Ableitung von rollenspezifischen Sichten auf projektbegleitend gesammelten Daten

IRINA FEUERBACH

26. September 2003

AG Software Engineering
Fachbereich Informatik
Universität Kaiserslautern

Betreuer: Prof. Dr. H. Dieter Rombach
Diplom.-Inform. Jens Heidrich

Erklärung

Hiermit erkläre ich, Irina Feuerbach, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kaiserslautern, den 26. September 2003

Zusammenfassung

Um die wachsende Komplexität von Software-Entwicklungsprojekten und die wachsenden Anforderungen an die Funktionalität des zu erstellenden Systems gerecht zu werden, ist es notwendig, Mechanismen zur Steuerung und Kontrolle der Projekte einzusetzen Hierzu müssen projektbegleitend Messdaten z.B. mit Hilfe des GQM-Ansatzes gesammelt, entsprechend verarbeitet und zielgerichtet visualisiert werden. Diese Aufgabe wird vom Software Project Control Center (*SPCC*) unterstützt, welcher Funktionen für die Verarbeitung von Messdaten und Sichten zur Visualisierung bereitstellt.

An einem Software-Entwicklungsprojekt sind eine Vielzahl von unterschiedlichen Rollen beteiligt, welche unterschiedliche Aufgaben zu erfüllen haben und an unterschiedlichen Messdaten interessiert sind. Damit müssen für die unterschiedlichen Rollen verschiedene maßgeschneiderte Sichten bereitgestellt werden.

Die Ziele dieser Arbeit liegen darin, für die unterschiedlichen Rollen eine Menge von Sichten zu identifizieren und zu definieren. Dies umfasst sowohl eine Auswahl von Messdaten, welche für die Rolle von Interesse sind, und welchen Nutzen diese aus den Daten ziehen kann, als auch eine Darstellung der Daten derart, dass der Nutzen, den man daraus erzielen kann, optimiert wird.

Die hier implementierten Sichten können von verschiedenen Rollen bei der Durchführung ihrer Arbeit benutzt werden. So kann der Projektplaner die Sichten verwenden, um einen Projektplan zu erstellen bzw. anzupassen. Der Projektmanager kann durch die Verwendung der Sichten nicht nur die Einhaltung des Projektplans überwachen, sondern auch feststellen, ob andere Werkzeuge und Methoden eingesetzt werden müssen, um den Prozess bei zukünftigen Projekten zu verbessern. Der Qualitätssicherer kann begründete Aussagen über die Qualität der resultierenden Produkte und des Projektes machen und die Einhaltung von vorgegebenen Qualitätszielen überprüfen. Auch der Entwickler kann die Sichten nutzen, da er dadurch leichter entscheiden kann, in welche Aufgaben er seine Zeit investieren muss.

Im einzelnen werden in dieser Arbeit existierende Sichten aus der Literatur hinsichtlich ihres Ansatzes, der verwendeten Darstellungsarten und der verwendeten Metriken untersucht. Außerdem wird eine Menge von Anforderungen, die an Sichten gestellt werden, abgeleitet und deren Einhaltung für die Sichten der Literatur untersucht. Weiterhin wird ein allgemeines Metamodell für Sichten entwickelt. Am Ende werden Sichten, die den Anforderungen entsprechen, abgeleitet, implementiert und anschließend evaluiert.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Einführung SPCC . . . 2
1.3 Das SPCC Tool . . . 5
1.4 Aufgabenstellung . . . 8
1.5 Übersicht . . . 8
1.6 Metriken . . . 9
1.7 GQM (*Goal Question Metric*) . . . 9
1.8 Diagrammtypen . . . 10
1.8.1 Liniendiagramm . . . 11
1.8.2 Balkendiagramm . . . 12
1.8.3 Kreisdiagramm . . . 14
1.8.4 Matrix . . . 15
1.8.5 Netzwerk . . . 17
1.8.6 Tabelle . . . 18
1.8.7 Zeitplan . . . 19
1.8.8 Box-Plots . . . 20

2 Literaturrecherche **23**
2.1 Anmerkungen zur Recherche . . . 23
2.2 Gefundene Literaturquellen . . . 23
2.2.1 Software Measurement Guidebook . . . 24
2.2.2 Collecting and Analyzing Web-Based Project Metrics . . . 27
2.2.3 A Practical View of Software Measurement and Implementation Experiences Within Motorola . . . 28
2.2.4 Visualizing Software Changes . . . 31
2.2.5 Software Management Environment (SME) - Components and Algorithms . . . 33
2.3 Zusammenfassung der Recherche-Ergebnisse . . . 35

3 Sichten-Metamodell **37**
3.1 Anforderungen an Sichten . . . 37
3.1.1 Übersichtlichkeit . . . 37
3.1.2 Verteilung . . . 37
3.1.3 Vergleichsmöglichkeit . . . 38
3.1.4 Hervorhebung . . . 38
3.1.5 Skalierung . . . 39
3.1.6 Robustheit . . . 39
3.1.7 Konsistenz . . . 39
3.1.8 Zielorientierte Darstellung . . . 39
3.1.9 Variabilität/Anpassbarkeit . . . 40
3.2 Klassifikation von Sichten . . . 40
3.3 Meta-Modell . . . 42
3.4 Bewertung der gefundenen Sichten . . . 43
3.4.1 Bewertung der Sichten: „Software Measurement Guidebook” 43

3.4.2 Bewertung der Sichten: „Collecting and Analyzing Web-Based Project Metrics" . . . 44
3.4.3 Bewertung der Sichten „A Practical View of Software Measurement and Implementation Experiences Within Motorola" . 44
3.4.4 Bewertung der Sichten: „Visualizing Software Changes" . . . 45
3.4.5 Bewertung der Sichten: „Software Management Environment (SME) - Components and Algorithms" . . . 46

