditPolicy`, von denen eine oder mehrere pro EditPart-Controller registriert werden. Durch das Entwurfsmuster **Zuständigkeitskette** (*Chain of Responsibility*) überprüft eine EditPolicy, ob sie die an den Controller gerichtete (Änderungs-)Anfrage bearbeiten kann. Kann sie dies, ist die Rückgabe ein Objekt, das eine Aktion und deren Parameter kapselt und einer Implementierung des **Kommando**-Entwurfsmusters entspricht. Andernfalls wird die Anfrage an die nächste EditPolicy weitergereicht.
- Das Anlegen neuer Domänen-Elemente und neuer grafischer Elemente im Editor erfolgt durch das **Factory**-Muster.

Bei der Verwendung von Rahmenwerken zur Entwicklung von Anwendungen wird bezüglich der Erweiterbarkeit unterschieden zwischen Blackbox- und Whitebox-Erweiterbarkeit.

Bei GMF handelt es sich im Bezug darauf um einen Hybrid: Im Sinne der Blackbox-Erweiterbarkeit ist es möglich, viele Anpassungen bereits im Rahmen des Erstellung des grafischen Modells, der Tooling-Definition und des Mapping-Modells vorzunehmen. Gleichzeitig ist es jedoch durch das Konzept der Extension Points in Eclipse, durch die Möglichkeit der Anpassung des generierten Codes und durch Registrieren eigener Listener in den Controllern möglich, nach dem Prinzip der Whitebox-Erweiterung an *Hot Spots* [Pre97] Anpassungen vorzunehmen. Nach [LL07, S. 417] spricht man in diesem Fall auch von einer *Grey Box*.

5.2.5.1 Konfigurationswizards

Im Qualitätsmodell-Editor muss es möglich sein, Eigenschaften von Qualitätsknoten durch interaktive Dialoge zu konfigurieren. Diese Eigenschaften umfassen unter anderem die Wertebereiche und die Berechnungsvorschriften. Wie in der Anforderungsanalyse festgehalten, muss diese Konfiguration mit Wizards erfolgen.

Eclipse-Plug-ins bieten dazu die Möglichkeit, an fertig vordefinierten *Extension Points* individuelle Erweiterungen zu registrieren, die dann zum Beispiel einen Editor starten.

Jede Erweiterung besteht aus einer Aktionsklasse, die das Interface `IActionDelegate` implementiert. Die Methode `run` dieser Klasse ist für die Ausführung der konkreten Erweiterung, hier das Wizards, zuständig. Die Aktionsklasse verwendet eine Unterklasse der abstrakten Klassen `Wizard`, die den konkreten Wizard implementiert und dessen Seiten in Form von Unterklassen von `WizardPage` anlegt.

Die Methode `performFinish()` der Wizardklasse wird am Ende der Datenerfassung aufgerufen und kann die konfigurierten Informationen über den entsprechenden `EditPart`-Controller im Model setzen.

Die allgemeine Struktur der Wizard-Erweiterungen ist in Abbildung 5.7 wiedergegeben.

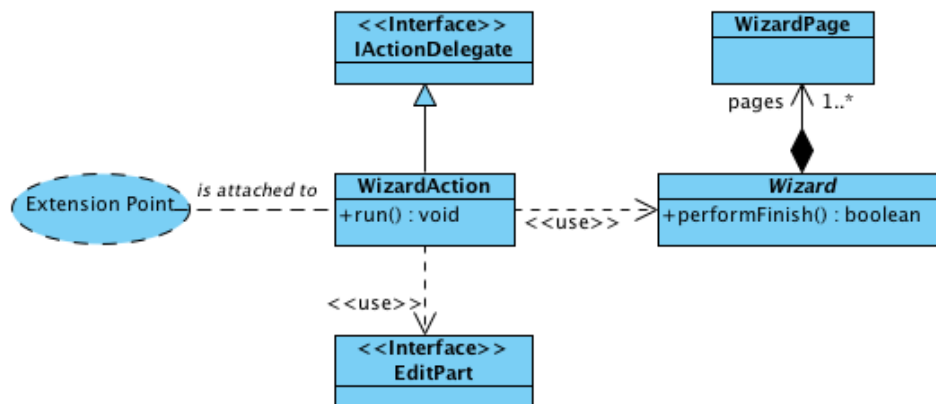


Abbildung 5.7: Struktur der Wizard-Erweiterungen

5.3 Architektur des Auswertungswerkzeuges

Der folgende Abschnitt beschreibt die Architektur der Auswertungskomponente, mit deren Hilfe auf Basis eines gegebenen Qualitätsmodells eines oder mehrere Projekte bezüglich ihrer Qualität vermessen und bewertet werden können.

5.3.1 Einleitung

Die Idee hinter dem Auswertungswerkzeug besteht darin, eine Anwendung zu entwickeln, mit deren Hilfe die Qualität eines oder mehrerer (Software-) Projekte(s) ermittelt und bewertet werden kann. Die Struktur der Anforderungen an die Qualität wird durch ein mittels des Qualitätsmodell-Editors erstelltes Qualitätsmodell festgelegt.

Grobarchitektonisch betrachtet wird das Auswertungswerkzeug zerlegt in zwei Hauptkomponenten:

1. Die **Logikkomponente** implementiert die notwendigen Algorithmen und Verfahren, um ein Qualitätsmodell auf Projekte anzuwenden. Es ist losgelöst von einem grafischen Benutzerinterface und gibt lediglich entsprechende Datenstrukturen an die aufrufenden Systeme zurück.

In Analogie zu den Bestandteilen des Qualitätsmodells (siehe 5.2.4.1) implementiert diese Komponente sowohl die Werkzeuge zur Umsetzung von Messvorschriften als auch die Algorithmen, die notwendig sind, um Berechnungsvorschriften, die im Qualitätsmodell mit Qualitätseigenschaften verknüpft sind, umzusetzen.

2. Die **Frontend-Komponente** stellt eine grafische Benutzerschnittstelle in Form eines Plug-In für Eclipse zur Verfügung, mit dem eine Auswertung konfiguriert und durchgeführt werden kann. Darüber hinaus ist es in dieser Komponente möglich, die ermittelten Informationen nach Durchführung einer Auswertung auf verschiedene Art und Weise darzustellen und weiterzuverarbeiten.

Der Vorteil der Modularisierung in zwei Teilkomponenten liegt darin, dass es später möglich ist, weitere Benutzerschnittstellen beispielsweise in Form von Webanwendungen zu erstellen, die auf den gleichen Qualitätsmodellen und den gleichen Auswertungsverfahren arbeiten wie das Frontend auf Eclipse-Basis.

Zur späteren Weiterverarbeitung und Archivierung von Auswertungen unterstützt das Auswertungswerkzeug das Erfassen und Speichern aller mit einer Auswertung verknüpften Informationen in Instanzen eines auf Basis von EMF [EMF] erstellten Datenmodells. Wie bei der Implementierung des Qualitätsmodell-Editors findet hier daher ebenfalls das Model-Driven Software Development (MDD) Anwendung.

5.3.2 Das Datenmodell für Auswertungen

In diesem Abschnitt wird das Datenmodell, in dem Informationen über eine Auswertung erfasst werden, beschrieben.

5.3.2.1 Idee

Das im Auswertungswerkzeug verwendete Datenmodell muss in der Lage sein, alle Informationen, die im Rahmen der Auswertung anfallen, zu erfassen und für zeitlich spätere Wiederverwendung oder Weiterverarbeitung zu speichern.

Die anfallenden Informationen umfassen zum einen **Datenreihen** (siehe 5.3.2.4), in denen die für Qualitätsindikatoren gemessenen bzw. für Qualitätseigenschaften berechneten Werte für jedes Projekt, das im Rahmen der Auswertung berücksichtigt worden ist, vorgehalten werden. Zum anderen umfassen sie die **Konfigurationsparameter** (siehe 5.3.2.3), mit denen festgelegt wird, auf welche Projekte und Zeiträume sich eine Auswertung erstreckt.

Das Datenmodell muss dergestalt sein, dass die Werte einer Auswertung unmittelbar nach dem Laden einer gespeicherten Modell-Instanz des Modells ohne weitere Vorbereitungen verfügbar sind. Außerdem muss es möglich sein, die Auswertung direkt nach dem Laden erneut durchzuführen, ohne weitergehende Konfigurationen durchzuführen.

Die technische Implementierung des Datenmodells erfolgt dabei unter Verwendung des EMF-Rahmenwerks, das Bestandteil der Eclipse-Plattform ist: Aus einem strukturierten Modell, das eine Instanz des Ecore-Metamodells darstellt, wird gemäß des Paradigma des Model-Driven Software Development automatisiert Code erstellt, der die Arbeit auf und mit Instanzen des Datenmodells erlaubt.

Der Vorteil der Verwendung von EMF ist darin zu sehen, dass mit wenig Aufwand eine voll funktionsfähige Datenmodell-Komponente inklusive der Möglichkeit, die Daten zu persistieren und zu einem späteren Zeitpunkt wieder auszulesen, auf Basis einer intuitiven grafischen Darstellung erzeugt werden kann. Durch die Verwendung einer grafischen Darstellung ist es darüber hinaus im Laufe der Entwicklung möglich, das Modell um neue Anforderungen zu erweitern oder existierende Features an neue Gegebenheiten anzu-

passen, ohne große Anpassungen im Programmcode vorzunehmen.

5.3.2.2 Generelle Struktur des Auswertungsmodells

Der folgende Abschnitt beschreibt die generelle Struktur des Datenmodells, das einer Auswertung zugrunde liegt. Diese Struktur ist ebenfalls in Abbildung 5.8 wiedergegeben.

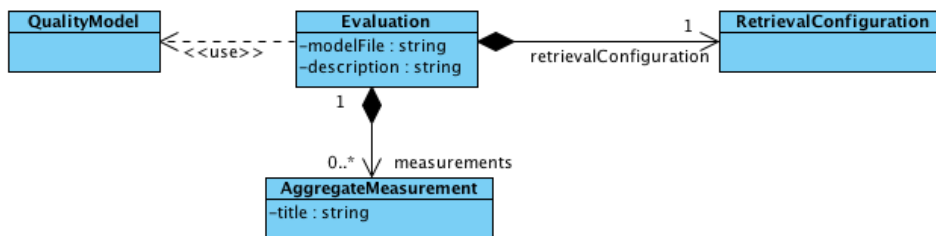


Abbildung 5.8: Generelle Struktur des Auswertungsmodells

Der Einstiegspunkt für jede Instanz des Datenmodells ist die Klasse `Evaluation`. Sie kapselt die Information, welches Qualitätsmodell im Rahmen der entsprechenden Auswertung verwendet wird, in der Eigenschaft `modelFile`. Diese Eigenschaft enthält den Dateinamen derjenigen Datei, die mit dem Qualitätsmodell-Editor angelegt wurde und das die Repräsentation des Qualitätsmodells enthält. Aus zwei Gründen ist es hier nicht praktikabel, eine direkte Objekt-Repräsentation der entsprechenden `QualityModel`-Instanz im Modell zu speichern:

1. Erstens handelt es sich bei dem Metamodell, auf dem Qualitätsmodelle basieren, um ein anderes als jenes, auf dem die Datenmodelle der Auswertung basieren. Beide Modelle in einem gemeinsamen Modell zu vereinen ist weder sinnvoll noch praktikabel, da es im Sinne der Trennung von Zuständigkeiten möglich sein muss, Qualitätsmodelle unabhängig einer etwaigen Auswertung sowie in mehreren Auswertungen verwenden zu können.
2. Der zweite Grund ist darin zu sehen, dass es bei der direkten Referenzierung einer `QualityModel`-Instanz zu Konsistenzproblemen käme, wenn das Qualitätsmodell nachträglich verändert würde. Wird, wie es im vorliegenden Fall geschieht, das Modell jedoch vor jeder Aus-

wertung aus der entsprechenden Datei geladen, ist stets sichergestellt, dass die aktuelle Fassung verwendet wird.

Dies bedeutet für bereits durchgeführte Auswertungen, dass diese bei einer Änderung am assoziierten Qualitätsmodell bis zu einer erneuten Durchführung der Auswertung weiterhin Daten entsprechend des vorherigen Standes des Qualitätsmodells vorhalten. Erst nach einer erneuten Durchführung beziehen sich die Ergebnisse auf den neueren Stand.

In der Eigenschaft `description` der Klasse `Evaluation` kann eine Beschreibung der Auswertung hinterlegt werden.

Um in einer Auswertung festzulegen, auf welchen Projekten, über welchem Zeitraum und mit welchen Datenquellen gemessen wird, verfügt die Klasse `Evaluation` über eine Assoziation zur Klasse `RetrievalConfiguration`. Die Architektur dieser *Konfigurations-Klasse* wird im Detail in Abschnitt 5.3.2.3 beschrieben.

Für jedes Projekt, das von der Auswertung berücksichtigt wird, existiert mit einer Instanz der Klasse `AggregateMeasurement` eine *Datengruppe*. Sie vereint, wie in Abschnitt 5.3.2.4 genauer erläutert wird, alle für Qualitätsindikatoren gemessenen bzw. für Qualitätseigenschaften berechneten Datenreihen des betreffenden Projektes.

In der Eigenschaft `title` dieser Klasse kann eine Beschriftung (z.B. der Name des repräsentierten Projektes) für die betreffenden Instanz abgelegt werden.

Durch die Assoziation mehrerer Datengruppen mit einer Auswertung ist eine flexible gleichzeitige Auswertung beliebig vieler Projekte möglich.

5.3.2.3 Konfiguration einer Auswertung

Dieser Abschnitt beschreibt die Struktur des Teiles des Datenmodells, in dem die Konfiguration für eine Auswertung erfasst wird. Diese Struktur ist ebenfalls in Abbildung 5.9 wiedergegeben.

Die Aufgabe der Konfiguration einer Auswertung ist, einzuschränken, auf welchen Daten gearbeitet wird. Dazu wird erfasst, auf welche Produkte die Auswertung angewandt wird, Daten welchen Zeitraumes zu berücksichtigen

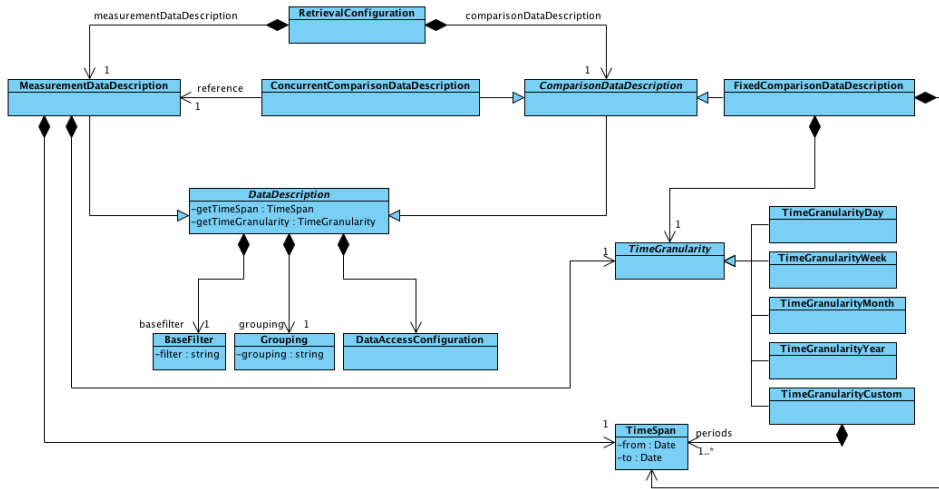


Abbildung 5.9: Konfiguration einer Auswertung

sind und wie die Datenquellen wie beispielsweise BugzillaMetrics angebunden sind. Es wird dabei unterschieden zwischen der Konfiguration für das Ermitteln der direkt weiterzuverarbeitenden, projektbezogenen Daten über eines oder mehrere Projekte sowie der Konfiguration für die Ermittlung der empirischen Vergleichsdaten.

Diese Idee schlägt sich in der Architektur der Auswertungskomponente im Datenmodell durch Verwendung der Klasse `RetrievalConfiguration` sowie der abstrakten Klasse `DataDescription` nieder. Instanzen von `RetrievalConfiguration` werden von Instanzen der Klasse `Evaluation` (siehe 5.3.2.2) referenziert und kapseln zwei Instanzen von Unterklassen von `DataDescription`:

- Eine ist ein Objekt vom Typ `MeasurementDataDescription` und beschreibt die Konfiguration für die weiterzuverarbeitenden projektbezogenen Daten.
- Bei der anderen handelt es sich um ein Objekt vom Typ `ComparisonDataDescription`. Es repräsentiert die Konfiguration für die Ermittlung der empirischen Vergleichsdaten.

Sowohl für die Konfiguration der weiterzuverarbeitenden projektbezogenen Daten als auch für die der empirischen Vergleichsdaten muss festgelegt wer-

den, welche Produkte bzw. Projekte im Rahmen der Auswertung zu berücksichtigen sind.