4 Herleitung möglicher Sichten 49
4.1 Rollen . . . 49
4.2 Ausgewählte Metriken . . . 51
4.2.1 Zeit . . . 51
4.2.2 Aufwand . . . 51
4.2.3 Fehler . . . 52
4.2.4 Änderungen . . . 53
4.2.5 Priorität . . . 54
4.2.6 Kode-Metriken . . . 54
4.2.7 Kosten . . . 54
4.3 Resultierende Sichten . . . 55
4.3.1 Aufwandsverteilung auf Phasen bzw. Aktivitäten . . . 55
4.3.2 Zeitlicher Projektverlauf . . . 55
4.3.3 Fehlerverteilung auf Phasen und Affiliation und zugehörigem Aufwand . . . 56
4.3.4 Fehlerverteilung auf Fehlerklassen . . . 56
4.3.5 Fehlerverteilung nach Status . . . 57
4.3.6 Fehlerverteilung nach Status und Fehlerklasse . . . 57
4.3.7 Auflistung der Anforderungen: . . . 58
4.3.8 Ort der Änderungen . . . 58
4.3.9 Kategorie der Änderungen . . . 58
4.3.10 Anzahl der Änderungen pro Phase . . . 59
4.3.11 Aufwand für die Änderungen pro Phase . . . 59
4.3.12 Änderungen und damit verbundener Aufwand pro Phase . . . 59
4.3.13 Prozentuales Verhältnis zwischen Kodezeilen und Dokumentationszeilen . . . 60
4.3.14 Aufwand für Tests pro Klasse . . . 60
4.3.15 Kostenvergleich pro Phase . . . 61
4.3.16 Verteilung der Gesamtkosten . . . 61

5 Implementierung und Evaluation 63
5.1 Einleitung . . . 63
5.2 Implementierung der Sichten . . . 66
5.2.1 Aufwandsverteilung auf Phasen (*Phases Line Plot*) . . . 66
5.2.2 Aufwandsverteilung auf Phasen und Aktivitäten (*Phases Line Plot With Activities*) . . . 69
5.2.3 Fehlerverteilung auf Phasen und Affiliation mit zugehörigem Aufwand (*Defect View*) . . . 73
5.2.4 Zeitlicher Projektverlauf (*Project Plan*) . . . 75
5.2.5 Kreisdiagramm (*Pie Chart*) . . . 79

5.3 Bewertung der Sichten 80
5.4 Einsatz der Sichten 84

6 Zusammenfassung und Ausblick **85**
6.1 Überblick 85
6.2 Ausblick 86

A Sichten **95**
A.1 „Software measurement guidebook" (aus [2]) 95
A.2 „Collecting and Analyzing Web-Based Project Metrics" (aus [5]) . . 98
A.3 „A Practical View of Software Measurement and Implementation Experiences Within Motorola" (aus [1]) 99
A.4 „Visualizing Software Changes" (aus [4]) 102
A.5 „Software Management Environment (SME) - Components and Algorithms" (aus [3]) 106

B Implementierung der Sichten **108**
B.1 Aufwandsverteilung auf Phasen (*Phases Line Plot*) 108
B.2 Aufwandsverteilung auf Phasen und Aktivitäten (*Phases Line Plot With Activities*) 114
B.3 Fehlerverteilung auf Phasen und Affiliation mit zugehörigem Aufwand (*Defect View*) 129
B.4 Zeitlicher Projektverlauf (*Project Plan*) 133
B.5 Kreisdiagramm (*Pie Chart*) 137

C Implementierte Funktionen **141**
C.1 Convert Project Plan 2 List of Time Series 141
C.2 Defect Processing 142
C.3 Defect Sorting (Affiliation and Phase) 144
C.4 Effort Processing 145
C.5 Reported Effort Processing 147

1 Einleitung

1.1 Motivation

Durch die stärker wachsenden Anforderungen sowohl an die Funktionalität, als auch an die Qualität eines Softwaresystems, nimmt die Komplexität eines Software-Entwicklungsprojektes zu. Dies führt unter anderem dazu, dass häufig die gesetzten Pläne in Bezug auf Budget und Zeitrahmen für die Produkt-Fertigstellung nicht eingehalten werden können und die produzierte Qualität der Software relativ schlecht ist [11]. Die Ursachen hierfür sind vielfältig und können einerseits durch mangelnde Kommunikation zwischen Entwickler und Kunden zur Klärung von Anforderungen verursacht werden und andererseits durch mangelnde Kontrolle über das Software-Entwicklungsprojekt. Um dies zu bewältigen, ist es notwendig, ingenieurmäßig bei der Entwicklung vorzugehen [9]. Dies bedeutet, dass zuerst der Prozess geplant und ein Messplan erstellt wird. Bei der Durchführung des Projektes werden begleitend Messdaten gesammelt. Schließlich geht das messbasierte Feedback in eine Erfahrungsdatenbank ein, um zukünftig Verbesserungen erreichen zu können (Erfahrungsmanagement).

An diesem Punkt setzt der Software Projekt Control Center (*SPCC*, siehe auch 1.2) an. Der SPCC ist ein Mittel zur Interpretation und Visualisierung von projektbegleitend gesammelten Messdaten. Die Aufgabe des SPCC liegt darin, den beteiligten Personen die Daten zielgerichtet darzustellen, so dass diese bei der Durchführung ihrer Aufgabe unterstützt werden. Hierzu werden Messdaten des Projekts benötigt, welche in verschiedene Funktionen eingehen und dort verarbeitet werden. Die Funktionen übernehmen hierbei die Interpretation der Daten, wobei sowohl die Messdaten in Funktionen eingehen können, als auch bereits durch Funktionen verarbeitete Daten. Außerdem sollen Diagramme erzeugt werden, die die interpretierten Daten und Messdaten visualisieren. Dies wird im SPCC durch Sichten realisiert. Für diese muss jeweils eine Auswahl getroffen werden, welcher Sachverhalt dargestellt werden soll und welche Daten (Messdaten, interpretierte Daten) hierfür benötigt werden (siehe auch 1.2).

Der SPCC stellte bisher einen allgemeinen Rahmen zur zielgerichteten Visualisierung von Messdaten bereit ohne bereits eine Menge von Sichten zu definieren, welche die an einem Projekt beteiligten Rollen bei ihrer Arbeit unterstützen. Dies soll nun nachgeholt werden. Daher liegt das Ziel dieser Arbeit darin, für die verschiedenen beteiligten Rollen geeignete Sichten zu identifizieren und zu definieren. Dies umfasst nicht nur die Analyse, welche Daten für welche Rollen von Bedeutung sind, und welchen Nutzen daraus entstehen kann, sondern auch die geeignete Visualisierung, damit der Nutzen optimiert wird.

Die implementierten Sichten helfen dem Entwickler bei seiner Arbeit, indem er beispielsweise feststellen kann, wieviele gefundene Fehler noch zu beheben sind, oder bereits korrigiert wurden (in welchem Status sich folglich einzelne, dokumentierte Fehler befinden). Der Qualitätssicherer dagegen kann Aussagen über die Qualität der Produkte und des Prozesses machen sowie die Einhaltung von gegebenen Qualitätszielen kontrollieren. Der Projektplaner hingegen wird bei der Erstellung und Anpassung von Projektplänen unterstützt, deren Einhaltung der Projektmanager wiederum mit Hilfe der Sichten kontrollieren kann. Außerdem kann dieser auch Entscheidungen darüber treffen, ob andere Werkzeuge und Methoden verwendet werden müssen, um die gesteckten Projektziele einzuhalten.

1.2 Einführung SPCC

Bei dem Software Project Control Center (*SPCC*) ist eine Art Leitstand oder Kontrollzentrum. Mit dessen Hilfe kann die Einhaltung von vorgegebenen Zielen, z.B. Kosten, überwacht werden [12]. Der SPCC interpretiert die Daten, welche während der Software-Entwicklung anfallen und visualisiert diese angemessen, um den Benutzer bei seiner Arbeit zu unterstützten. Letzteres bedeutet unter anderem, dass die Visualisierung der Daten sowohl rollenorientiert und als auch zweckorientiert stattfindet.

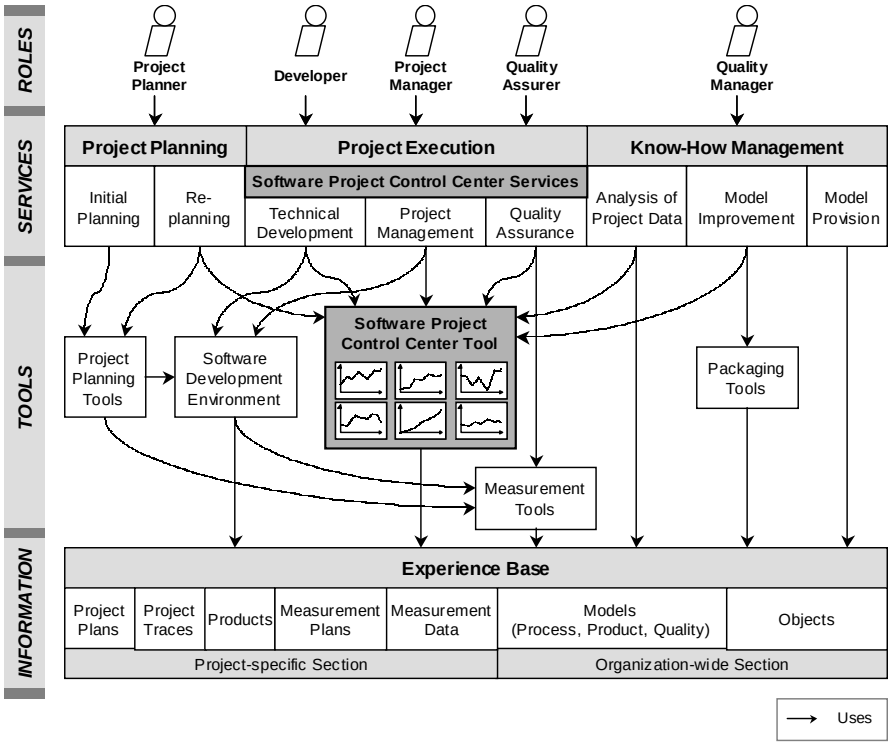


Abbildung 1: Einordnung des SPCCs in eine Software-Entwicklungsumgebung (aus [11]).