Dies erfolgt über Instanzen der Klassen `BaseFilter` und `Grouping`. Sie orientieren sich an der durch `BugzillaMetrics` implizierten Struktur einer Metrikdefinition: Mithilfe der `BaseFilter`-Klasse wird festgelegt, welche Produkte, Versionen oder Komponenten in der Auswertung berücksichtigt werden. Die `Grouping`-Klasse definiert, wie die Menge von Daten gruppiert wird. Damit ist es beispielsweise möglich, bei der Auswertung über mehrere Produkte nach dem jeweiligen Produkt zu gruppieren. Somit wirkt sich diese Konfigurationsoption direkt auf die Anzahl der Datengruppen, die in der Auswertung enthalten sind, aus.

Beide Klassen enthalten in der aktuellen Ausbaustufe jeweils nur die XML-Darstellung des `basefilter`- bzw. `groupingParameters`-Elementes, das in `BugzillaMetrics` verwendet wird, da derzeit nur `BugzillaMetrics` als Messwerkzeug unterstützt wird. Trotzdem ist es später möglich, die Konfiguration so zu verallgemeinern, dass auch andere Messwerkzeuge unterstützt werden. Dafür sind lediglich Anpassungen an `BaseFilter` bzw. `Grouping` notwendig.

Darüber hinaus verfügt die Klasse `DataDescription` über eine Assoziation zur abstrakten Klasse `DataAccessConfiguration`. Mit deren Hilfe werden die Zugriffsparameter auf die der Auswertung zugrunde liegenden Datenquellen wie `BugzillaMetrics` festgelegt. Die Detailstruktur dieser Teilkomponente wird später im Abschnitt *Konfiguration der Datenquellen* genauer beschrieben.

In `DataDescription` werden zwei abstrakte Methoden `getTimeSpan` und `getTimeGranularity` deklariert, die den Zeitraum, über den sich die Auswertung erstreckt, und die Granularität, nach der Zeitabschnitte gebildet werden, zurückgeben. Die konkrete Implementierung bleibt Unterklassen vorbehalten, da je nach Situation diese Informationen aus verschiedenen Quellen stammen. Diese Idee wird in den folgenden beiden Abschnitten genauer erläutert:

Konfiguration der projektbezogenen Daten

Der Teil der Konfiguration, in dem festgelegt wird, auf welchen Projekten und innerhalb welcher Zeiträume die Auswertung zu erfolgen hat, wird im Datenmodell in der Klasse `MeasurementDataDescription` modelliert.

Der Zeitraum, über den sich die Auswertung erstreckt, wird in der Instanz von `TimeSpan` erfasst, die mit der jeweiligen `MeasurementDataDescription`-Instanz assoziiert ist. Die Klasse `TimeSpan` repräsentiert einen Zeitraum über die beiden Attribute `from` und `to`, die den Zeitstempel des Beginns bzw. Endes des Zeitraums enthalten.

Die Granularität, nach der Messwerte innerhalb des konfigurierten Zeitraumes gruppiert werden, wird über die Assoziation zur abstrakten Klasse `TimeGranularity` bzw. deren konkreter Unterklassen definiert. Es werden fünf Arten von Granularitäten unterstützt:

- Eine Granularität auf Tagesebene wird spezifiziert durch die Verwendung einer Instanz von `TimeGranularityDay`.
- Eine Granularität auf Wochenebene wird spezifiziert durch die Verwendung einer Instanz von `TimeGranularityWeek`.
- Eine Granularität auf Monatsebene wird spezifiziert durch die Verwendung einer Instanz von `TimeGranularityMonth`.
- Eine Granularität auf Jahresebene wird spezifiziert durch die Verwendung einer Instanz von `TimeGranularityYear`.
- Darüber hinaus unterstützt das Auswertungswerkzeug auch die Verwendung einer individuellen Granularität, wie sie in `BugzillaMetrics` in Version 0.9.3 eingeführt wurden. Damit ist es möglich, beispielsweise nach Release-Zyklen oder Geschäftsjahren, die vom regulären Kalenderjahr abweichen, zu gruppieren.

Hierzu wird eine Instanz der Klasse `TimeGranularityCustom` verwendet. Diese Klasse kapselt eine Menge von `TimeSpan`-Instanzen, welche die Zeiträume definieren, nach denen die Gruppierung jeweils zu erfolgen hat.

Konfiguration der empirischen Vergleichsdaten

Über welche Projekte und Zeiträume sich die Erfassung der empirischen Vergleichsdaten zu erstrecken hat, wird in Analogie zur Situation bei den projektbezogenen Daten über die abstrakte Klasse `ComparisonDataDescription` modelliert. Sie erbt alle Eigenschaften der Basisklasse `DataDescription`, mit denen der Projektfilter sowie die Gruppierungsparameter festgelegt werden. Die Klasse verfügt über zwei konkrete Ausprägungen:

Die Klasse `FixedComparisonDataDescription` setzt die gleiche Idee um wie die Klasse `MeasurementDataDescription`: Der Zeitraum, auf dem zu messen ist, wird über eine Assoziation zu `TimeSpan` festgelegt, während die Granularität über eine Assoziation zu einer konkreten Ausprägung von `TimeGranularity` modelliert wird. Anschaulich bedeutet die Verwendung von `FixedComparisonDataDescription`, dass die Erfassung der Vergleichsdaten stets auf einem konstanten und festen Zeitraum erfolgt, der nicht von der Konfiguration der Erfassung der projektbezogenen Daten abhängt.

Bei der zweiten Ausprägung handelt es sich um die Klasse `ConcurrentComparisonDataDescription`. Instanzen von ihr werden in der Konfiguration verwendet, wenn der Zeitraum und die Granularität für die Erfassung empirischer Vergleichsdaten stets dem Zeitraum und der Granularität für die Erfassung der projektbezogenen Daten entsprechen. Hierzu verfügt diese Klasse über eine Assoziation zu `MeasurementDataDescription`, die als Referenz für die Ermittlung von Zeitraum und Granularität verwendet wird: Die Methoden `getTimeSpan` und `getTimeGranularity` von `ConcurrentComparisonDataDescription` rufen dementsprechend die Methoden gleichen Namens der `MeasurementDataDescription`-Instanz auf und geben deren Werte zurück. Somit ist sichergestellt, dass die Erfassung der empirischen Vergleichsdaten sich stets auf die gleichen Zeiträume erstreckt wie die Erfassung der projektbezogenen Daten.

Der Vorteil der Aufteilung der Messkonfiguration in die Teilkonfiguration für die projektbezogenen Daten und die Teilkonfiguration für die empirischen Vergleichsdaten ist, dass es damit möglich ist, beide Datenmengen vollkommen unabhängig voneinander und flexibel zu erfassen. Außerdem wird durch die Verwendung der vorgestellten Klassenstruktur ein übersichtliches Prinzip zur Konfiguration etabliert, das einer Lösung, bei der alle möglichen Konfigurationsparameter als atomare Eigenschaften der Klasse `RetrievalConfiguration` vorliegen, hinsichtlich Wartbarkeit, Übersichtlichkeit und Strukturiertheit überlegen ist.

Konfiguration der Datenquellen

Neben den bereits genannten Parametern zur Konfiguration einer Auswertung ist es darüber hinaus notwendig, die Zugangsparameter für die zu verwendenden Messwerkzeuge, die im Rahmen der Messung als Datenquelle verwendet werden sollen, zu erfassen. Zur Zeit beschränkt sich die Verfügbarkeit

von Messwerkzeugen auf BugzillaMetrics. Daher muss in der aktuellen Ausbaustufe nur Augenmerk auf dessen Konfiguration gelegt werden. Nichtsdestotrotz muss eine Möglichkeit für die zukünftige Erweiterbarkeit um weitere Werkzeuge vorgesehen werden.

Im Rahmen der Anforderungsanalyse wurde festgestellt, dass die BugzillaMetrics-Messvorschriften aller Qualitätsindikatoren stets auf der gleichen Bugzilla-Datenbank arbeiten. Insofern ist es nicht notwendig, die Zugangsdaten pro Qualitätsindikator zu erfassen, sondern es reicht aus, diese einmal pro Auswertung festzulegen. Es ist allerdings vonnöten, für die Ermittlung der empirischen Vergleichsdaten eine separate Datenbankanbindung definieren zu können.

Diesen Anforderungen wird durch die Assoziation der `DataDescription`-Klasse mit der Klasse `DataAccessConfiguration` begegnet. Konkrete Unterklassen dieser Klasse repräsentieren jeweils die Konfiguration für ein Messwerkzeug. Diese Idee ist ebenfalls in Abbildung 5.10 wiedergegeben.

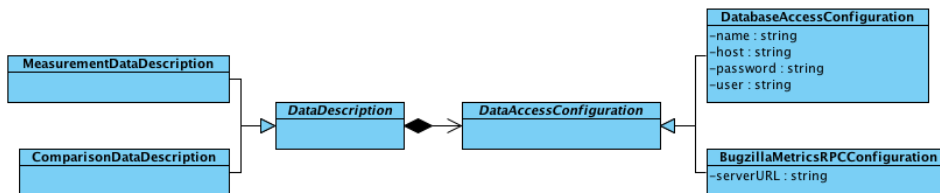


Abbildung 5.10: Konfiguration der Datenquellen

Dadurch ist es möglich, sowohl mit der Konfiguration für die Ermittlung der projektbezogenen Daten (`MeasurementDataDescription`) wie auch mit der Konfiguration für die Ermittlung der empirischen Vergleichsdaten (`ComparisonDataDescription`) eine eigene Messwerkzeugkonfiguration zu verknüpfen.

Von der abstrakten Klasse `DataAccessConfiguration` existieren derzeit zwei Unterklassen:

1. `DatabaseAccessConfiguration` ist eine Klasse, die eine generische Datenbankkonfiguration kapselt. Sie kann für alle die Messwerkzeuge verwendet werden, die als Konfigurationsparameter lediglich Zugriffs-

daten für eine Datenbank benötigen. Zu den gekapselten Parametern zählen neben dem Hostnamen des Datenbankservers auch der Name der Datenbank, der Benutzername für den Zugriff sowie das entsprechende Passwort.

Diese Klasse findet aktuell keine Verwendung im Auswertungswerkzeug, da sie während der Entwicklungsphase nur solange verwendet wurde, bis die XML-RPC-Anbindung für BugzillaMetrics (siehe Abschnitt 5.4) fertiggestellt war. Sie verbleibt allerdings in der Anwendung, um eine spätere Erweiterung um neue Messwerkzeuge zu vereinfachen.

2. Die Klasse `BugzillaMetricsRPCConfiguration` repräsentiert die Konfiguration des BugzillaMetrics-Messwerkzeuges. Die Kommunikation für dieses Werkzeug erfolgt ausschließlich über eine XML-RPC-Schnittstelle (siehe Abschnitt 5.4). Daher ist es in Instanzen dieser Klasse nur notwendig, die URL des entsprechenden Webservices zu erfassen.

Der Grund für dieses Architekturmuster ist erneut in der Erweiterbarkeit des Systems zu sehen. Durch die Verwendung weiterer Unterklassen von `DataAccessConfiguration` ist es möglich, zu einem späteren Zeitpunkt andere Messwerkzeuge zu verwenden. Darüber hinaus steht mit `DatabaseAccessConfiguration` eine generische Möglichkeit zur Konfiguration von Messwerkzeugen mit Datenbankbindung zur Verfügung, da hier ein im Datenbankumfeld sehr gebräuchliches und allgemein gehaltenes Schema für die Abbildung von Verbindungsdaten Verwendung findet.

5.3.2.4 Datenreihen

Jede Datengruppe, die im Modell durch eine Instanz von `AggregateMeasurement` repräsentiert wird (siehe 5.3.2.2), enthält für jeden Qualitätsindikator und jede Qualitätseigenschaft, die im der Auswertung zugrunde liegenden Qualitätsmodell enthalten sind, eine Datenreihe mit Werten für den betreffenden Qualitätsknoten. Die Anzahl der Werte in einer Datenreihe ergibt sich aus dem im Rahmen der Auswertungskonfiguration (siehe 5.3.2.3) konfigurierten Zeitraum und der gewählten Granularität. Darüber hinaus ist eine Datenreihe optional mit einer Menge von Werten assoziiert, die die empirischen Vergleichsdaten für die Reihe darstellen.

Die Architektur der Teilkomponente des Modells, in der diese Idee umgesetzt wird, wird in diesem Abschnitt erläutert. Sie findet sich ebenfalls in

Abbildung 5.11 wieder.

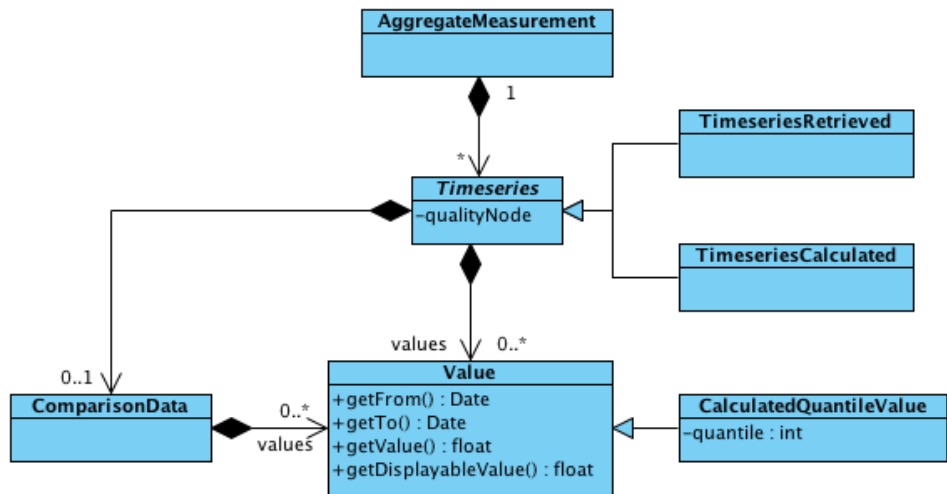


Abbildung 5.11: Datenreihen und Werte

Datenreihen werden im Modell durch die abstrakte Klasse `Timeseries` und deren Unterklassen `TimeseriesRetrieved` und `TimeseriesCalculated` repräsentiert. Jede Instanz von `Timeseries` ist assoziiert mit einem Qualitätsknoten des verwendeten Qualitätsmodells. Diese Beziehung wird durch das Attribut `qualityNode` realisiert. Die Datenreihe kapselt dann jeweils die Werte, die für den betreffenden Qualitätsindikator, eingeschränkt durch die Auswertungskonfiguration, gemessen bzw. für die betreffende Qualitätseigenschaft berechnet wurden. Für *gemessene Datenreihen* werden Instanzen von `TimeseriesRetrieved` verwendet, während für *berechnete Datenreihen* Instanzen der Klasse `TimeseriesCalculated` Verwendung finden. Diese Unterscheidung in zwei Arten von Datenreihen sorgt für eine erhöhte Flexibilität und macht zukünftige Wartungen und Umstrukturierungen einfacher, wenn es darum geht, konkrete Unterschiede im Verhalten berechneter und gemessener Datenreihen zu implementieren.

Werte einer Datenreihe

Die einzelnen Werte, die in einer Datenreihe gekapselt sind, werden durch die Klasse `Value` und deren Unterklassen repräsentiert.

Grundsätzlich arbeitet das Auswertungswerkzeug mit Fließkommazahlen als Basisdatentyp. Es reicht jedoch nicht aus, ausschließlich eine Menge von Fließkommazahlen als Werte der Datenreihen zu verwenden, da mit jedem Wert neben dem rein numerischen Wert weitere Informationen verknüpft sind:

- Jeder Wert einer Datenreihe ist einem Zeitraum, auf den er sich bezieht, oder einem Zeitpunkt, zu dem er angefallen ist, zugeordnet. Bei Verwendung von `BugzillaMetrics`, das zeitraumorientiert arbeitet, als Messwerkzeug ist jedem Wert daher ein Zeitraum zugeordnet, der durch die gewählte Granularität festgelegt wird. Bei Verwendung snapshot-orientierter, d.h. zeitpunktbezogener, Messwerkzeuge ist mit jedem Wert der Zeitpunkt verknüpft, zu dem der *Snapshot* erstellt wurde.
- Die Werte der Datenreihe für einen Qualitätsknoten n haben zwei Aufgaben: Zum einen dienen sie in der Frontend-Komponente als Quelle für die darzustellenden Daten. Zum anderen verwendet die Logik-Komponente sie als Eingabe für die Berechnung der Datenreihen der Qualitätsknoten, die eine Assoziation zum Knoten n haben. (Siehe 5.3.3)

Durch diese Konstellation kann die Situation auftreten, dass die Darstellung eines Wertes in der Frontend-Komponente abweicht von der Darstellung des Wertes als Eingabe in einem Berechnungsschritt in der Logik-Komponente. Ein konkretes Beispiel dafür findet sich weiter unten in der Erklärung zur Klasse `CalculatedQuantileValue`.

Die soeben skizzierten Anforderungen an Werte einer Datenreihe werden in der Architektur durch Verwendung der Klasse `Value`, die neben dem rein numerischen Wert weitere Parameter kapselt, umgesetzt. Der Zeitraum, auf den sich der Wert bezieht, wird durch die Parameter `from` und `to` festgelegt. Bei zeitpunktorientierten Messungen enthalten sowohl `from` als auch `to` den gleichen Wert, nämlich den Zeitpunkt.

Die Unterscheidung in die Darstellung des Wertes in der Frontend-Komponente und in der Logik-Komponente erfolgt über die Methoden `getValue()` und `getDisplayableValue()`: Die Methode `getValue()` gibt den Wert zurück, der als Eingabe für weitere Berechnungsschritte verwendet wird, während `getDisplayableValue()` die Darstellung des Wertes, die in der Frontend-Komponente verwendet wird, als Rückgabewert hat. Aus Gründen der

Effizienz geben beide Methoden in der Implementierung in der Standardklasse `Value` den Wert des gleichen Datenfeldes zurück. Im Folgenden wird allerdings die Klasse `CalculatedQuantileValue` vorgestellt, bei der von diesem Prinzip abgewichen wird und die der eigentliche Anlass für die Einführung dieses Konzeptes war.

Für Werte einer Datenreihe, die auf Basis der Berechnungsvorschrift `ValueSpecificationLocalQuantile` (Quantil-Berechnungen, siehe 5.2.4.3), berechnet worden sind, werden in der Frontend-Komponente die Quantil-Werte angezeigt, während es bei Verwendung der Werte als Eingabe in einem Berechnungsschritt ggf. notwendig ist, auf den absoluten Wert zuzugreifen. Aus diesem Grund werden bei dieser Art von Werten nicht Instanzen der Klasse `Value` sondern der Unterklasse `CalculatedQuantileValue` verwendet.

Diese unterscheidet sich insofern von ihrer Oberklasse, als dass sie zwei Datenfelder für Werte enthält: Die Eigenschaft `quantile` repräsentiert den Quantil-Wert und kann über die Methode `getDisplayableValue()` abgefragt werden. Die Eigenschaft `value` enthält, in Analogie zur Situation in der Klasse `Value`, den absoluten Wert, dient als Eingabe für weitere Berechnungsschritte und kann über `getValue()` abgefragt werden.

Empirische Vergleichsdaten einer Datenreihe

Zum Zwecke der Ermittlung empirischer Vergleichsdaten im Rahmen einer Berechnung ist jede Datenreihe optional mit einer Menge von Werten verknüpft, die eine empirische Datenbasis darstellen. Zu dieser Datenbasis kann jeder Wert der Datenreihe ins Verhältnis gesetzt werden, um dessen Quantil-Einordnung zu ermitteln.

Eine empirische Datenbasis wird im Datenmodell repräsentiert durch die Klasse `ComparisonData`, deren Instanzen von `Timeseries`-Instanzen referenziert werden. Jede Instanz von `ComparisonData` kapselt eine Menge von Werten in Form von `Value`-Instanzen.

Der Vorteil der Kompositionsbeziehungen zwischen `AggregateMeasurement` und `Timeseries` sowie dieser Klasse und `Value` ist größtmögliche Flexibilität bei der Erfassung von Datenreihen und Messwerten: Die Anzahl der Werte pro Datenreihe ist im Vorfeld der Auswertung nicht bekannt, weshalb die Verwendung einer Datenstruktur fester Größe nicht möglich ist. Durch die

verwendete Listenstruktur ist es stattdessen möglich, im Laufe der Auswertung beliebig viele Werte zu einer Datenreihe hinzuzufügen. Gleiches gilt für die Anzahl der Datenreihen. Diese ergibt sich aus der Anzahl der Qualitätsknoten im zugrunde liegende Qualitätsmodell, die ebenfalls im Vorfeld nicht bekannt ist.

5.3.3 Messen und Berechnen

In diesem Abschnitt wird die Architektur der Teilkomponente beschrieben, die im Auswertungswerkzeug die Aufgabe übernimmt, die Kernfunktionen einer Auswertung durchzuführen. Der Ablauf einer Auswertung ist im Kern in drei Schritte untergliedert:

1. **Vorbereiten der Auswertung:** Anlegen einer Konfiguration
2. **Qualitätsindikatoren vermessen**
3. **Qualitätseigenschaften berechnen**

Der erste Schritt erfolgt durch das Erzeugen einer Instanz von `RetrievalConfiguration` (siehe Abschnitt 5.3.2.3). Das konkrete Vorgehen hierbei ist dem Frontend überlassen, das die vorgestellte Logik-Komponente verwendet.

Die Schritte zwei und drei werden durch die hier vorgestellte Teilkomponente realisiert. Sie arbeiten auf Grundlage der Konfiguration aus Schritt eins und dem in der Auswertung Anwendung findenden Qualitätsmodell; dieses legt fest, welche Qualitätsindikatoren vermessen und welche Qualitätseigenschaften berechnet werden müssen, welche Messwerkzeuge zum Einsatz kommen und welche Berechnungsvorschriften anzuwenden sind.

Die Grobstruktur der Teilkomponente ist ebenfalls in Abbildung 5.12 wiedergegeben.

Der zentrale Einstiegspunkt für jede Auswertung ist die Klasse `Evaluator`. Diese stellt eine Methode `evaluate()` zur Verfügung, die auf Basis eines Qualitätsmodells, gegeben als Instanz von `QualityModel`, und einer Auswertungskonfiguration in Form eines Objektes vom Typ `RetrievalConfiguration` eine Auswertung durchführt. Die Rückgabe dieser Methode und damit das Ergebnis der Auswertung ist eine Instanz der Klasse `Evaluation` (siehe Abschnitt 5.3.2.2).

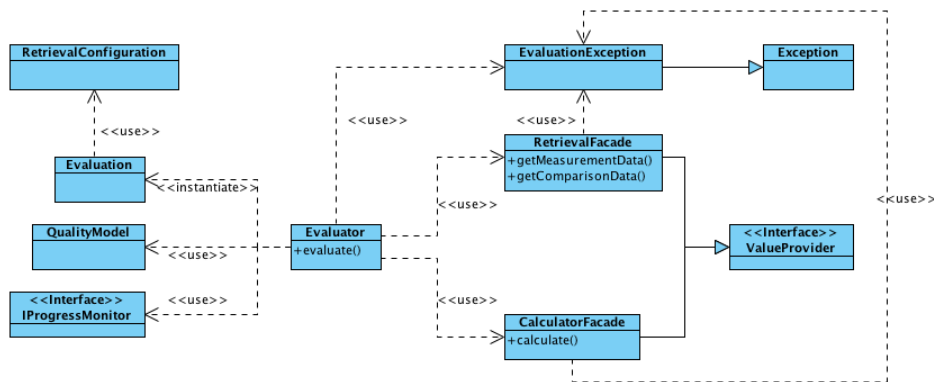


Abbildung 5.12: Grobstruktur der Komponente zum Vermessen und Berechnen einer Auswertung

Die eigentlichen Aufgaben des Messens bzw. Berechnens der Werte für einen Qualitätsknoten delegiert `Evaluator` an die beiden Klassen `RetrievalFacade` und `CalculatorFacade`. Beide implementieren das Entwurfsmuster Fassade [GHJV95] und stellen somit vereinfachte Schnittstellen zu den komplexeren Subsystemen, die die Messwerkzeuge bzw. die Berechnungsvorschriften realisieren, zur Verfügung. Die genaue Funktionsweise dieser beiden Fassaden sowie der von ihnen verwalteten Subsysteme wird in Abschnitt 5.3.4 bzw. Abschnitt 5.3.5 beschrieben.

Etwaige Fehler, die bei der Durchführung der Auswertung auftreten, werden in Form von Ausnahmen des Typs `EvaluationException` an die aufrufende Komponente (i.d.R. eine Frontend-Komponente, jedoch beispielsweise auch Test-Rahmenwerke wie JUnit [JUn]) zurückgegeben. Über die benutzt-Beziehung zum Interface `IProgressMonitor`¹, das Bestandteil des Eclipse Plug-In-API ist, können sich aufrufende Komponenten periodisch vom Fortschritt der Durchführung der Auswertung unterrichten lassen. Dies ermöglicht es zum Beispiel grafischen Benutzeroberflächen, einen Fortschrittsbalken einzublenden und regelmäßig zu aktualisieren.

¹<http://download.eclipse.org/eclipse/downloads/documentation/2.0/html/plugins/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/IProgressMonitor.html>

5.3.4 Messwerkzeuge

Die Messwerkzeuge der Auswertungskomponente realisieren die Anbindung externer Werkzeuge wie BugzillaMetrics, mit deren Hilfe für die Qualitätsindikatoren des in der Auswertung verwendeten Qualitätsmodells Werte gemessen werden. Die Struktur dieser Komponente ist in Abbildung 5.13 dargestellt.

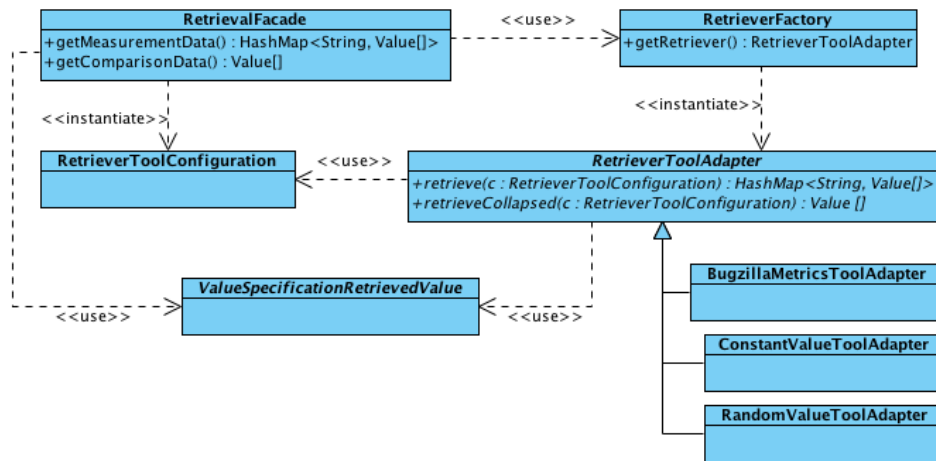


Abbildung 5.13: Architektur der Messwerkzeuge

Die Teilkomponente, in der die Messwerkzeuge realisiert sind, ist als eigenständiges Subsystem innerhalb des Auswertungswerkzeuges gekapselt und wird im Normalfall nur über die Fassadenimplementierung in `RetrievalFacade` angesprochen.

Das Fassade-Entwurfsmuster fördert dabei das Entwurfsprinzip der *losen Kopplung* [LL07, S. 385f], weil die Schnittstellen der einzelnen Messwerkzeuge zu einer einzigen zusammengefasst werden. Darüber hinaus wird die *Komplexität reduziert*, da das Subsystem hinter der Fassade versteckt wird und die sie aufrufenden Komponenten nur gegen die Schnittstelle von `RetrievalFacade` programmiert werden müssen.

Welches Messwerkzeug bei der Vermessung eines Qualitätsindikators Verwendung findet, hängt von der Messvorschrift ab, die in Form einer Instanz von `ValueSpecificationRetrievedValue` mit dem entsprechenden Indika-

tor verknüpft ist. Diese Instanz wird an `RetrievalFacade` übergeben. Zum Erzeugen des passenden Messwerkzeuges dient die Klasse `RetrieverFactory`, die das Fabrik-Entwurfsmuster nach [GHJV95] implementiert. Die Methode `getRetriever()` dieser Fabrik-Klasse gibt, in Abhängigkeit der übergebenen Messvorschrift, ein Objekt vom Typ `RetrieverToolAdapter` zurück, bei dem es sich um die konkrete Realisierung eines Messwerkzeugs handelt.

Der Vorteil der Verwendung des Fabrik-Entwurfsmusters ist, dass der Aufrufer in Form von `RetrievalFacade` entkoppelt ist von den Implementierungen konkreter Messwerkzeug-Klassen. Dies ist besonders dann von Vorteil, wenn zu einem späteren Zeitpunkt neben `BugzillaMetrics` weitere Messwerkzeuge unterstützt werden — in diesem Fall können dann Instanzen des neuen Messwerkzeuges erzeugt werden, ohne dass sich die aufrufende Komponente ändern muss.

Alle Implementierungen von Messwerkzeugen basieren auf der abstrakten Klasse `RetrieverToolAdapter`. Die Klasse definiert die Methoden `retrieve()` sowie `retrieveCollapsed()`. Die erste Methode gibt das Ergebnis einer Messung in Form einer Menge Datenreihen (siehe 5.3.2.2) gruppiert nach dem in der Auswertungskonfiguration definierten Gruppierungsparameter (siehe 5.3.2.3) zurück. Diese Methode wird von der Methode `getMeasurementData()` der Klasse `RetrievalFacade` aufgerufen, um die projektbezogenen Daten für einen Qualitätsindikator zu ermitteln. Die Rückgabe von `retrieve()` impliziert die Anzahl der Datengruppen, die für den betreffenden Qualitätsindikator in der Auswertung vorliegen, da jede Gruppe von Datenreihen in der Rückgabe einer Datengruppe entspricht. Die zweite Methode `retrieveCollapsed()` dient der Ermittlung empirischer Vergleichsdaten durch `getComparisonData()` von `RetrievalFacade`. Bei der Vergleichsdatenermittlung ist keine Gruppierung nach beispielsweise dem Projekt notwendig. Daher besteht die Rückgabe dieser Methode aus lediglich einer Datenreihe.

Derzeit existieren drei konkrete Implementierungen von Messwerkzeugen in Form von Spezialisierungen von `RetrieverToolAdapter`:

- Die Klasse `RandomValueToolAdapter` implementiert ein Messwerkzeug, das stets eine Datenreihe mit genau einem **zufällig ausgewählten Wert** zurück gibt. Wenn für den verwendeten Qualitätsindikator ein Wertebereich (siehe 5.2.3) definiert worden ist, liegt der Wert innerhalb dieses Wertebereiches.

Diese Klasse diente während der Entwicklung, als die Anbindung an BugzillaMetrics noch nicht fertiggestellt war, als Datenquelle für Testzwecke. Darüber hinaus wird sie in Unit-Tests mit demselben Zweck verwendet.

- Bei der Klasse `ConstantValueToolAdapter` handelt es sich um die Implementierung eines Messwerkzeuges, das stets genau eine Datenreihe mit genau einem im Vorfeld festgelegten **konstanten Wert** zurück gibt.

Wie `RandomValueToolAdapter` auch wird dieses Messwerkzeug in Unit-Tests eingesetzt, um Berechnungsvorschriften mit stets gleicher Eingabe testen zu können.

- Die **Einbindung von BugzillaMetrics** als Messwerkzeug erfolgt über die Klasse `BugzillaMetricsToolAdapter`. Ihre Architektur und Funktionsweise wird im nächsten Abschnitt erläutert.

5.3.4.1 BugzillaMetrics als Messwerkzeug

Die Anbindung von BugzillaMetrics als Messwerkzeug in der Auswertungskomponente erfolgt über die Klasse `BugzillaMetricsToolAdapter` und einige weitere Hilfsklassen. Die Struktur dieser Anbindung ist in Abbildung 5.14 wiedergegeben.

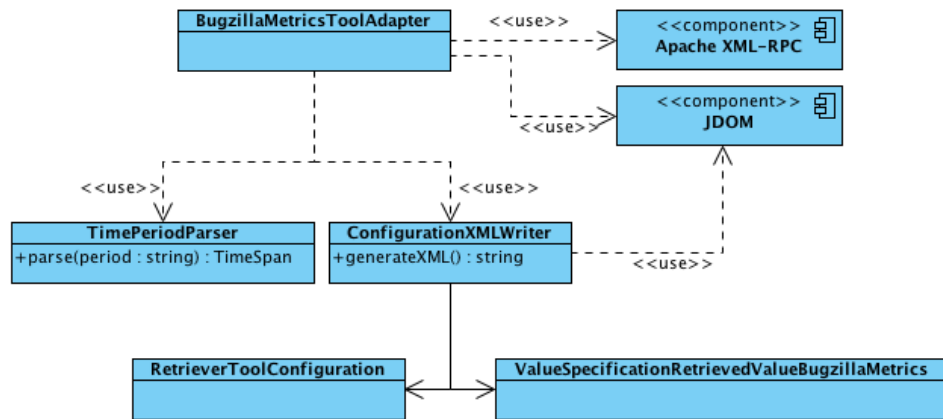


Abbildung 5.14: Detailarchitektur des Messwerkzeuges für BugzillaMetrics

Bei der Messung von Daten mithilfe von BugzillaMetrics wird die Metrik-Spezifikation als XML-Zeichenkette an den in Abschnitt 5.4 beschriebenen XML-RPC-Webservice, den BugzillaMetrics beinhaltet, übergeben. Die Rückgabe des XML-RPC-Aufrufes ist eine Zeichenkette, in der das Ergebnis der Messung in XML kodiert ist.