Abbildung 1 zeigt, wie der SPCC in eine Software-Entwicklungsumgebung integriert werden kann. Hierbei werden vier verschiedene Ebenen unterschieden, welche teilweise aufeinander aufbauen. Diese Ebenen sind: Rollen, Dienste, Werkzeuge und Informationen, wobei der SPCC in zwei Ebenen einzuordnen ist: Dienste und Werkzeuge. Die an einem Software-Entwicklungsprojekt beteiligten Rollen benutzen verschiedene Dienste in verschiedenen Phasen, um mit Hilfe dieser, ihre Aufgaben zu erfüllen. So benutzt beispielsweise der Projektmanager verschiedene Managementdienste, um das Projekt zu kontrollieren. Ein solcher Dienst kann in drei Kategorien eingeteilt werden: Planung, Ausführung und Erfahrungsmanagement. Für jede

dieser Kategorien gibt es mehrere Dienste, die zum Teil verschiedene Werkzeuge benutzen, um den Dienst auszuführen. Da der SPCC im Wesentlichen zur Kontrolle gedacht ist, werden seine Dienste hauptsächlich während der Projekt-Ausführung benutzt. Die verschiedenen Werkzeuge verwenden Informationen aus einer Erfahrungsdatenbank einerseits und Informationen von anderen Werkzeugen andererseits. In diese Ebene gehört auch das SPCC-Werkzeug und unterstützt den Projektmanager, Qualitätssicherer usw., indem er die Daten des Prozesses verarbeitet und an die jeweiligen Bedürfnisse angepasst visualisiert. Ein SPCC ist ein Mittel zur prozessbegleitenden Interpretation und Visualisierung von Messdaten und besteht aus drei Bestandteilen [11]. Diese sind die logische Architektur, das Werkzeug, welches die Architektur implementiert, und das Repository.

Die letzte Ebene umfasst die Erfahrungsdatenbank, in welcher Daten abgelegt werden, die zur Durchführung eines Software-Entwicklungsprojekts benötigt werden. Diese Daten können sowohl von Werkzeugen, als auch direkt von Diensten benutzt werden.

Nachdem der SPCC innerhalb einer Software-Entwicklungsumgebung eingeordnet wurde, wird nun genauer auf die einzelnen Komponenten des SPCCs eingegangen. Da ein SPCC Daten benötigt, müssen diese zuerst bereitgestellt werden. Zur Identifizierung der benötigten Daten, wird der GQM-Ansatz verwendet (siehe 1.7), um aus einer Menge von Zielen die geeigneten Metriken oder Maße abzuleiten.

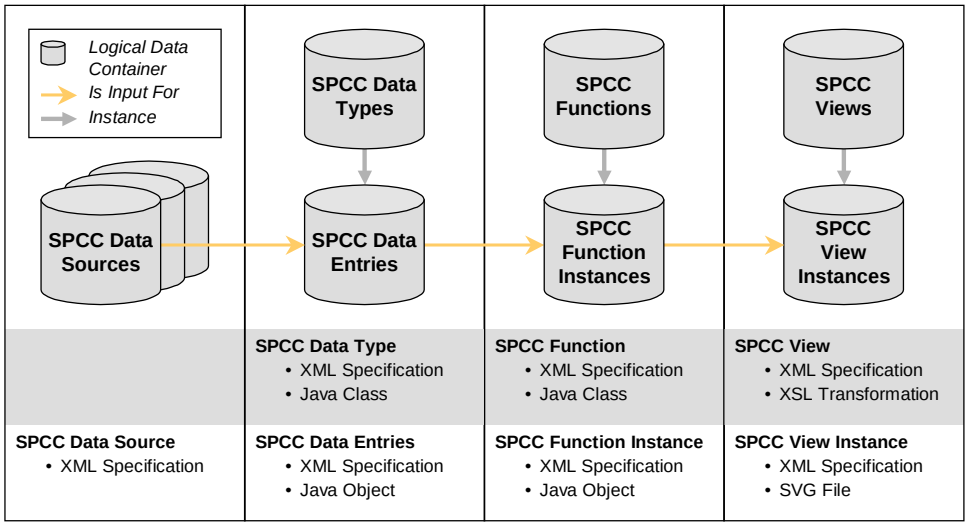


Abbildung 2: Überblick über die einzelnen Komponenten des SPCCs (aus [11]). Die oberen Teile zeigen den Datenfluss und die Instanziierungskette zwischen den Elementen der Visualisierungskette. Der untere Teil zeigt die zugehörigen Komponenten mit ihrer formalen Spezifikation.

Nach der Datenerfassung müssen die Daten verarbeitet und visualisiert werden. Hierbei ist es sowohl denkbar, die Daten direkt darzustellen oder mit anderen Daten in Zusammenhang (z.B. aus der Erfahrungsdatenbank) zu bringen. Die Verarbeitung der Daten findet in einer so genannten Visualisierungskette statt. Eine Visualisierungskette ist eine Hierarchie von Daten, Funktionsinstanzen und Sichten-

instanzen, welche aufeinander aufbauen, um die Daten mittels der Funktionen zu transformieren und anschließend zu visualisieren. Diese Visualisierungskette wird vom SPCC aufgebaut, verwaltet und aktualisiert.

Eine solche Kette besteht aus den folgenden sechs Komponenten:

- Datenquelle: Bei den Datenquellen handelt es sich um die Orte, an welchen sich die Daten befinden, wobei es sich hier beispielsweise um SQL-Server oder XML-Dateien handeln könnte.

- Datentypen: Mit Hilfe der Datentypen kann festgelegt werden, welche Struktur die eingehenden Daten haben. So kann festgelegt werden, welches Format die Daten haben, die in eine Funktion bzw. Sicht eingehen. Ein solcher Datentyp muss in der Lage sein, die Daten aus den Datenquellen auszulesen. Ein Beispiel für einen Datentyp ist beispielsweise eine Zeitreihe, welche aus einer Menge von Zeitstempeln und zugehörigen Werten besteht.

- Dateneinträge: Bei den Dateneinträgen handelt es sich um die Messdatensätze, wobei zwischen internen und externen Dateneinträgen unterschieden wird. Bei externen Dateneinträgen handelt es sich um Messdatensätze, die auf einer Datenquelle bzw. deren Instanz liegen. Interne Dateneinträge werden durch eine Funktion erzeugt. Jedem Eintrag wird außerdem ein bestimmter Datentyp zugeordnet. Diese Einträge können in eine Funktion bzw. Sicht eingehen.