Die Metrik-Spezifikation wird durch die Hilfsklasse `ConfigurationXMLWriter` erzeugt. Sie setzt sich zusammen aus den Informationen aus der Messvorschrift (*Case Value Calculations*, *Group Evaluations*, *Fixed Fields*) sowie dem *Basefilter* und den *Grouping Parameters* aus der Auswertungskonfiguration. Die Container mit diesen Informationen liegen in Form von Referenzen auf Instanzen von `ValueSpecificationRetrievedValueBugzillaMetrics` bzw. `RetrieverToolConfiguration` vor. Zum Erzeugen der XML-Zeichenkette wird die JDOM-Bibliothek, die zur Arbeit mit XML in Java dient, verwendet.

Die Kommunikation mit dem BugzillaMetrics-Webservice erfolgt mithilfe des XML-RPC-Klienten `XmlRpcClient` aus der *Apache XML-RPC* Bibliothek.

Da es sich bei der Rückgabe von BugzillaMetrics um einen XML-Datenstrom handelt, muss dieser vor der Weiterverarbeitung in Datengruppen bzw. -gruppen überführt werden. Das Parsen der XML-Daten erfolgt dabei erneut mit der in der JDOM-Bibliothek enthaltenen Funktionalität. Die Hilfsklasse `TimePeriodParser` übernimmt dabei die Aufgabe, die durch BugzillaMetrics in "natürlichsprachlicher" Form wie `week 20/2008` übergebenen Zeitraumspezifikationen in eine Instanz der Klasse `TimeSpan`, mit der eine einfachere Weiterverarbeitung und Analyse möglich ist, zu überführen. Sie akzeptiert als Eingabe der statischen Methode `parse()` einen String mit einer natürlichsprachlichen Zeitraumangabe und gibt eine entsprechende Instanz von `TimeSpan` zurück.

Fehler, die in BugzillaMetrics während der Messung auftreten, werden im XML-RPC-Ergebnis ebenfalls in eine entsprechende XML-Struktur kodiert. Beim Parsen des Datenstroms erkennt `BugzillaMetricsToolAdapter` dies und reagiert ggf. entsprechend, indem eine Ausnahme vom Typ `EvaluationException` geworfen wird.

5.3.5 Algorithmen für Berechnungsvorschriften

Dieser Abschnitt beschreibt die Architektur und die Funktionsweise der Komponente des Auswertungswerkzeuges, in der die Algorithmen für Berechnungsvorschriften von Qualitätseigenschaften implementiert werden.

5.3.5.1 Idee

Die Algorithmen für die Berechnungsvorschriften, die in einem Qualitätsmodell mit Qualitätseigenschaften verknüpft werden können, werden im dritten Schritt der Auswertung (siehe Seite 72) verwendet. Sie werden damit aufgerufen, nachdem alle Qualitätsindikatoren des verwendeten Qualitätsmodells vermessen worden sind und deren Werte in Form von Datengruppen und -reihen vorliegen.

Die Datenreihen der Qualitätseigenschaften, d.h. der inneren Knoten, des Modells werden sodann ermittelt, in dem für jede Qualitätseigenschaft die passende Implementierung der Berechnungsvorschrift ermittelt wird. Für alle Datengruppen der untergeordneten Qualitätsknoten (hierbei kann es sich sowohl im Qualitätsindikatoren als auch im weitere Qualitätseigenschaften) wird diese mit den jeweiligen Datenreihen als Eingabeparameter aufgerufen und errechnet ebenfalls Datengruppen und -reihen.

Der Algorithmus zum Aufrufen der Berechnungsvorschriften und berechnen der Datenreihen arbeitet nach dem Prinzip *Depth-first, Left-to-right* auf dem Qualitätsmodell. Das bedeutet, dass die Berechnung der Qualitätseigenschaften bei einer der Eigenschaften startet, die nicht Unterknoten anderer Eigenschaften sind. Dann wird für jeden Unterknoten die Berechnung seiner Datenreihen angestoßen. Für alle Unterknoten dieses Unterknotens wird das gleiche Prinzip rekursiv angewandt, bis eine Qualitätseigenschaft erreicht ist, die nur noch bereits vermessene Qualitätsindikatoren als Unterknoten besitzt. Dadurch ist sichergestellt, dass alle Unterknoten eines Knotens über Werte, die als Eingabe dienen können, verfügt, bevor der für die Berechnungsvorschrift passende Algorithmus angewandt wird.

Bei diesem Verfahren kann die Situation eintreten, dass für eine Qualitätseigenschaft n die Berechnung der Datenreihen mehrmals angestoßen wird. Dies ist dann der Fall, wenn n als Unterknoten mehrerer anderer Qualitätseigenschaft fungiert. Für diesen Fall protokolliert die Auswertungskomponente im Laufe einer Auswertung die Knoten, deren Datenreihen bereits berech-

net wurden, und kann daher auf die erneute Berechnung der Datenreihen verzichten.

Im Sinne der *Trennung von Zuständigkeiten* und der *Reduzierung von Abhängigkeiten und Komplexität* innerhalb der Auswertungskomponente verfügen die einzelnen Algorithmen für die Berechnungsvorschriften über keinerlei Kenntnisse bezüglich des Kontextes, in dem sie eingesetzt werden. Das bedeutet, dass sie lediglich die Existenz von Argument-Werten in Form von `Value`-Instanzen (siehe Abschnitt 5.3.2.4) und ggf. empirischer Vergleichsdaten als `ComparisonData`-Instanzen voraussetzen, jedoch keine Annahmen über die Struktur des zugrunde liegenden Qualitätsmodells treffen oder Kenntnis darüber voraussetzen. Die Architektur, die diese Trennung von Zuständigkeiten unterstützt, wird in den folgenden Abschnitten beschrieben.

5.3.5.2 Grundstruktur

Die Grundstruktur der Berechnungs-Komponente wird in diesem Abschnitt beschrieben und ist in Abbildung 5.15 wiedergegeben.

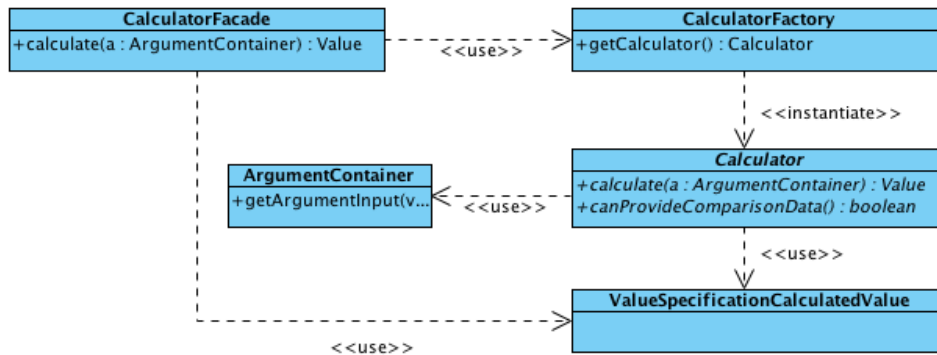


Abbildung 5.15: Grundstruktur der Berechnungs-Komponente

Die Komponente, die für die Berechnung der Werte von Qualitätseigenschaften zuständig ist, ist wie die Komponente zur Vermessung von Qualitätsindikatoren auch als geschlossenes Subsystem realisiert und wird im Normalfall nur über die Fassadenimplementierung in `CalculatorFacade` angesprochen.

Das Fassade-Entwurfsmuster fördert auch hier die *lose Kopplung* und *reduziert die Komplexität*, da das Subsystem hinter der Fassade versteckt wird.

Welche Algorithmus-Implementierung bei den Berechnungen für eine Qualitätseigenschaft verwendet wird, hängt von der Berechnungsvorschrift ab, die in Form einer Instanz von `ValueSpecificationCalculatedValue` mit der entsprechenden Eigenschaft verknüpft ist. Zum Erzeugen der passenden Algorithmus-Implementierung dient die Klasse `CalculatorFactory`, die wie `RetrievalFactory` das Fabrik-Entwurfsmuster implementiert. Die Methode `getCalculator()` dieser Fabrik-Klasse gibt, in Abhängigkeit von der übergebenen Berechnungsvorschrift, ein Objekt vom Typ `Calculator` zurück, bei dem es sich um die konkrete Implementierung einer Berechnungsvorschrift handelt.

Die Vorteile, die hier in der Verwendung des Fabrik-Entwurfs zu sehen sind, sind die gleichen wie bei der Verwendung von `RetrieverFactory`: Der Aufrufer ist entkoppelt von der Implementierung konkreter Berechnungsvorschriften und es ist einfach möglich, das Werkzeug später um weitere Algorithmen zu erweitern, ohne die aufrufenden Komponenten anzupassen.

Alle Algorithmus-Implementierungen basieren auf der abstrakten Basisklasse `Calculator`. Deren öffentliche Schnittstelle besteht aus zwei abstrakten Methoden `calculate()` und `canProvideComparisonData()`, die beide von den jeweiligen Algorithmus-Implementierung zu implementieren sind. Die Methode `canProvideComparisonData()` hat als Rückgabe einen booleschen Wert, der angibt, ob der Algorithmus in der Lage ist, eine empirische Datenbasis für Quantileinordnungen zu berechnen. Diese Funktionalität ist relevant bei der Verwendung der Berechnungsvorschrift `ValueSpecificationCalculatedValueLocalQuantile` sowie des Filters `FilterFunctionQuantile`, auf die im Folgenden noch eingegangen wird. Die Methode `calculate()` führt die eigentliche Berechnung, die durch die Klasse repräsentiert wird, durch. Ihre Rückgabe ist ein Wert für eine Datenreihe der zugehörigen Qualitätseigenschaft in Form einer Instanz der Klasse `Value`. Ihr einziges Argument ist eine Menge von Werten der untergeordneten Qualitätsknoten in Form einer `ArgumentContainer`-Instanz. Auf das Konzept der Argumente von Berechnungsvorschriften wird im nächsten Abschnitt eingegangen.

5.3.5.3 Argumente der Berechnungen

Die Algorithmen, welche die Berechnungsvorschriften implementieren, führen die Berechnungen für eine Qualitätseigenschaft durch auf Basis der gemessenen oder ebenfalls berechneten Werte der dieser Qualitätseigenschaft untergeordneten Qualitätsknoten. An die Art und Weise, auf welche die Werte vorliegen müssen, werden die folgenden Anforderungen gestellt:

1. Je nach Art des verwendeten Qualitätsmodells ist die Anzahl der Werte, die als Eingabe dienen, variabel und hängt nur von der Anzahl der Unterknoten einer Qualitätseigenschaft ab.

Daher muss es möglich sein, eine variable Anzahl von Werte als Argumente einer Berechnungsvorschrift zu übergeben. Im Rahmen der Implementierung ist dies erwähnenswert, da Sprachen wie das im vorliegenden Fall verwendete Java [Jav] in der Regel eine variable Anzahl von Methodenparametern nicht direkt unterstützen.

2. Je nach vorliegender Berechnungsvorschrift ist es unter Umständen erforderlich, dass neben dem eigentlichen Wert eines untergeordneten Qualitätsknotens in Form einer Fließkommazahl auch andere Informationen wie die dazugehörige empirische Datenbasis Bestandteil der Eingabe für die Berechnungsvorschrift sind.

Dies ist beispielsweise erforderlich bei der Berechnungsvorschrift zur Ermittlung der Quantileinordnung.

Den beiden soeben skizzierten technisch motivierten Anforderungen wird im Auswertungswerkzeug in Form der beiden Klassen **ArgumentContainer** und **ArgumentInput** begegnet. Dieser Ansatz ist in Abbildung 5.16 wiedergegeben und wird im Folgenden beschrieben.

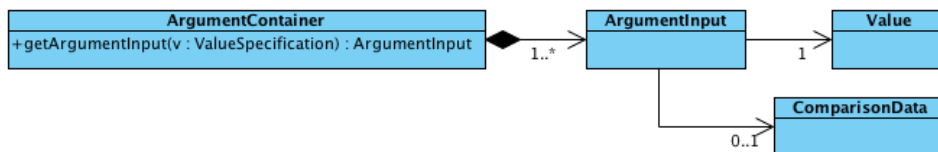


Abbildung 5.16: Argumente der Berechnungen

Die Anforderung, dass es möglich sein muss, an die Algorithmen für Berechnungsvorschriften eine variable Anzahl von Argumenten zu übergeben,

wird durch die Klasse `ArgumentContainer` umgesetzt. Dabei handelt es sich um eine *Containerklasse*, in der beliebig viele Argumente in Form von `ArgumentInput`-Instanzen enthalten sind. Die Methode `getArgumentInput()` wird durch die Algorithmus-Implementierungen verwendet, um den passenden Eingabewert für die Berechnungsvorschrift eines Unterknotens zu finden.

Die zweite, zu Beginn dieses Abschnitts skizzierte, Anforderung, dass neben dem eigentlich Wert eines Unterknotens auch die empirische Datenbasis übergeben werden muss, wird realisiert durch die Klasse `ArgumentInput`. Sie kombiniert einen Wert aus einer Datenreihe eines untergeordneten Qualitätsknotens in Form einer `Value`-Instanz mit den empirischen Vergleichsdaten der entsprechenden Datengruppe in Form einer Instanz der Klasse `ComparisonData`. Durch diese Kombination der beiden Informationen in eine gemeinsame Klasse ist es nicht erforderlich, dass die Algorithmen zur Implementierung der Berechnungsvorschriften Kenntnis über das Konzept der Datengruppen und -reihen haben müssen, da alle für sie relevanten Informationen in `ArgumentInput` gruppiert sind. Dies fördert die Konzepte der *Trennung von Zuständigkeiten* sowie der *losen Kopplung*. Darüber hinaus ist es möglich, von einer `ArgumentInput`-Instanz zu dem Qualitätsknoten zu navigieren, zu dem der gekapselte Wert und die Vergleichsdaten gehören. Dies ist im Rahmen der Berechnungs-Algorithmen notwendig, da dort für jeden Qualitätsknoten der korrekte Wert aus dem `ArgumentContainer` extrahiert werden muss.

5.3.5.4 Algorithmen für komplexe Funktionen

Für jede in Qualitätsmodellen mögliche Berechnungsvorschrift, die dort in Form von Implementierungen des Interface `ValueSpecificationComplexFunction` (siehe S. 51) modelliert werden, stellt die Auswertungskomponente eine Implementierung des der Vorschrift zugrunde liegenden Algorithmus bereit.

Die Struktur der Algorithmen für Berechnungsvorschriften wird in Abbildung 5.17 wiedergegeben.

Alle Algorithmus-Implementierungen verwenden die abstrakte Basisklasse `ComplexFunctionCalculator` als Oberklasse, die wiederum eine Unterklasse von `Calculator` ist. Diese Basisklasse stellt eine Standardimplementierung von `canProvideComparisonData()` bereit, da davon ausgegangen

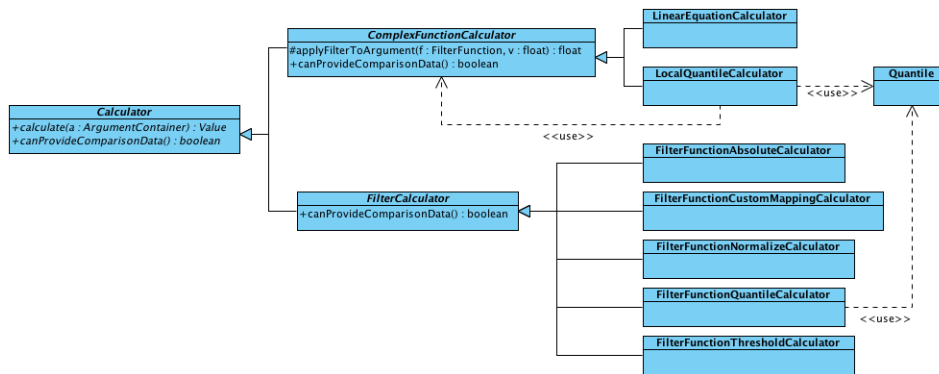


Abbildung 5.17: Algorithmen für Berechnungsvorschriften

wird, dass komplexe Berechnungsvorschriften üblicherweise in der Lage sind, empirische Datenbasen zu berechnen. Ist dies für eine konkrete Vorschrift nicht der Fall, kann diese Tatsache durch Überschreiben der Methode in der entsprechenden Klasse signalisiert werden. Darüber hinaus stellt die abstrakte Klasse ihren Unterklassen die Methode `applyFilterToArgument()` zur Verfügung, mit deren Hilfe ein etwaiger Filter, der im Qualitätsmodell für ein Argument der Berechnungsvorschrift definiert ist (siehe S. 51), auf den Wert des Argumentes angewendet werden kann.

Für jede Berechnungsvorschrift, die im Qualitätsmodell mit einer Qualitätseigenschaft verknüpft werden kann, steht eine Implementierung bereit:

- **Lineare Gleichungen**, die im Qualitätsmodell durch Instanzen von `ValueSpecificationLinearEquation` repräsentiert werden, werden implementiert durch die Klasse `LinearEquationCalculator`.