- Funktionen: Die Funktionen verarbeiten die erfassten Daten, d.h. sie repräsentieren eine Methode oder ein Verfahren, welches auf die Daten angewendet wird. Das kann sowohl eine Konvertierungsfunktion sein, die die Daten von einem Datentyp in einen anderen umwandelt, oder auch eine Funktion, welche die Einhaltung von Schranken überprüft (z.B. Vergleich des Aufwands mit vorgegebenen Schranken).

- Funktionsinstanzen: Bei den Funktionsinstanzen handelt es sich um die konkrete Instanziierung einer Funktion. Es findet eine Zuordnung von Messdaten bzw. Funktionsausgaben zu den Funktionen statt. In eine Funktioninstanz können sowohl externe als auch interne Dateneinträge (also Ergebnisse von anderen Funktionsinstanzen) eingehen.

- Sichten: Die Sichten visualisieren die Messdaten, d.h. die Daten werden dem Betrachter auf eine angemessene Art präsentiert. Ein Beispiel hierfür kann ein Kreisdiagramm oder auch ein Liniendiagramm sein.

- Sichteninstanzen: Analog zu den Funktionsinstanzen werden bei einer Sichteninstanz, einer Sicht die entsprechenden Messdaten bzw. Funktionsausgaben zugeordnet. Es ist auch möglich, dass andere Sichteninstanzen in eine Instanz eingehen.

In Abbildung 3 ist ein Beispiel einer Visualisierungskette, und damit einer Hierarchie von Daten, Funktionsinstanzen und Sichteninstanzen, zu sehen. Die Struktur einer Visualisierungskette besteht aus mehreren Ebenen. Die Eingaben der Funktionen in der untersten Transformationsebene bauen auf den externen Daten auf, die sich in der so genannten ersten Datenebene (*1st Data Layer*). In der zweiten Datenebene (*2nd Data Layer*) finden sich sowohl die Ausgangsdaten, die durch

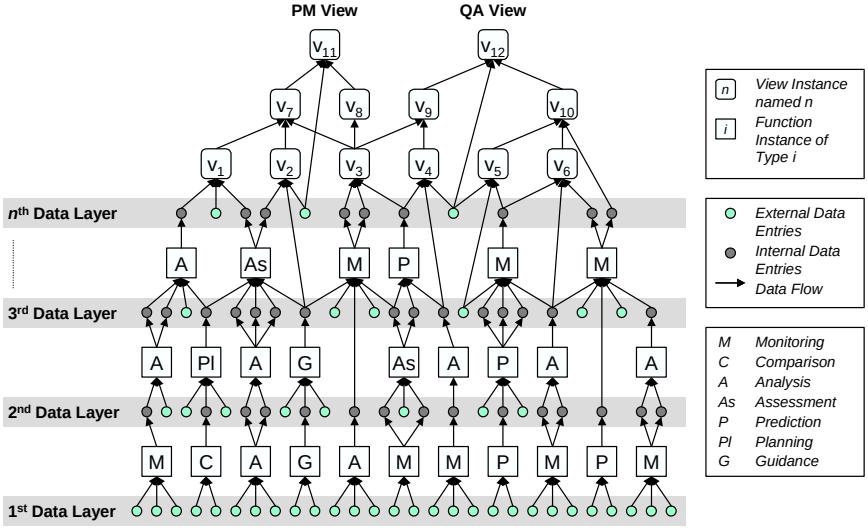


Abbildung 3: Beispiel einer möglichen Visualisierungskette für die Rollen des Projektmanagers und des Qualitätssicherers (aus [11]).

Funktionen in der ersten Transformationsebene erzeugt werden, als auch weitere benötigte externe Daten. Diese gehen in die Funktionen der zweiten Transformationsebene ein. Dies wird bis zur n-ten Daten-Ebene fortgesetzt. Die Daten dieser Ebene gehen in die erste Ebene der Sichten ein., diese erzeugten Sichten können wieder als Teilsichten in andere Sichten eingehen bis alle Informationen, die für eine Rolle von Interesse sind, enthalten sind.

Hierbei stehen *External Data Entries* für Daten aus XML-Dateien und *Internal Data Entries* für die Ausgaben von Funktionen. Wie zu erkennen ist, haben alle Funktionsinstanzen der ersten Ebene nur reine Messdaten als Einträge, während die Funktionsinstanzen einer höheren Ebene auch Funktionsausgaben als Einträge haben. Dies gilt auch für die Sichteninstanzen; so bestehen die Eingaben der Sichteninstanzen auf der ersten Ebene (beispielsweise V_1 und V_2) nur aus Messdaten und Funktionsausgaben, während die Sichten der höheren Ebenen (z.B. V_{12}) auch andere Sichteninstanzen als Eingaben haben können.

1.3 Das SPCC Tool

Der SPCC wurde als Client-Server Applikation realisiert, wobei die Datenerfassung durch ein anderes Werkzeug oder von Hand erfolgen muss, erst die erfassten Daten werden vom SPCC verwendet. Der Server übernimmt dann das Auslesen der Messdaten und wendet auf diese die Funktionen an, um sie anschließend in eine SVG-Grafik zu konvertieren.

Mit Hilfe der Clientapplikation können die Spezifikationsdateien der Visualisierungskette erstellt und angepasst werden. In Abbildung 4 ist das Fenster des Clients zu sehen, nachdem dieser gestartet wurde, mit dem Server (*Factory*) verbunden wurde und sich ein Benutzer eingeloggt hat.