Der Algorithmus iteriert über die Liste der Koeffizienten der linearen Gleichung, die in der gegebenen Instanz von `ValueSpecificationLinearEquation` in Form von `LinearEquationArgument`-Instanzen vorliegen und multipliziert den jeweiligen Koeffizienten mit dem Wert aus der `ArgumentInput`-Instanz, der zum entsprechenden Unterknoten gehört. Wenn ein Filter für einen Unterknoten definiert ist, wird dieser vor der Multiplikation auf den Wert mithilfe von `applyFilterToArgument()` angewandt. Die einzelnen Ergebnisse der Multiplikationen werden aufsummiert und gekapselt in einer Instanz von `Value` als Ergebnis zurückgegeben.

- **Quantil-Berechnungen** werden im Qualitätsmodell durch Instanzen von `ValueSpecificationLocalQuantile` definiert. Der entsprechende Algorithmus zu deren Berechnung im Rahmen der Auswertung wird durch die Klasse `LocalQuantileCalculator` implementiert.

Der Algorithmus verwendet einen zweiten Algorithmus für Berechnungsvorschriften zur Berechnung der mit der Quantil-Berechnungsvorschrift verknüpften Hilfsfunktion. Darüber hinaus verwendet er die Klasse `Quantile`, die eine generische und universell einsetzbare Implementierung von Quantil-Berechnungen bereitstellt.

Bei der Quantil-Berechnung kommt ein **dreischnittiger Algorithmus** zum Einsatz:

1. **Absolutwert-Berechnung:**

Mithilfe der in der Eigenschaft `complexHelperFunction` mit der Berechnungsvorschrift assoziierten Hilfsfunktion wird ein Absolutwert für die Qualitätseigenschaft ermittelt.

2. **Berechnung empirischer Daten:**

Basierend auf den empirischen Datenbasen der untergeordneten Knoten, die über die jeweiligen `ArgumentInput`-Instanzen abgefragt werden können, wird durch Einsatz der einzelnen Werte der Datenbasen in die assoziierte Hilfsfunktion eine *lokale empirische Datenbasis* berechnet.

3. **Berechnung der Quantil-Einordnung:**

Die Quantil-Einordnung des im ersten Schritt ermittelten Absolutwerts in der im zweiten Schritt ermittelten Datenbasis wird mithilfe einer Instanz von `Quantile` ermittelt.

Das Ergebnis der Ausführung des Algorithmus ist eine Instanz der Klasse `CalculatedQuantileValue`. Dabei enthält diese als darstellbaren Wert die Quantil-Einordnung, die im dritten Schritt berechnet worden ist; gleichzeitig (siehe S. 71) kapselt sie für die Verwendung in weiteren Berechnungsschritten übergeordneter Qualitätsknoten allerdings auch den im ersten Schritt berechneten Absolutwert.

5.3.5.5 Algorithmen für Filterfunktionen

In Ergänzung zu den Algorithmen für Berechnungsvorschriften stellt das Auswertungswerkzeug darüber hinaus Algorithmen für die Filterfunktionen

(siehe Abschnitt 5.2.4.3), die für eine Qualitätseigenschaften eine optionale Vorverarbeitung der Werte untergeordneter Qualitätsknoten vornehmen.

Deren Architektur wird in Abbildung 5.17 wiedergegeben.

Alle Filter-Implementierungen verwenden die abstrakte Klasse `FilterCalculator` als gemeinsame Basisklasse. Diese Klasse, die eine Unterklasse von `Calculator` ist, implementiert die Methode `canProvideComparisonData()`, so dass diese stets `FALSE` zurück gibt — bei Filtern ist nicht vorgesehen, dass diese über empirische Datenbasen verfügen können.

Für alle Filterfunktionen, die im Qualitätsmodell als Unterklassen von `FilterFunction` modelliert werden, existiert im Auswertungswerkzeug eine Implementierung:

1. **Identische Abbildungen**, die im Qualitätsmodell durch die Klasse `FilterFunctionAbsolute` repräsentiert werden, werden im Auswertungswerkzeug durch die Klasse `FilterFunctionAbsoluteCalculator` implementiert.
2. **Quantil-Filter** werden im Qualitätsmodell durch Instanzen der Klasse `FilterFunctionQuantile` repräsentiert. Im Auswertungswerkzeug erfolgt ihre Implementierung in der Klasse `FilterFunctionQuantileCalculator`, die wie `LocalQuantileCalculator` die Klasse `Quantile` als Hilfsklasse zur Berechnung von Quantileinordnungen verwendet.
3. **Skalierungs-Filter** werden im Qualitätsmodell durch Instanzen der Klasse `FilterFunctionNormalize` repräsentiert. Sie werden im Auswertungswerkzeug durch die Klasse `FilterFunctionNormalizeCalculator` umgesetzt.
4. Filter zur **Abbildung auf benutzerdefinierte Wertebereiche**, durch `FilterFunctionCustomMapping` im Qualitätsmodell repräsentiert, werden durch die Klasse `FilterFunctionCustomMappingCalculator` im Auswertungswerkzeug implementiert.
5. **Schwellenwert-Filter** werden durch die Klasse `FilterFunctionThresholdCalculator` im Auswertungswerkzeug implementiert.

5.3.6 Frontend

Die Aufgabe des Frontends des Auswertungswerkzeuges besteht in der Bereitstellung einer grafischen Benutzerschnittstelle zum **Vorbereiten** und **Durchführen einer Auswertung**.

Gemäß dieser beiden Aufgaben setzt sich das Frontend aus zwei Teilen zusammen:

- Ein Teil ist für die Konfiguration der Auswertung mithilfe von Konfigurationswizards zuständig.
- Der zweite Teil erlaubt es, die Vermessung und Berechnung einer Auswertung zu starten und stellt mehrere grafische Sichten auf die Datenmodelle, die Ergebnis der Auswertung sind (siehe Abschnitt 5.3.2), zur Verfügung. Außerdem ist es dort möglich, die Datenmodelle abzuspeichern sowie zur erneuten Bearbeitung zu öffnen.

Wie das Frontend des Qualitätsmodell-Editors auch basiert diese Benutzerschnittstelle auf dem Architekturmuster **Model/View/Controller (MVC)**, das als Muster zur Erstellung grafischer und mit einem Benutzer interagierender Anwendungen etabliert ist.

- Beim **Model** handelt es sich um eine Instanz des Datenmodells der Auswertungskomponente. Diese kapselt die Ergebnisse der aktuell aktiven bzw. einer vergangenen, wieder geladenen Auswertung.
- Die unterschiedlichen grafischen Sichten auf das Model werden durch die **Views** bereitgestellt. Es handelt sich dabei technisch um Unterklassen der abstrakten Klasse `FrontendComposite`, die wiederum eine Unterklasse von `Composite` aus dem SWT-Rahmenwerk [SWT] ist.

Die Sicht auf eine Datenreihe in Form eines XY-Diagramms verwendet darüber hinaus die Bibliothek `JFreeChart` zur Erstellung des Diagramms. Der Export einer Datenreihe als CSV-Datei erfolgt unter Verwendung der Bibliothek `opencsv`.

- Die Klasse `EvaluationToolFrontend` übernimmt als **Controller** die Koordination von Model und Views und dient als zentraler Einstiegspunkt in den Kontrollfluss des Frontends.

Der Controller stellt sicher, dass die verschiedenen grafischen Sichten über Änderungen, die sich an Datenmodell ereignen, Kenntnis erhalten. Dies wird realisiert über das Observer-Entwurfsmuster [GHJV95],

indem die Views als Beobachter bei den Instanzen des Datenmodells registriert werden. Das Datenmodell verfügt aufgrund der Verwendung von EMF bereits über Unterstützung für das Observer-Muster, weshalb dort keine weiteren Anpassungen notwendig macht. Der Vorteil der Verwendung dieses Architektur-Musters ist darin zu sehen, dass es dadurch ohne weiteres möglich ist, beliebige grafische Sichten auf eine Auswertung zu verwenden und diese bei Änderungen am zugrunde liegenden Modell automatisch zu synchronisieren.

Außerdem ist der Controller verantwortlich für die Interaktion mit der Logikkomponente, indem durch ihn eine Instanz der Klasse `Evaluator` erstellt und die Auswertung durch Aufruf von `evaluate()` gestartet wird.

Darüber hinaus stellt `EvaluationToolFrontend` die Infrastruktur zum Laden und Speichern von Datenmodell-Instanzen zur Verfügung und ist zuständig für das Initialisieren der grafischen Elemente beim Laden des Auswertungswerkzeuges.

Technisch handelt es sich bei `EvaluationToolFrontend` um eine Unterklasse von `MultiPageEditorPart`. Hierbei handelt es sich um eine abstrakte Klasse, die Bestandteil der Eclipse-Plattform ist und die Basis für die Erstellung von *Editoren* mit mehreren, auf einer Registerkarte angeordneten Seiten ist. Unter dem Begriff "Editor" sind in der Eclipse-Plattform alle die Komponenten zusammengefasst, die eine Datei auf irgendeine Art und Weise verändern. Dies umfasst sowohl Texteditoren wie den standardmäßig aktivierten Editor für Java-Dateien als auch das vorliegende Frontend des Auswertungswerkzeug, das Dateien, in denen Repräsentationen des Datenmodells abgelegt sind, verändert. Auf den einzelnen Seiten des Editors sind die verschiedenen Sichten auf die Datenreihen der einzelnen Datengruppen der Auswertung untergebracht.

Die Verwendung von `MultiPageEditorPart` als Grundlage für das Frontend hat mehrere Vorteile:

- Der *Grad an Wiederverwendung* ist hoch, da Funktionalität wie das Speichern und Öffnen von Dateien direkt aus `MultiPageEditorPart` übernommen werden kann.
- Die Möglichkeit zur Anordnung mehrerer Seiten im Editor auf Registerkarten deckt die Anforderung, dass es auf eine Auswertung mehrere grafische Sichten, zwischen denen nahtlos umgeschaltet

werden können muss, gut ab. Außerdem ist es somit möglich, mit überschaubarem Entwicklungsaufwand Ergebnisse zu erzielen.

- Die Klasse `MultiPageEditorPart` implementiert bereits die notwendigen Schnittstellen, die für die Integration des Editors in die Eclipse-Plattform erforderlich sind. Auch dies fördert die *Geschwindigkeit der Entwicklung*.

5.3.6.1 Konfigurationswizards

Die Aufgabe des Konfigurationswizards im Auswertungswerkzeug besteht im Erfassen der Informationen, die in die Konfiguration der Auswertung in Form einer Instanz von `RetrievalConfiguration` einfließen. Er wird beim Anlegen einer neuen Auswertung verwendet, kann jedoch auch nachträglich für eine bestehende Auswertung aufgerufen werden, um deren Konfiguration nachträglich anzupassen.

Wie in der Beschreibung der Wizard-Architektur im Qualitätsmodell-Editor bereits gezeigt (siehe Abschnitt 5.2.5.1), besteht ein Konfigurationswizard stets aus einer Unterklasse der abstrakten Klasse `Wizard`. Die einzelnen Seiten des Wizards werden in Form von Unterklassen von `WizardPage` repräsentiert.

Der Wizard wird als *Extension Point* in der Eclipse-Plattform registriert, so dass er beim Anlegen einer neuen Auswertung gestartet wird.

Das Base Filter Widget

Im Kontext der Konfigurationswizards findet darüber hinaus die Komponente *Base Filter Widget* Verwendung. Es handelt sich dabei um den Bestandteil des Frontends, mit dessen Hilfe die `basefilter`- und `grouping-Parameters`-Elemente der Konfiguration für projektbezogene Daten sowie empirische Vergleichsdaten auf grafischem Wege definiert werden können.

Die Komponente stellt sowohl grafische Benutzerelemente als auch entsprechende Businesslogik zur Verarbeitung der Eingaben aus den Benutzerelementen zur Verfügung. Die Struktur und die Optik der Benutzerelemente orientiert sich stark an den Gegebenheiten in Bugzilla sowie BugzillaMetrics. Abbildung 5.18 gibt die Situation bei BugzillaMetrics wieder: Es ist dort über Auswahllisten möglich, die Auswertung auf bestimmte Produkt-Klassifikationen, Produkte, Komponenten, Produkt-Versionen, Meilenstei-

ne und Bugs bzw. Verbesserungswünsche einzuschränken. Darüber hinaus kann über eine weitere Auswahlliste die Gruppierung festgelegt werden. Alle Auswahllisten erlauben eine Mehrfachauswahl. Außerdem ist die Produkt-Auswahlliste insofern von der Auswahlliste zur Produkt-Klassifikation abhängig, als dass eine Selektion einer bestimmten Klassifikation die zur Verfügung stehenden Einträge in der Produkt-Auswahlliste einschränkt. Nach dem gleichen Prinzip sind die Auswahllisten für Komponenten, Versionen und Meilensteine von der Produkt-Auswahlliste abhängig.

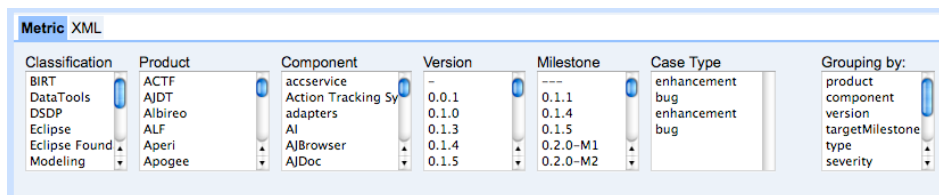


Abbildung 5.18: Das Base Filter Widget von BugzillaMetrics

Die Konfiguration des Aussehens des Base Filter Widget erfolgt über eine XML-Konfigurationsdatei, in der festgelegt wird, welche Auswahlfelder angezeigt werden, in welcher Reihenfolge dies erfolgt, welche Überschriften sie tragen und wie breit sie sind. Eine exemplarische Konfiguration, die den Gegebenheiten in Abbildung 5.18 entspricht, ist in folgendem Listing dargestellt:

```
<settings>
  <frontendPresentationSettings>
    <baseFilter>
      <baseFilterRow>
        <enumList title=" Classification"
          title=" classification"
          width=" 90px" />
        <enumList title=" Product"
          field=" product"
          width=" 115px" />
        <enumList title=" Component"
          field=" component"
          width=" 115px" />
        <enumList
          title=" Version"
          field=" version"
          width=" 90px" />
        <enumList
          title=" Milestone"
```

```

        field="targetMilestone"
        width="90px" />
<enumList
    title="Case_Type"
    field="type"
    width="110px" />
<space width="20px" />
<groupingParametersListBox
    title="Grouping_by:"
    width="100px">
    <groupingParameter>product</groupingParameter>
    <groupingParameter>component</groupingParameter>
    <groupingParameter>version</groupingParameter>
    <!-- [...] -->
</groupingParametersListBox>
</baseFilterRow>
</baseFilter>
</frontendPresentationSettings>
</settings>

```

Das Base Filter Widget für das Auswertungswerkzeug muss die gleiche Funktionalität unter Verwendung der gleichen Konfigurationsdateien zur Verfügung stellen. Dazu muss es mehrere Funktionalitäten bereitstellen:

1. Die Abfrage der möglichen Einträge für die Auswahllisten muss über den XML-RPC-Webservice (siehe Abschnitt 5.4) erfolgen.
2. Die Komponente muss grafische Benutzerelemente für die Darstellung der Auswahllisten bereitstellen. Diese müssen das Management von Abhängigkeiten zwischen Auswahllisten implementieren.
3. Für die XML-Konfigurationsdatei zur Festlegung von Reihenfolge und Größe der Auswahllisten muss ein Parser bereitgestellt werden.
4. Nach erfolgter Auswahl muss diese in das von BugzillaMetrics verwendete XML-Konfigurationsformat überführt werden.
5. Beim erneuten Aufruf des Wizards muss das XML-Konfigurationsformat von BugzillaMetrics eingelesen und die Auswahllisten müssen entsprechend vorselektiert werden.

Die Einträge der Auswahllisten werden im Folgenden *Parameter* genannt. Ihre Abfrage erfolgt über die Klasse `DataFacade`, die den XML-RPC-Webservice von BugzillaMetrics anspricht. Die verschiedenen Arten von Parametern werden in einem Domänenmodell mit der abstrakten Oberklasse

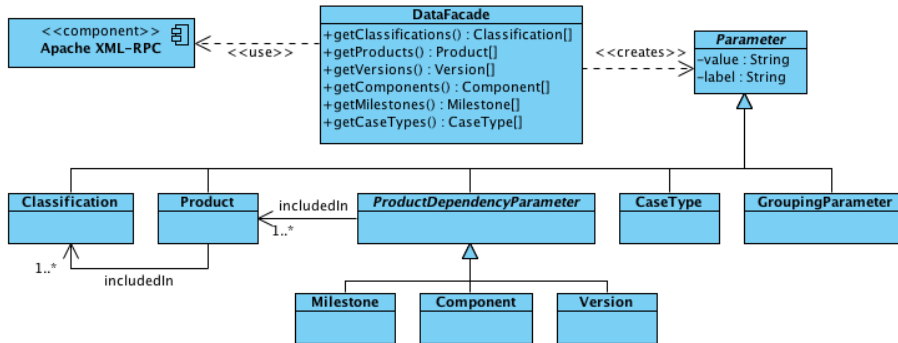


Abbildung 5.19: Das Domänenmodell des Base Filter Widget

Parameter repräsentiert, das in Abbildung 5.19 wiedergegeben ist.

Jeder Parameter differenziert zwischen einem *anzuzeigenden Wert* in Form des Attributs `label` und einem *Datenbankwert*, der im Attribut `value` enthalten ist. Der anzuzeigende Werte wird dem Benutzer als Beschriftung des entsprechenden Eintrages in der Auswahlliste präsentiert, während der Datenbankwert derjenige Wert ist, mit dem der Parameter intern in Bugzilla(Metrics) verwendet wird. Je nach Art des Parameters sind anzuzeigender Wert und Datenbankwert identisch oder aber der Datenbankwert ist eine numerische ID, die vom anzuzeigenden Wert als Beschriftung abweicht.

Die konkreten Ausprägungen von Parametern werden modelliert in den Unterklassen `Classification`, `Product`, `Milestone`, `Component`, `Version`, `CaseType` und `GroupingParameter`. Wie bereits zu Beginn des Abschnitts erwähnt, sind die Parameter im Produkt-Auswahlfeld abhängig von der Selektion in der Auswahlliste zur Produkt-Klassifikation. Im Domänen-Modell wird dieser Tatsache dadurch begegnet, dass die Parameter vom Typ `Product` eine `includedIn`-Assoziation zu einer oder mehreren Instanzen von `Classification` unterhalten. Die gleiche Situation liegt vor zwischen Instanzen von `Milestone`, `Component` und `Version` und der Klasse `Product`. Auch hier wird die Abhängigkeitsbeziehung der Einträge der Auswahllisten im Domänenmodell durch die Assoziation `includedIn` modelliert. Da die Beziehung für alle drei Klassen identisch ist, wird sie von der abstrakten Oberklasse `ProductDependencyParameter` modelliert, von der `Milestone`, `Component` und `Version` abgeleitet werden.

Aufbauend auf den Parametern, die als Instanzen der Klassen aus dem Domänenmodell vorliegen, realisiert die zweite Teilkomponente des Base Filter Widget die in Punkt 2 geforderte grafische Benutzerschnittstelle. Deren Struktur ist in Abbildung 5.20 wiedergegeben.

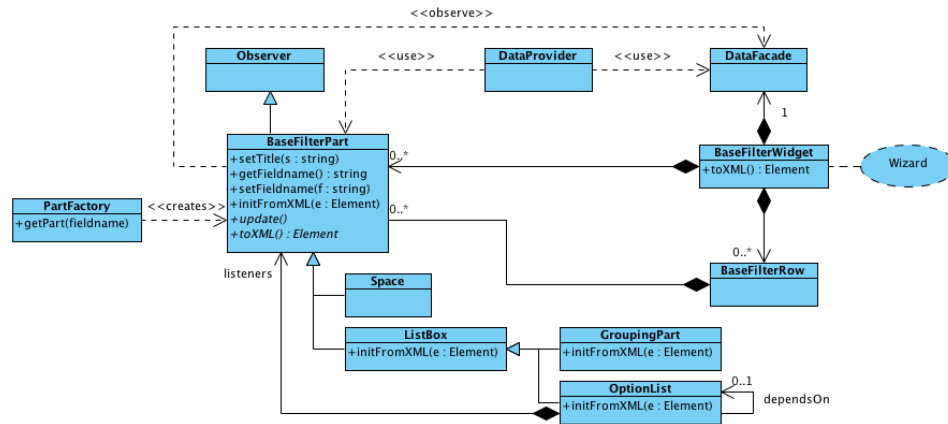


Abbildung 5.20: Grafische Benutzerschnittstelle für das Base Filter Widget

Der zentrale Einstiegspunkt ist die Klasse `BaseFilterWidget`. Sie wird auf den entsprechenden Seiten der Konfigurationswizards instanziiert und erhält als Eingabe den Namen der Datei, welche die XML-Konfiguration `<frontendPresentationSettings />` enthält. Intern implementiert die Klasse eine Methode zum Einlesen und Verarbeiten der Konfigurationsdatei auf Basis der JDOM-Bibliothek, wie es in Punkt 3 gefordert wird. Diese Methode verwendet die Fabrik-Klasse `PartFactory`, um für ein XML-Element die passende grafische Repräsentation zu erzeugen.

Jedes Element der grafischen Benutzerschnittstelle ist realisiert als Unterklasse der abstrakten Klasse `BaseFilterPart`. Es verfügt dementsprechend über eine Überschrift im Attribut `title` und kann einem Feldnamen aus der XML-Konfiguration über das Attribut `fieldname` zugeordnet werden.

`BaseFilterPart` implementiert das Interface `Observer`. Dadurch ist es den grafischen Elemente möglich, sich als Beobachter an anderen Objekten zu registrieren, um über deren Änderungen informiert zu werden. Es handelt sich dabei um eine Implementierung des Observer-Entwurfsmuster nach [GHJV95]. Dieses Entwurfsmuster wird hier verwendet, um die Abhängigkeiten zwischen einzelnen Auswahllisten zu realisieren: So wird die grafi-

sche Benutzerschnittstelle für die Auswahl der Produkte bei der Schnittstelle für die Produkt-Klassifikationen als Beobachter registriert. Ändert sich nun die Selektion der Produkt-Klassifikation, wird die Produkt-Auswahlliste darüber informiert und blendet darauf nur noch die Parameter ein, die in der Klassifikations-Selektion enthalten sind. Für die Abhängigkeiten von Komponenten, Versionen und Meilensteinen zur Produkt-Auswahlliste wird analog verfahren. Bestandteil des Entwurfsmusters in Java ist die Implementierung der Methode `update`, die in beobachtenden Objekten aufgerufen, wenn das Subjekt der Beobachtung eine Änderung signalisiert. Deren Implementierung ist im vorliegenden Fall den Unterklassen von `BaseFilterPart` überlassen. Darüber hinaus registrieren sich Objekte vom Typ `BaseFilterPart` auch als Beobachter an der `DataFacade`, um über dort anfallende Änderungen informiert zu werden.

Der **Vorteil der Verwendung des Observer-Entwurfsmuster** ist darin zu sehen, dass es somit möglich ist, mit maximaler Flexibilität das Management von Abhängigkeiten innerhalb der Benutzerschnittstelle zu realisieren. Außerdem wird damit der asynchronen Natur der XML-RPC-Schnittstelle gerecht, da die Schnittstelle nicht auf das Laden der Werte in der `DataFacade` warten muss, sondern von dieser nach dem Ladevorgang über die neuen Parameter informiert wird.

Das Base Filter Widget unterstützt drei Arten von grafischen Elementen:

1. Einfache Auswahllisten werden durch die Klasse `Listbox` modelliert. Die Klasse `GroupingPart` ist eine Spezialisierung davon, welche die Auswahlliste für die Gruppierungs-Parameter realisiert. Eine Spezialisierung ist dort erforderlich, da die Parameter für diese Auswahlliste nicht über die `DataFacade` geladen werden, sondern direkt in der XML-Konfiguration in `<frontendPresentationSettings />` verfügbar sind.
2. Erweiterte Auswahllisten werden durch die Klasse `OptionList` realisiert. Sie unterscheiden sich in zwei Punkten von einfachen Auswahllisten:
 - Es wird unterschieden zwischen einem Datenbankwert und dem anzuzeigenden Wert eines Parameters. Es muss daher eine Abbildung von Beschriftungen auf Datenbankwerte erfolgen.
 - Instanzen von `OptionList` können von anderen grafischen Elementen im Rahmen des bereits erläuterten Observer-Entwurfsmu-

ster auf Änderungen hin überwacht werden. In der aktuellen Version hat es sich als ausreichend erwiesen, wenn nur `OptionList`-Objekte eine solche Abhängigkeitsbeziehung untereinander realisieren. Dies erfolgt über die Assoziation `dependsOn`.

3. Horizontale Abstände zwischen einzelnen Elementen werden über die Klasse `Space` realisiert. Diese Klasse entspricht dem Konfigurationstag `<space />` und verfügt über ein Attribut `width`, das die Größe des Abstands angibt.

Das in Punkt 4 aufgeführte Überführen der Auswahl in die XML-Konfiguration von `BugzillaMetrics` wird durch die Methode `toXML` der Klasse `BaseFilterWidget` realisiert. Diese iteriert über die Menge der ihr zugeordneten Instanzen von `BaseFilterPart` und ruft für jede deren `toXML`-Implementierung auf. Diese bildet die im betreffenden grafischen Element getroffene Auswahl in XML ab und gibt dieses zurück. Die Menge aller Rückgaben wird in `BaseFilterWidget` zu einem gemeinsamen XML-Dokument vereinigt und an den Aufrufer zurückgegeben.

In Punkt 5 wird gefordert, dass Elemente von Auswahllisten bei Aufruf des Widgets ggf., basierend auf einer bestehenden Auswertungskonfiguration, vorausgewählt werden. Dies wird durch die Methode `initFromXML` der entsprechenden Instanz realisiert.

Bei der statischen Klasse `DataProvider` handelt es sich um einen Proxy, der zwischen Feldnamen, die grafischen Benutzerelementen zugeordnet sind, und den entsprechenden Methoden in der `DataFacade`-Klasse vermitteln kann. Über den `DataProvider` erfragen die einzelnen `BaseFilterPart`-Instanzen die anzuzeigenden Parameter basierend nur auf Kenntnis des ihnen zugeordneten Feldnamens. Der Vorteil dieser Proxy-Klasse ist die Förderung einer generischen Architektur, da die grafischen Elemente somit einzig auf Basis des ihnen zugeordneten Feldnamens mit der Außenwelt interagieren.

5.4 Architektur der RPC-Schnittstelle für BugzillaMetrics

Zur Erfüllung der Anforderung, dass das Auswertungswerkzeug und `BugzillaMetrics` zum Zeitpunkt einer Auswertung auf physikalisch getrennten, durch ein Computernetzwerk verbundenen, Rechnern installiert sind, ist

es erforderlich, eine netzwerkbasierte *Kommunikationsschnittstelle* zwischen den beiden Systemen zu realisieren.

In diesem Abschnitt wird zuerst in die Problemstellung eingeführt und mit XML-RPC eine zur Realisierung geeignete Komponente vorgestellt. Danach wird die konkrete architektonische Umsetzung im vorliegenden Fall beschrieben.

5.4.1 Einleitung

Die Java-Plattform, mit der sowohl die Auswertungskomponente als auch BugzillaMetrics realisiert sind, bietet mit der *Remote Method Invocation*-Komponente (*RMI*) [RMI] ein Werkzeug, um eine solche Schnittstelle bereitzustellen. In der Regel laufen die entsprechenden Dienste dabei auf den Netzwerkports 1098 und 1099, die in vielen durch Firewalls gesicherten Netzwerken nicht oder nur mit großem Aufwand freigeschaltet werden können. Darüber hinaus ist RMI nur zur Kommunikation zwischen in Java realisierten Systemen verwendbar.

Aufgrund der skizzierten Nachteile, die sich aus der Verwendung von RMI ergeben, ist es nahe liegend, einen *Webservice* als Grundlage für die Kommunikationsschnittstelle zu verwenden. Diese zeichnen sich dadurch aus, dass sie auf offenen Standards wie XML basieren und somit zur Kommunikation zwischen Systemen geeignet sind, die auf Basis verschiedener Plattformen oder Rahmenwerke realisiert sind: Bei XML [XMLb] handelt es sich um ein durch das World Wide Web Consortium standardisiertes rein textorientiertes Format, das einen hohen Grad an Interoperabilität ermöglicht, da jede Programmiersprache, mit der Zeichenketten verarbeitet werden können, ebenfalls XML-Daten verarbeiten kann. Darüber hinaus verwenden Webservices das HTTP-Protokoll als Transportmedium, was in Firewall-Konfigurationen weitaus unproblematischer integriert werden kann als beispielsweise RMI.

Ein Standard, der Webservices implementiert, ist XML-RPC [SJD01]. Es basiert auf XML als Serialisierungsformat und verwendet HTTP als Transportmedium. Eine auf XML-RPC basierende Kommunikation erfolgt zwischen zwei Systemen, dem *Klienten* und dem *Server*: Der Server bietet eine Menge von *Diensten* an, die über *Methodenaufrufe* gegen einen URL, den sog. *Endpoint*, erreichbar sind. Der Klient ruft Methoden des Servers auf und verarbeitet die Ergebnisse dieser Aufrufe bzw. gibt sie an die ihn verwendende Komponente zurück.

XML-RPC-Aufrufe sind durch das folgende Ablaufschema charakterisiert:

1. Das Programm, das den Klienten verwendet, führt einen Methodenaufruf durch Verwendung des Klienten durch. Dabei gibt es an, wie diese Methode heißt, welche Parameter zu übergeben sind und wie der Endpoint des zu verwendenden Servers lautet.
2. Der XML-RPC-Klient serialisiert den Methodennamen und die Namen und Werte der Parameter in eine interne XML-Struktur. Er führt daraufhin eine HTTP POST-Anfrage gegen die URL des gegebenen Endpoints durch, in der die serialisierten Informationen übergeben werden.
3. Der HTTP-Server, der für den Endpoint zuständig ist, empfängt die POST-Anfrage und leitet sie an den XML-RPC-Server weiter.
4. Der XML-RPC-Server deserialisiert die übermittelten XML-Daten und weiß damit, welche Methode aufgerufen wurde und welche Parameter dafür vorliegen. Er ruft diese Methode auf und übergibt ihr die Parameter.
5. Die aufgerufene Methode wird ausgeführt und gibt einen Rückgabewert an den XML-RPC-Server zurück, der sodann vom Server in eine XML-Struktur serialisiert wird.
6. Der für den Endpoint zuständige HTTP-Server gibt die XML-Daten aus dem vorherigen Schritt als Antwort an den aufrufenden Klienten zurück.
7. Der XML-RPC-Klient deserialisiert die Antwort des Servers und erhält somit den Rückgabewert. Er gibt diesen an das aufrufende Programm zurück.
8. Das aufrufende Programm setzt die Verarbeitung mit dem erhaltenen Wert fort.

5.4.2 Implementierung

Im vorliegenden Fall wird die XML-RPC-Implementierung *Apache XML-RPC* [XMLa] in Version 3.1 verwendet. Das freie Softwarepaket beinhaltet sowohl einen Server als auch einen Klienten für die Java-Umgebung.

Der HTTP-Server, über den der Endpoint erreichbar ist, wird mithilfe des Apache Tomcat 5.5 [Tom] Servlet-Containers realisiert. Der XML-RPC-Server wird dabei in einem *Servlet* gekapselt, das über den Tomcat-Container aufgerufen wird.

Die Struktur des XML-RPC-Servers ist in Abbildung 5.21 wiedergegeben.

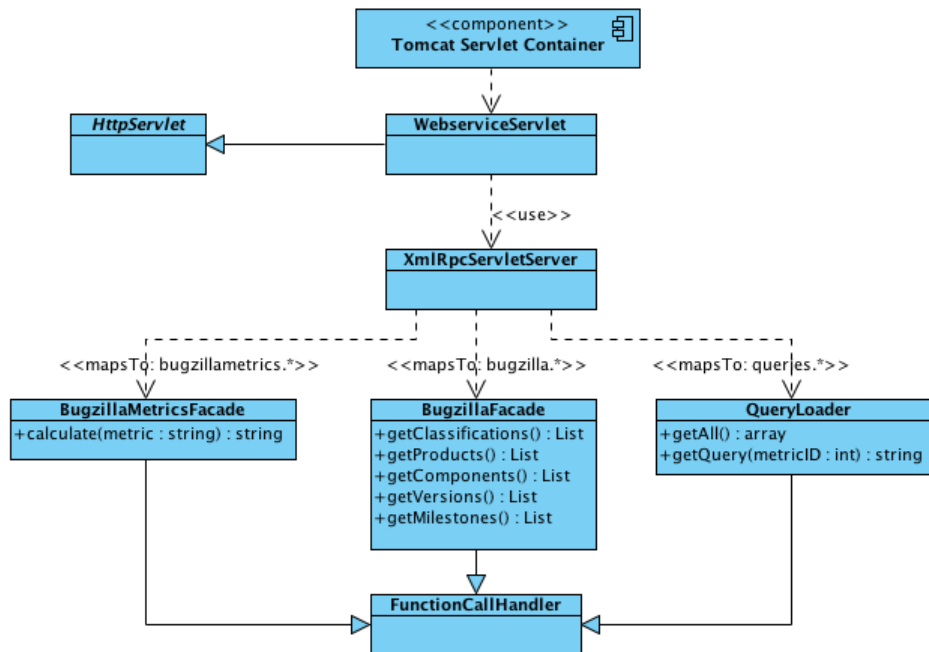


Abbildung 5.21: Struktur des XML-RPC-Servers für BugzillaMetrics

Eine Anfrage, die ein Klient an den Endpoint stellt, werden vom Servlet an eine Instanz der Klasse *XmlRpcServletServer* delegiert. Diese Instanz ermittelt auf Basis des Methodennamens, welcher *Handler* für Bearbeitung der Anfrage zuständig ist. Zur Zeit verfügt der Webservice über einen Handler, der als Fassade vor BugzillaMetrics agiert, einen, der Zugriff auf die zugrunde liegende Bugzilla-Datenbank erlaubt, und einen weiteren Handler, der die in BugzillaMetrics verwaltete Liste *My Queries* abfragbar macht.