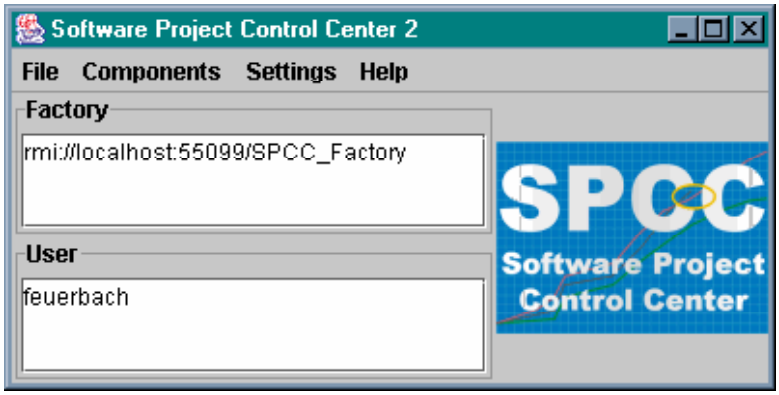


Abbildung 4: Hauptfenster Client nach dem Starten, Verbinden mit dem Server und Einloggen.

Außerdem ist es möglich, die Visualisierungsketten auszuwählen, die gerade bearbeitet werden sollen (siehe Abbildung 5). Hierzu muss jedoch zuvor beim Starten des Servers spezifiziert worden sein. Hierbei können mehrere Ketten angegeben und mit dem Client die benötigte ausgewählt werden.

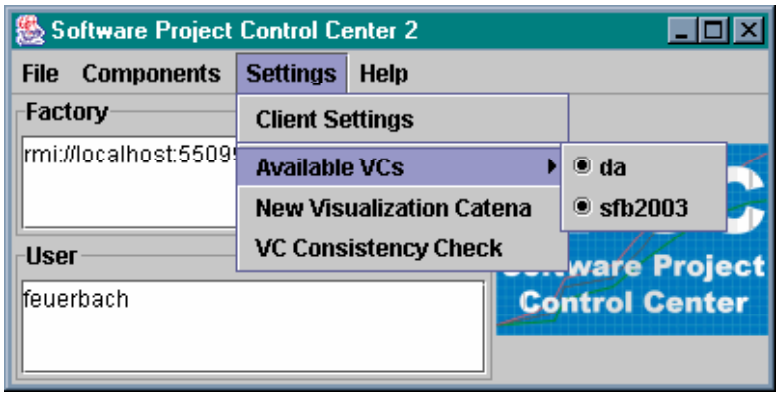


Abbildung 5: Auswahl der Visualisierungskette, die bearbeitet werden soll.

In Abbildung 6 ist das Menü zu sehen, in welchem man die die verschiedenen Komponenten der Visualisierungsketten auswählen kann, die bearbeitet werden sollen.

Ein Beispiel wie eine Funktion bearbeitet werden kann, ist in Abbildung 7 zu sehen. Es besteht die Möglichkeit, bestehende Eingaben, Parameter und Ausgaben zu bearbeiten, zu löschen oder auch neue hinzuzufügen.

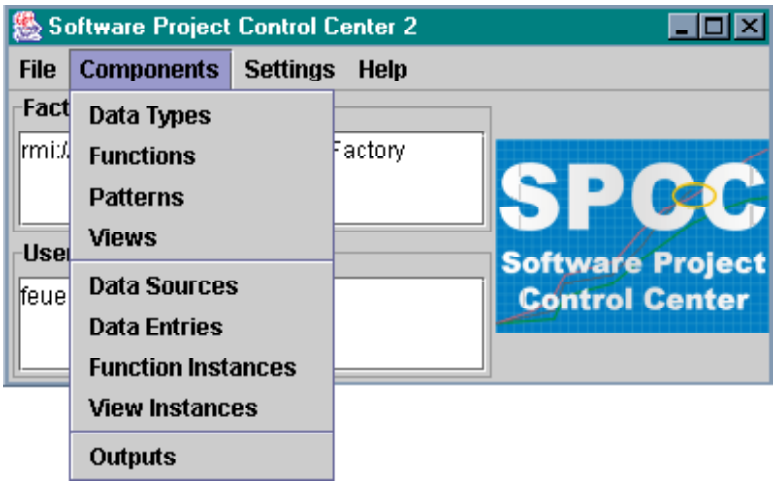


Abbildung 6: Menü zur Auswahl der Komponenten.

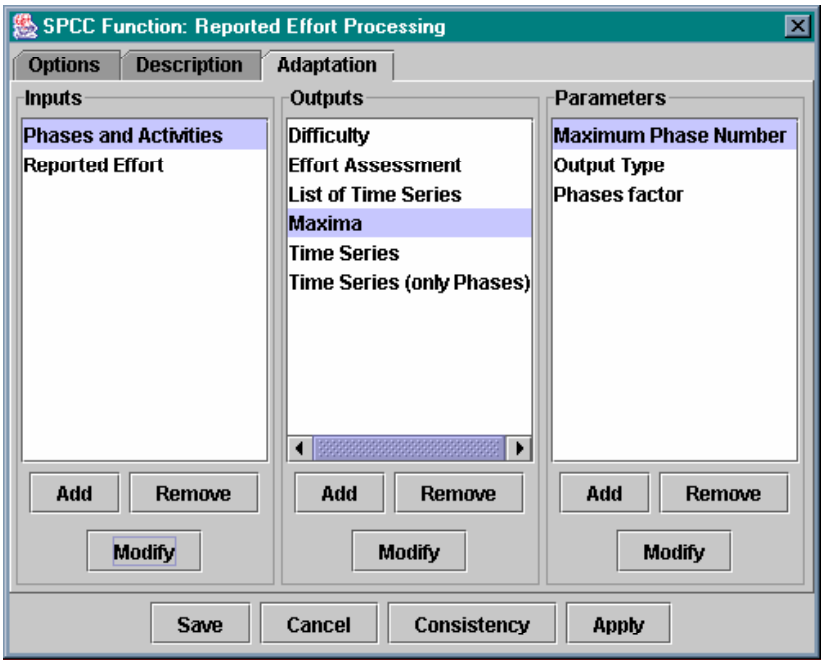


Abbildung 7: Fenster zum Bearbeiten der Funktion „Reported Effort Processing".

1.4 Aufgabenstellung

Das Thema der Diplomarbeit lautet „Ableitung von rollenspezifischen Sichten auf projektbegleitend gesammelten Daten". Hierbei sind folgende Arbeitspunkte durchzuführen:

- Einarbeiten in den grundlegenden SPCC Ansatz.
- Einarbeiten in Ansätze zum Messen und Bewerten und zur sichtenspezifischen Visualisierung von Messdaten.
- Recherchieren von gängigen Sichten auf Messwerte für projektbeteiligte Rollen aus der Literatur.
- Identifizieren von Anforderungen an Sichten für ausgezeichnete Rollen der Software-Entwicklung (z.B. Projektplaner, Projektmanager, Qualitätssicherer und Entwickler).
- Bewerten der gängigen Sichten und Identifizieren von eventuellen Schwachstellen.
- Formale Definition einer Menge von Sichten für die betrachteten Rollen.
- Erproben der definierten Sichten anhand von Daten eines realen Software-Entwicklungsprojektes und Hervorheben der Vorteile für ausgezeichnete Rollen.

1.5 Übersicht

Die Arbeit umfasst sechs Kapitel, wobei im ersten Kapitel die Grundlagen, auf denen die Arbeit aufbaut und notwendig für das weitere Verständnis sind, erläutert werden. Hierzu zählt auch die Vorstellung des SPCCs und Einführung von Metriken und des GQM-Ansatzes. Außerdem werden die wichtigsten Diagrammtypen vorgestellt, und deren jeweilige Vor- und Nachteile erläutert.

In Kapitel 2 werden verschiedene existierende Ansätze aus der Literatur einander gegenüber gestellt und die resultierenden Sichten erläutert.

In Kapitel 3 werden zuerst Anforderungen an Sichten abgeleitet und im Anschluss eine mögliche Klassifikation von Sichten erläutert. Danach wird ein Metamodell vorgestellt, anhand dessen die Zusammenhänge zwischen Sichten dargestellt werden können. Zuletzt werden die gefundenen Sichten aus Kapitel 2 bewertet.

Das Kapitel 4 befasst sich mit der Herleitung von Sichten; hierfür werden zuerst verschiedene Rollen eines Software-Entwicklungsprozesses vorgestellt. Im Anschluss werden die verschiedenen Metriken definiert, auf denen die darauf folgenden Sichten basieren.

In Kapitel 5 wird die Implementierung ausgewählter Sichten beschrieben und die Sichten werden evaluiert.

Das letzte Kapitel fasst die Ergebnisse noch einmal zusammen und gibt einen Ausblick über mögliche Verbesserungen und Erweiterungen.

Im Anhang finden sich die Diagramme der Sichten aus der Literatur. Außerdem ist der XSL-Kode der implementierten Sichten und der Java-Kode der implementierten Funktionen angehängt.

1.6 Metriken

Wie bereits in Abschnitt 1.2 erwähnt wurde, dienen Metriken dazu, die Messdaten zu erfassen bzw. diese mittels Funktionen zu transformieren, um damit die Einhaltung von vorgegebenen Zielen zu überwachen. Bis jetzt wurde jedoch noch nicht genau erläutert, was man unter einer Metrik versteht und welche Möglichkeit es gibt, diese zu klassifizieren; daher wird dies in diesem Kapitel nun nachgeholt.

Mit Hilfe einer Metrik oder einem Maß können unterschiedliche Attribute von Prozessen, Projekten oder auch Produkten quantifiziert werden [14]. Mögliche Attribute sind hierbei z.B. Projekt-Kosten, Projekt-Größe und Fehler im Quellkode. Es gibt verschiedene Arten, Metriken einzuteilen. So gibt es die Möglichkeit zwischen einfachen und komplexen Metriken zu unterscheiden. Bei einfachen Metriken handelt es sich um solche, die nur auf einer unabhängigen Variablen basieren (z.B. Systemkomplexität = Anzahl der Komponenten), während komplexe Metriken auf einer Menge von unabhängigen Variablen basieren (z.B. Produktivität = Anzahl der entwickelten Kodezeilen/Zeitraum). Weiterhin ist es möglich eine Einteilung in Produkt- und Prozessmetriken zu machen. Hierbei sind Produktmetriken diejenigen, welche Produkteigenschaften bewerten, wie z.B. die Korrektheit abhängig von der Anzahl der Fehler ist. Prozessmetriken dagegen bewerten die Prozesseigenschaften; so sind die Kosten abhängig von der Anzahl der Arbeitsstunden. Man unterscheidet auch zwischen direkten und indirekten Metriken, wobei direkte Metriken direkte Qualitätseigenschaften, d.h. Qualitätseigenschaften, an welchen man direkt interessiert ist, bewerten (z.B. Zuverlässigkeit = Anzahl der Fehler/Jahr). Analog bewerten indirekte Metriken indirekte Qualitätseigenschaften, dies sind solche, die man eventuell braucht, um Aussagen über die Qualitätsaspekte zu machen; Beispiel für eine indirekte Qualitätseigenschaft ist die Komplexität. Schließlich gibt es noch die Unterteilung in objektive und subjektive Metriken. Dabei können objektive Metriken direkt vom Objekt, welches gemessen wird, abgeleitet werden. Das bedeutet außerdem, dass der Wert unabhängig vom jeweiligen Aktor ist (z.B. Umfang = Anzahl der Kodezeilen). Im Gegenteil dazu sind die Werte der subjektiven Metriken abhängig von der Person, die die Messung vornimmt (z.B. Verständlichkeit des Kodes).