Alle Handler sind Spezialisierungen der abstrakten Klasse *FunctionCallHandler*, in der häufig benötigte Hilfsmethoden implementiert sind. Da der

Apache XML-RPC-Server das Java-Reflection-API verwendet, um Methodennamen auf entsprechende Java-Methode abzubilden, müssen Methoden, die über den Webservice aufrufbar sind, als öffentliche Methoden des entsprechenden Handlers implementiert werden. Dadurch wird dann beispielsweise der Aufruf `bugzillametrics.calculate` auf die Methode `calculate` der Klasse `BugzillaMetricsFacade` abgebildet.

Kapitel 6

Evaluierung

Dieses Kapitel befasst sich mit einer Betrachtung der Qualität der entwickelten Werkzeuge und des Entwicklungsprozesses.

6.1 Qualität der Produkte

Zur Bewertung der Produktqualität kann die Norm ISO/IEC 9126 [JKC04] herangezogen werden. Sie gliedert die Qualität eines Softwareproduktes in die Bereiche Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit sowie Übertragbarkeit. Im deutschsprachigen Umfeld stand daneben auch die DIN-Norm 66272 zur Bewertung von Software zur Verfügung. Da diese jedoch im Jahr 2006 durch das DIN ersatzlos gestrichen wurde, wird sie hier nicht weiter betrachtet.

Mit Ausnahme des Merkmals “Übertragbarkeit” wird in allen Fällen auf die Diskussion des Unterqualitätsmerkmals “Konformität” verzichtet, da die Werkzeuge keinen externen Normen oder Gesetzen entsprechen müssen.

6.1.1 Funktionalität

Angemessenheit: Die in der Anforderungsanalyse ermittelten und im Rahmen der Anforderungsspezifikation erfassten funktionalen Anforderungen werden durch entsprechende Testfälle erfasst und können mit deren Hilfe verifiziert werden. Es wird dadurch sichergestellt, dass die funktionalen Anforderungen korrekt umgesetzt worden sind.

Dass die ermittelten funktionalen Anforderungen auch den tatsächlich benötigten Anforderungen entsprechen, wird dadurch deutlich, dass sich mit dem Werkzeug sowohl Qualitätsmodelle zum Ermitteln der erreichten Qualitätsstufe (vergleichbar zu den Reifegraden bei CMMI [CKS06]) als auch Modelle, in denen Qualitätsmerkmale gruppen- oder aspektorientiert strukturiert sind (beispielsweise die Merkmale “Sorgfalt bei der Planung” und “Bugzilla-Verwendung”), erstellen lassen. Für eine detaillierte Ausführung zu diesen beiden Modellarten und deren Verwendung im Rahmen von Qualitätsbetrachtungen siehe [Sch09].

Richtigkeit: Berechnungen, für welche die Korrektheit der Ergebnisse verifiziert werden muss, finden im Auswertungswerkzeug im Rahmen von Berechnungsvorschriften und Filtern statt. Für diese existieren Testfälle zur rechnergestützten Überprüfung. Darüber hinaus wurden praxisbezogene Anwendungstests durchgeführt, deren Ergebnisse plausibel erschienen.

Interoperabilität: Durch die Verwendung von XML als Speicherformat für Qualitätsmodelle und Auswertungen wird ein hoher Grad an Interoperabilität erreicht. Die Komponente, die BugzillaMetrics um einen XML-RPC-Webservice erweitert, erfüllt dieses Qualitätsmerkmal ebenfalls, da dort mit XML-RPC ein standardisiertes und von vielen Plattformen unterstütztes Verfahren zum Einsatz kommt.

Sicherheit: Sicherheitsrelevante Aspekte spielen nur beim XML-RPC-Webservice eine Rolle. Dort wird sichergestellt, dass nur die vordefinierten Metriken abgefragt werden können, die explizit als öffentlich zugänglich markiert worden sind.

6.1.2 Zuverlässigkeit

Reife: Fehler in der Software sind trotz vorhandener Testfälle nicht auszuschließen, da beide Werkzeuge noch keine intensive praktische Anwendung erfahren haben.

Fehlertoleranz: Fehler, die bei einer Auswertung auftreten, führen zu deren Abbruch. Es ist jedoch jederzeit möglich, die Auswertung nach Behebung des Fehlers zu wiederholen. Außerdem ist diese Art von Fehlern auf eine einzelne Auswertung in Form einer `qme`-Datei beschränkt und hat keine Auswirkung auf weitere Dateien, die andere Auswertungen enthalten. Zu den

Fehlern, die auftreten können, zählen beispielsweise Metrik-Spezifikationen, die nicht dem XML-Schema entsprechen, das von BugzillaMetrics erwartet wird, sowie Probleme beim Verbindungsaufbau zum XML-RPC Webservice sowie Datenbankfehler beim Zugriff des Webservice auf die Datenbestände von Bugzilla oder BugzillaMetrics.

Robustheit: Fehlerhafte Benutzereingaben werden, soweit dies möglich ist, bereits durch entsprechende Benutzerschnittstellen vermieden, indem dort nur plausible Werte(-kombinationen) zur Auswahl zugelassen sind. Sollten dennoch fehlerhafte oder inkonsistente Eingaben erfolgt sein, werden diese bei der Auswertung abgefangen. Dem Benutzer wird dies durch entsprechenden Dialogboxen mitgeteilt.

Ein weiterer Aspekt, der zur Robustheit der Anwendung beiträgt, ist die Möglichkeit, Qualitätsmodelle mithilfe der in Abschnitt 5.2.1 beschriebenen OCL-Regeln zu validieren. Dadurch hat der Benutzer die Möglichkeit, bereits während der Bearbeitung eines Modells zu überprüfen, ob es potentiell zu Problemen bei Auswertungen mit diesem Modell kommen kann.

Wiederherstellbarkeit: Ist beispielsweise ein Messwerkzeug zum Zeitpunkt einer Auswertung nicht erreichbar, kann die Auswertung zu einem späteren Zeitpunkt erneut durchgeführt werden. Auch hier erfolgt eine Benachrichtigung des Benutzers über entsprechende Dialoge.

6.1.3 Benutzbarkeit

Verständlichkeit: Die Aufteilung in die Werkzeuge Qualitätsmodell-Editor und Auswertungswerkzeug fördert die Verständlichkeit, da hierdurch die Verwendungszwecke klar abgetrennt sind. Zusätzlich wird die Verständlichkeit dadurch gefördert, dass sich die Repräsentation der Qualitätsmodelle orientiert am bekannten Konzept der Qualitätenbäume nach Boehm [BBK⁺78].

Erlern- und Bedienbarkeit: Die intuitive grafische Darstellung der Qualitätsmodelle ist positiv im Sinne der Erlernbarkeit, da es sich dabei um ein gut verständliches und den Benutzern bekanntes Konzept handelt. Nach [JLMS03] tragen auch die Wizard-gestützten Konfigurationsdialoge zu einer einfachen Erlernbarkeit bei. Schließlich ist das Base Filter Widget (siehe Abschnitt 5.3.6.1) ein Abbild der entsprechenden Schnittstelle in BugzillaMetrics. Daher haben Benutzer, die diese Anwendung bereits kennen, auch

im Bezug darauf Vorteile bezüglich der Erlernbarkeit und des Verständnisses.

Attraktivität: Durch die enge Integration der Werkzeuge in die Eclipse Entwicklungsumgebung ist der Aufwand für das Starten der Werkzeuge gering, was einer Nutzung durch den Benutzer förderlich ist. Auf der anderen Seite empfinden Benutzer Eclipse unter Umständen als überfrachtet mit Benutzerelementen und Dialogen. In solchen Fällen ist denkbar, die Werkzeuge als eigenständige Programme, die außerhalb von Eclipse als sog. *Rich Client Platform*-Anwendungen [ML05] ausgeführt werden, bereitzustellen.

6.1.4 Effizient

Zeitverhalten: Maßgeblich bestimmt wird das Zeitverhalten des Auswertungswerkzeuges durch das Zeitverhalten der verwendeten externen Messwerkzeuge bzw. des XML-RPC-Webservice. Die Zeit, die im Verhältnis dazu direkt vom Auswertungswerkzeug für die Berechnungen benötigt wird, kann vernachlässigt werden. Eine parallele Ausführung der Messvorgänge für Qualitätsindikatoren kann das Verhalten weiter optimieren. Siehe hierzu auch Abschnitt 7.1. Beim Qualitätsmodell-Editor bezieht sich die Anforderung an gutes Zeitverhalten in erster Linie darauf, wie flüssig Änderungen an der graphischen Darstellung der Qualitätsmodelle durchgeführt werden können. Diesbezüglich wurden keine Probleme festgestellt.

Verbrauchsverhalten: Der zusätzliche Ressourcenverbrauch bei Verwendung der Werkzeuge ist zu vernachlässigen, da der Verbrauch im Wesentlichen durch Eclipse selbst dominiert wird. Eine Instanz von Eclipse ohne die beiden Werkzeuge verwendet unter Mac OS X unmittelbar nach dem Start 227 MB Arbeitsspeicher. Bei Hinzufügen der Plug-ins für beide Werkzeuge ergibt sich unter sonst gleichen Bedingungen nach dem Start keine nennenswerte Erhöhung. Bei der anschließenden Verwendung der durch die Werkzeuge bereitgestellten Funktionen erhöht sich der Speicherbedarf um etwa 30 MB.

6.1.5 Änderbarkeit

Analysierbarkeit und Stabilität: Diese Qualitätsmerkmale werden maßgeblich durch die Architektur der Anwendung und die daraus resultierende Implementierung bestimmt. Die Beschreibung der Architektur und der Verwendung von Entwurfsmustern sowie der Zerlegung in Komponenten (siehe

	Gesamt	\emptyset	Maximum
Programmzeilen (LOC)	1594		
<i>Weighted Methods per Class</i> (WMC)	261	6,402	37
<i>Number of Children</i> (NOC)	13	0,240	5
<i>Depth of Inheritance Tree</i> (DIT)		1,473	3

Tabelle 6.1: Metriken für die Logikkomponente des Auswertungswerkzeuges

	Gesamt	\emptyset	Maximum
Programmzeilen (LOC)	3789		
<i>Weighted Methods per Class</i> (WMC)	533	9,636	52
<i>Number of Children</i> (NOC)	28	0,5	5
<i>Depth of Inheritance Tree</i> (DIT)		2,764	6

Tabelle 6.2: Metriken für das Frontend des Auswertungswerkzeuges

Kapitel 5) legt bereits nahe, dass Analysier- und Änderbarkeit der Werkzeuge gut sind. Untermauert wird dies durch über den Quellcode erhobene Metriken, die in Tabelle 6.1 sowie 6.2 für das Auswertungswerkzeug wiedergegeben sind. Diese Daten wurden mithilfe des *Metrics*-Plug-in [Met] für Eclipse gemessen.

Eine Erfassung von Metriken für den Qualitätsmodell-Editor ist nicht sinnvoll, da es sich dabei um automatisch generierten Code handelt, der auf Quellcodeebene in der Regel nicht extensiv analysiert wird, da diese Arbeit bereits auf Modellebene erfolgt. Die Stabilität des automatisch generierten Codes ist außerdem unabhängig von den Ergebnissen einer Metrikanalyse als gut anzusehen. Analog wird für die Logikkomponente des Auswertungswerkzeuges nur der Teil des Codes analysiert, der nicht per EMF generiert worden ist. Eine Analyse des XML-RPC-Webservice macht keinen Sinn, da dieser aus nur ca. 300 Zeilen Programmcode besteht.

Die Metrik *Weighted Methods per Class* belegt, dass die Komplexität der Methoden und Klassen generell gering ist, was insbesondere die Analysierbarkeit fördert. Der im Vergleich höhere Wert bei der Frontend-Komponente ergibt sich in erster Linie aus dem Naturell des SWT-Rahmenwerks. Für beide Komponenten ergeben sich niedrige Werte für die Metriken *Number of Children* und *Depth of Inheritance Tree*. Dies bedeutet, dass die Fehleranfälligkeit aufgrund komplexer Vererbungsstrukturen im vorliegenden Fall gering ist.

Modifizierbarkeit: Für dieses Qualitätsmerkmal gelten die gleichen Argumente wie im Fall von Analysierbarkeit und Stabilität. Es wird des weiteren gefördert durch die Verwendung von Model-Driven Development, da hierdurch zukünftige Änderungen oftmals bereits im zugrunde liegenden Modell eingepflegt werden können. Im Kontext des Model-Driven Development ist jedoch darauf zu achten, dass individuelle Anpassungen am generierten Code so vorgenommen werden, dass sie späteren Codeänderungen, die bedingt sind durch Änderungen im Modell, nicht im Wege stehen. Dies wird im vorliegenden Fall beispielsweise dadurch realisiert, dass individuelle Programmlogik über das Observer-Entwurfsmuster [GHJV95] aus einer vollständig separaten Klasse eingebunden wird.

Testbarkeit: Durch die Verwendung des Test-Driven Development Prinzips ist die Testbarkeit der Werkzeuge generell gut. Eine Ausnahme bildet der im Rahmen des Model-Driven Development generierte Code, da dieser per Definition unter Voraussetzung der Korrektheit der Modelle korrekt ist und daher dafür keine Tests erstellt wurden.

6.1.6 Übertragbarkeit

Anpassbarkeit: Der Webservice ist als Servlet implementiert und kann daher in jedem Servlet-Container, der die Java Servlet Specification [JSS] unterstützt, ausgeführt werden. Die Werkzeuge liegen als Eclipse-Plug-ins vor und können in allen Versionen von Eclipse ab 3.4 (Ganymede) verwendet werden.

Installierbarkeit: Die Installation der Werkzeuge erfolgt durch Kopieren der entsprechenden JAR-Dateien, welche die einzelnen Plug-ins repräsentieren, in eine bestehende Eclipse-Installation. Dieser Prozess kann durch das Einrichten einer sog. Update-Site (siehe hierzu die Diskussion in Abschnitt 7.1) noch vereinfacht werden.

Koexistenz: Die Koexistenz mit anderen Werkzeugen ist möglich. Der XML-RPC-Webservice kann ohne Probleme aufgrund der Natur von HTTP parallel durch mehrere Anwendungen verwendet werden.

Austauschbarkeit: Durch die Verwendung von XML als Speicherformat können sowohl der Qualitätsmodell-Editor als auch das Auswertungswerkzeug gegen andere Anwendungen, die deren Funktionalität übernehmen, ein-

fach ausgetauscht werden. Darüber hinaus kann innerhalb des Auswertungswerkzeuges aufgrund der entsprechend ausgelegten Architektur das Frontend unter Beibehaltung der Logikkomponente gegen ein andersartiges Frontend wie beispielsweise ein Webinterface ausgetauscht werden.

Konformität: Der XML-RPC-Webservice ist konform zur Java Servlet Specification 2.5 sowie dem XML-RPC-Standard. Alle Komponenten sind konform zur Java 5.0 Umgebung und können daher auf allen Systemen, die diese Plattform und Eclipse ab Version 3.4 unterstützen, eingesetzt werden.

6.2 Qualität des Entwicklungsprozesses

Nach der Betrachtung der Produkt-Qualität folgt nun eine Untersuchung über die Qualität des verwendeten Softwareentwicklungsprozesses.

Eine Bewertung auf Basis eines bidirektionalen Qualitätsmodells unter Verwendung der entwickelten Werkzeuge selbst ist leider nicht möglich, da im vorliegenden Fall keine Datenbasis in Form von Change Requests existiert. Es handelt sich dabei um eine für Neuentwicklungen oft anzutreffende Situation. Etablierte Bewertungsmodelle wie CMMI oder SPICE [LL07, Kapitel 11] sind für den verwendeten Prozess ebenfalls ungeeignet, da sie primär für mittlere und große Softwareprojekte mit mehreren Entwicklern ausgelegt sind. Im Gegensatz dazu handelt es sich im vorliegenden Fall um ein im Verhältnis kleines Projekt mit einem einzelnen Entwickler.

Aus diesem Grund kann nur eine weniger formale Prozessbewertung erfolgen. Im Folgenden beschränkt sich die Untersuchung auf die Prozessmerkmale, die als problematisch anzusehen sind. Sommerville [Som04, S. 667] schlägt für die Bewertung sog. “informal processes” eine Reihe von Kriterien vor. Dazu gehört unter anderem, dass Prozesse **Robustheit** bezüglich unvorhergesehener Probleme während der Softwareentwicklung aufweisen.

Der hier gewählte Prozess kann einerseits als robust angesehen werden, da mit dem Schwellenwert-Filter eine neue Anforderung erst im letzten Drittel der Implementierungsphase erkannt wurde und trotzdem innerhalb kurzer Zeit umgesetzt werden konnte. Gleiches gilt für das Base Filter Widget. Auch die Tatsache, dass zu Beginn der Implementierung die endgültige Gestalt des Auswertungs-Frontends noch nicht abschließend klar war, hat zu keinen nennenswerten Problemen oder Verzögerungen geführt.

Andererseits deuten die gerade beschriebenen Punkte jedoch darauf hin, dass der Prozess die Anforderungsanalyse nur unzureichend unterstützt hat, da dort offensichtlich nicht alle Anforderungen ausreichend gut analysiert worden sind.

Kapitel 7

Fazit & Ausblick

Das Ziel des bis hierher beschriebenen Softwareprojektes war die Erstellung von Werkzeugen zur Arbeit mit Qualitätsmodellen für Softwareentwicklungsprozesse und zur Anwendung dieser Qualitätsmodelle auf Softwareprojekte.

Das Projekt begann mit einer Einarbeitung in die Materie der (bidirektionalen) Qualitätsmodelle sowie einer Analyse, wie diese um Datenstrukturen zur Umsetzung der Anforderungen der Qualitätsbewertung mittels des Werkzeugs QMetric erweitert werden können. Daran an schloss eine erste Analyse der funktionalen und nichtfunktionalen Anforderungen. Hieraus ergaben sich nicht nur wertvolle Informationen über den notwendigen Funktionsumfang, sondern es wurde auch das grobe Vorgehen für den Rest des Projektablaufs deutlich.

Dieser grobe Plan wurde sodann im Rahmen eines konkreten Zeitplans detailliert. Daran an schloss sich eine Analyse der Rahmenbedingungen und Einschränkungen, unter denen das Projekt ablaufen werde. Die Ergebnisse dieser Analyse legten den Grundstein für die Entscheidung, einen agilen Softwareentwicklungsprozess mit einem iterativen Vorgehen während der Implementierung zu verwenden.

Nach einer detaillierteren Anforderungsanalyse begann mit der Implementierungsphase die konkrete Umsetzung der Werkzeuge. Erster Bestandteil dieses Abschnitts war das Erstellen eines einfachen Prototypen. Dieser setzte einige grundlegende Elemente der Anforderungen um, diente primär jedoch dazu, mit den technischen Gegebenheiten wie GMF und Model-Driven Development vertraut zu werden. Die Entscheidung für die Erstellung des

Prototypen hat sich im Nachhinein als richtig erwiesen: Obwohl es sich dabei um einen reinen Wegwerf-Prototypen handelte, konnte dabei erste Praxiserfahrung mit den unbekanntem Technologien gesammelt werden und es wurden mögliche technische Problembereiche für den weiteren Verlauf der Implementierung deutlich.

Der Entwurf und die Realisierung der beiden eigentlichen Werkzeuge erfolgte dann auf iterativem Wege; die Dokumentation der Architektur fand parallel dazu statt.

In Ergänzung zur Beschreibung der Anforderungen und der Dokumentation der Architektur wurde der gewählte Softwareentwicklungsprozess ebenfalls dokumentiert und es wurde eine Evaluierung der entstandenen Werkzeuge durchgeführt. Abgeschlossen wurde der Entwicklungsprozess durch die Erstellung eines kurzen *User Guide*, in dem die grundlegenden Funktionen der Werkzeuge erläutert werden.

Das Ergebnis der Entwicklung liegt vor in Form zweier Plug-ins für die Entwicklungsumgebung Eclipse: Der Qualitätsmodell-Editor ermöglicht das Anlegen und Bearbeiten von Qualitätsmodellen mithilfe einer auf Graphen basierenden Benutzerschnittstelle. Das Auswertungswerkzeug erlaubt es, auf Basis von Qualitätsmodellen unter Verwendung externer Messwerkzeuge (derzeit beschränkt auf BugzillaMetrics) die Softwareprozessqualität gegebener Projekte zu bewerten und auf verschiedene Arten zu visualisieren.

Eine praktische Anwendung auf einen in der Praxis stattfindenden Softwareentwicklungsprozess ist noch nicht erfolgt. Trotzdem deutet der erste Eindruck nach Fertigstellung der initialen Version darauf hin, dass die Werkzeuge sich gut in den Arbeitsablauf von Unternehmen integrieren lassen und ein intuitives und flexibles System zur Bewertung der Qualität von Softwareentwicklungsprozessen bereitstellen.

7.1 Ausblick

Wie bei den meisten Software-Anwendungen, die initial in einer ersten Version vorliegen, bieten sich auch beim Qualitätsmodell-Editor und dem Auswertungswerkzeug Ansatzpunkte für zukünftige Erweiterungen und Verbesserungen an. Einige werden nachfolgend vorgestellt.

- Durch die **parallele Ausführung der Messvorgänge** für Qualitätsin-

dikatoren in Threads kann es zu einer Verbesserung der Performance kommen. Dies liegt daran, dass dann parallel zu einem Messvorgang, der viel Zeit in Anspruch nimmt, bereits andere Messungen durchgeführt werden können.

Demgegenüber steht jedoch beispielsweise bei BugzillaMetrics das Problem, dass es bei Verwendung der gleichen Installation für mehrere Indikatoren dort zu Einbußen durch *Swapping*¹ kommen kann, wenn mehrere aufwändige Messungen parallel ausgeführt werden. Daher muss vor der Implementierung des Threadings geprüft werden, ob dies zu einer Performance-Verbesserung führt oder kontraproduktiv ist.

- Das Auswertungswerkzeug kann um **alternative Darstellungsformen für die Ergebnisse** erweitert werden. Denkbar ist beispielsweise das Auftragen der Messergebnisse mehrerer Qualitätsknoten in einem gemeinsamen Netzdiagramm (auch: Kiviat- oder Radar-Diagramm).

Diese Funktionalität kann allerdings schon mit dem jetzigen Stand der Werkzeuge mithilfe des CSV-Exports erzielt werden, in dem die CSV-Daten in Microsoft Excel oder einer anderen Anwendung zur Tabellenkalkulation eingelesen und dort entsprechend visualisiert werden.

- Aus der praktischen Anwendung der Werkzeuge wird sich ergeben, ob Bedarf für **weitere Filterfunktionen und Berechnungsvorschriften** existiert. Dass eine Erweiterung um neue Filter mit vertretbarem Aufwand möglich ist, zeigte sich bereits während der Implementierungsphase, als der Schwellenwert-Filter nachträglich mit nur wenigen Stunden Zeitaufwand integriert wurde.
- Eine Erweiterung der Anwendung um **weitere Messwerkzeuge** neben BugzillaMetrics ist denkbar. In diesem Zusammenhang kann darüber nachgedacht werden, die Integration der Messwerkzeuge über das Konzept der sog. *Extension Points* [HS], das durch das Plug-in Development Environment (PDE) von Eclipse zur Verfügung gestellt wird, zu realisieren.
- Um eine einfachere Installation und Aktualisierung der Werkzeuge zu ermöglichen, kann der **Update-Site-Mechanismus** [GB03, S. 86] der Eclipse Plug-in Plattform Verwendung finden. Damit ist es möglich,

¹Swapping bezeichnet den Vorgang, bei dem Daten aus Kapazitätsgründen aus dem schnellen Primärspeicher (RAM) in den langsameren Sekundärspeicher (Festplatte) verschoben oder von dort geladen werden.

Neuinstallationen und Updates über ein HTTP-basiertes System mit automatischer Auflösung von Abhängigkeiten und einer grafischen Benutzerschnittstelle durchzuführen.

- Langfristig bleibt zu prüfen, ob das Werkzeug auch für die **Modellierung und Bewertung von Qualität in anderen Bereichen** außerhalb der Softwareentwicklung eingesetzt werden kann. In diesem Fall würde sich die im Titel dieser Diplomarbeit angekündigte *Generizität* nicht nur auf die Möglichkeit zur generischen Bewertung einer Vielzahl verschiedener Softwareentwicklungsprozesse beziehen. Es würde sich nämlich bei den Werkzeugen sodann um eine generische Lösung zum Einsatz in vielen verschiedenen Anwendungsgebieten handeln.

Entsprechende Messwerkzeuge zur Anbindung an Qualitätsindikatoren vorausgesetzt, wäre es dann beispielsweise möglich, auch Gewässerqualitäten oder andere Umweltfaktoren zu bewerten, sofern deren Qualitätsmerkmale in bidirektionalen Qualitätsmodellen ausgedrückt werden können.

Zusammenfassend kann gesagt werden, dass mit den vorliegenden Werkzeugen zwei Anwendungen geschaffen wurden, die einerseits eine intuitive Methode zur Formulierung und Bearbeitung von Qualitätsmodellen und andererseits eine flexible und erweiterbare Auswertungsschnittstelle für die Bewertung von Softwareprozessqualität bereitstellen. Die Anwendungen sind in der Praxis verwendbar und können wertvolle Rückschlüsse auf den Stand der Dinge bezüglich der erreichten Prozessqualität und sich daraus ergebender Verbesserungsmaßnahmen liefern.

Literaturverzeichnis

- [BBK⁺78] BOEHM, Barry W. ; BROWN, John R. ; KASPAR, Hans ; LIPOW, Myron ; MACLEOD, Gordon J. ; MERRIT, Michael J.: *Characteristics of Software Quality*. North-Holland Publishing Company, 1978 (TRW Series of Software Technology). – ISBN 0-444-85105-4
- [BF01] BECK, Kent ; FOWLER, Martin: *Planning Extreme Programming*. Addison-Wesley, 2001. – ISBN 3-8273-1832-7
- [BSM⁺03] BUDINSKY, Frank ; STEINBERG, Dave ; MERKS, Ed ; ELLERSICK, Ray ; GROSE, Timothy J.: *Eclipse Modeling Framework*. Addison-Wesley, 2003. – ISBN 0-13-142542-0
- [Buga] *Bugzilla*. <http://www.bugzilla.org/>. – [Online; Stand 14. Oktober 2008]
- [Bugb] *BugzillaMetrics*. <http://www.bugzillametrics.org/>. – [Online; Stand 28. Juni 2008]
- [CKS06] CHRISISS, Mary B. ; KONRAD, Mike ; SHRUM, Sandy: *CMMI. Guidelines for Process Integration and Product Improvement*. 2. Addison-Wesley, 2006. – ISBN 0-321-27967-0
- [CM78] CAVANO, Joseph P. ; MCCALL, James A.: A framework for the measurement of software quality. In: *Proceedings of the software quality assurance workshop on functional and performance issues*, 1978, S. 133–139
- [CVW98] COOK, Jonathan E. ; VOTTA, Lawrence G. ; WOLF, Alexander L.: Cost-Effective Analysis of In-Place Software Processes. In: *IEEE Transactions on Software Engineering* 24 (1998), Aug, Nr. 8, S. 650–663. – ISSN 0098-5589

- [Ecl] *Eclipse*. <http://www.eclipse.org/>. – [Online; Stand 26. August 2008]
- [EMF] *Eclipse Modeling Framework (EMF)*. <http://www.eclipse.org/emf/>. – [Online; Stand 26. August 2008]
- [Fow03] FOWLER, Martin: *UML Distilled*. Addison-Wesley, 2003. – ISBN 0-321-19368-7
- [FP97] FENTON, Norman ; PFLEEGER, Shari L.: *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., 1997. – ISBN 0-5349-5600-9
- [GB03] GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse*. Addison-Wesley, 2003. – ISBN 0-321-20575-8
- [GEF] *Eclipse Graphical Editing Framework (GEF)*. <http://www.eclipse.org/gef/>. – [Online; Stand 26. August 2008]
- [Geo04] GEORGII, Hans-Otto: *Stochastik - Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Walter de Gruyter GmbH & Co. KG, 2004. – ISBN 3-11-018283-3
- [GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. – ISBN 0-201-63361-2
- [GMF] *Eclipse Graphical Modeling Framework (GMF)*. <http://www.eclipse.org/gmf/>. – [Online; Stand 26. August 2008]
- [Gra07] GRAMMEL, Lars: *Development of a Tool for the Evaluation of Change Requests*, RWTH Aachen, Diplomarbeit, Februar 2007
- [HS] HENNING, Manfred ; SEEBERGER, Heiko: Einführung in den Extension Point-Mechanismus von Eclipse. In: *JavaSPEKTRUM* 01/2008, S. 19-24
- [IEE90] IEEE standard glossary of software engineering terminology. In: *IEEE Std 610.12* (1990), Dec
- [IEE98] IEEE standard for a software quality metrics methodology. In: *IEEE Std 1061-1998* (1998), Dec
- [Jav] *Java*. <http://java.sun.com/>. – [Online; Stand 24. November 2008]

- [JKC04] JUNG, Ho-Won ; KIM, Seung-Gweon ; CHUNG, Chang-Shin: Measuring software product quality: a survey of ISO/IEC 9126. In: *Software, IEEE* 21 (2004), Sept.-Oct., Nr. 5, S. 88–92. – ISSN 0740–7459
- [JLMS03] JURISTO, Natalia ; LOPEZ, Marta ; MORENO, Ana M. ; SÁNCHEZ, M. I.: Improving software usability through architectural patterns. In: *ICSE Workshop on SE-HCI*, 2003, S. 12–19
- [JSS] *Java Servlet Specification 2.5*. <http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html>. – [Online; Stand 2. Januar 2009]
- [JUn] *JUnit*. <http://junit.org/>. – [Online; Stand 19. November 2008]
- [LL07] LICHTER, Horst ; LUDEWIG, Jochen: *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag GmbH, 2007. – ISBN 3–89864–268–2
- [Man] *Mantis Bug Tracker*. <http://www.mantisbt.org/>. – [Online; Stand 14. Oktober 2008]
- [Met] *Metrics 1.3.6*. <http://metrics.sourceforge.net/>. – [Online; Stand 21. Januar 2009]
- [ML05] MCAFFER, Jeff ; LEMIEUX, Jean-Michel: *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, 2005. – ISBN 0–321–33461–2
- [Oba07] OBAID, Malek: *Development of Query and Reporting Interface for a Change Request Analysis Tool*, RWTH Aachen, Diplomarbeit, Dezember 2007
- [PDE] *Eclipse Plug-in Development Environment (PDE)*. <http://www.eclipse.org/pde/>. – [Online; Stand 26. August 2008]
- [Pre97] PREE, Wolfgang: *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt.verlag, 1997. – ISBN 3–92099368–3
- [RMI] *Remote Method Invocation*. <http://java.sun.com/products/jdk/rmi/>. – [Online; Stand 11. November 2008]

- [Sch09] SCHÄFER, Henning: *Bewertung der Prozessqualität von Open-Source-Entwicklungsprojekten auf Basis von Software-Prozessdaten*, RWTH Aachen, Diplomarbeit, 2009
- [SJD01] ST.LAURENT, Simon ; JOHNSTON, Joe ; DUMBILL, Edd: *Programming Web Services with XML-RPC*. 1. O'Reilly & Associates, Inc., 2001. – ISBN 0-596-00119-3
- [SLLJ09] SCHACKMANN, Holger ; LICHTER, Horst ; LISCHKOWITZ, Christoph ; JANSEN, Martin: QMetric - A Metric Tool Suite for the Evaluation of Software Process Data. In: *ICSE 2009*, 2009
- [Som04] SOMMERVILLE, Ian: *Software Engineering*. 7. Pearson Education Limited, 2004. – ISBN 0-321-21026-3
- [SSM06] SIMON, Frank ; SENG, Olaf ; MOHAUPT, Thomas: *Code-Quality-Management*. dpunkt.verlag, 2006. – ISBN 3-89864-388-3
- [Sta73] STACHOWIAK, Herbert: *Allgemeine Modelltheorie*. Springer-Verlag, 1973. – ISBN 3211811060
- [Ste02] STEEGER, Angelika: *Diskrete Strukturen Band 1 - Kombinatorik, Graphentheorie, Algebra*. Springer-Verlag, 2002. – ISBN 3-540-67597-3
- [Sub] *Subversion*. <http://subversion.tigris.org/>. – [Online; Stand 5. September 2008]
- [SWT] *SWT: Standard Widget Toolkit*. <http://www.eclipse.org/swt/>. – [Online; Stand 25. November 2008]
- [Tha94] THALLER, Georg E.: *Software-Metriken einsetzen, bewerten, messen*. Verlag Heinz Heise GmbH & Co KG, 1994. – ISBN 3-88229-038-2
- [Tom] *Apache Tomcat*. <http://tomcat.apache.org/>. – [Online; Stand 11. November 2008]
- [TY97] THAYER, R. H. ; YOURDON, E.: *Software Engineering Project Management*. John Wiley & Sons, 1997. – 552 S.
- [XMLa] *Apache XML-RPC*. <http://ws.apache.org/xmlrpc/xmlrpc2/>. – [Online; Stand 11. November 2008]

- [XMLb] *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. <http://www.w3.org/TR/xml/>. – [Online; Stand 11. November 2008]