Metriken können nicht nur dazu dienen, Attribute zu quantifizieren, sondern auch um die Bestimmung von Meilensteinemöglich zu machen, indem man für ein Attribut einen Wert vorgibt, welcher erreicht werden muss, damit eine Phase abgeschlossen ist.

1.7 GQM (*Goal Question Metric*)

Mit Hilfe des SPCC soll die Einhaltung definierter Ziele überprüft und gewährleistet werden. Dazu muss eine Basis definiert werden, die als Grundlage zur Beurteilung dient. Hierbei helfen Metriken, da diese verschiedene Eigenschaften bewerten. Um jedoch ausgehend von den Zielen die geeigneten Metriken abzuleiten, wird der GQM-Ansatz benutzt, der in diesem Kapitel genauer erläutert wird.

Victor R. Basili und Dave Weiss entwickelten den *Goal Question Metric* (GQM) Top-Down-Ansatz, mit welchem es möglich ist, aus einer Menge von Messzielen Maße systematisch abzuleiten ([8]). Hierzu wird zuerst eine Menge von Zielen definiert, welche erreicht werden sollen. Im Anschluss wird eine Menge von Fragen gesucht,

die beim Erreichen der zuvor definierten Ziele hilfreich sein können. Für jede dieser Fragen wird nun eine Menge von Maßen gewählt, um diese zu beantworten. Jedem Ziel wird in der Regel mehr als eine Frage zugeordnet, ebenso wird jeder Frage mehr als ein Maß zugeordnet, wobei ein Maß auch mehr als einer Frage zugeordnet werden kann. Dies resultiert daraus, da jeweils eine Verfeinerung stattfindet.

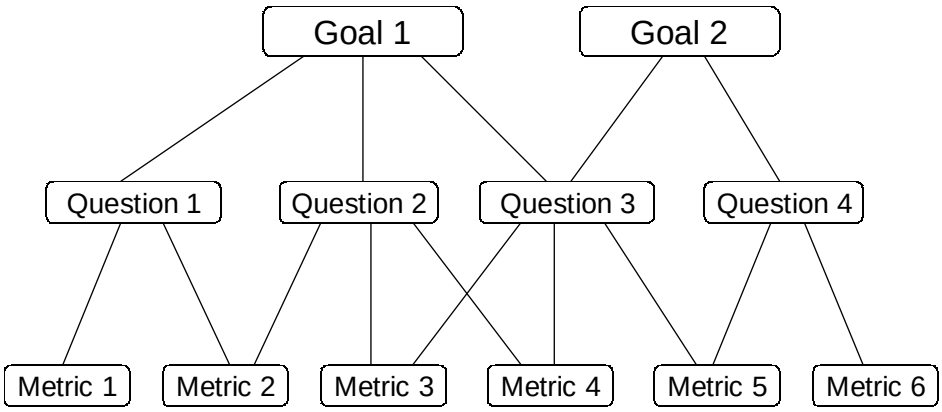


Abbildung 8: Beispiel einer möglichen Hierarchie von Zielen, Fragen und Maßen.

Für jedes Ziel gibt es folgende fünf Aspekte, anhand welcher dieses eingeordnet werden kann:

- Objekt: Bezeichnet das Objekt, welches analysiert werden soll, es kann sich hierbei z.B. um Prozesse oder auch Produkte handeln.
- Zweck: Beschreibt zu welchem Zweck die Analyse durchgeführt werden soll. Mögliche Gründe können sein: Kontrolle, Vorhersage, Bewerten, usw.
- Aspekt: Mit dem Aspekt wird ein Attribut beschrieben, welches untersucht werden soll. Dabei kann es sich um Kosten, Einhaltung von Terminen, Wartbarkeit usw. handeln.
- Blickwinkel: Beschreibt die Rollen/Personen, für welche die Ergebnisse von Bedeutung sind.
- Kontext: Hiermit wird beschrieben, in welcher Umgebung das Projekt durchgeführt wird, z.B. Universität oder konkrete Organisation. Dies ist wichtig, bei der Wiederverwendung und Interpretation der Daten; so sind für den Aufwand eines Praktikum an der Universität die Daten eines Projektes aus der Industrie kein aussagekräftiger Anhaltspunkt, da allein der Umfang der Projekte schon stark von einander abweicht.

1.8 Diagrammtypen

Es gibt eine Vielzahl von unterschiedlichen Diagrammtypen, die wiederum für verschiedene Arten von Daten geeignet sind. Hierbei kann zwischen zwei- und dreidimensionalen Diagrammen unterschieden werden. Häufig können die Typen auch

von zwei- nach dreidimensional erweitert werden, es gibt jedoch auch Fälle, in denen dies nicht möglich ist, da ein erheblicher Mehraufwand nötig ist, um die Diagramme von unterschiedlichen Ansichten betrachten zu können, da ansonsten der größte Teil der Daten von anderen überdeckt wird.

Im Anschluss werden verschiedene Typen beschrieben und jeweils auch eine Abbildung zur Verdeutlichung gegeben. Außerdem werden die Vor- und Nachteile der einzelnen Typen erläutert.

1.8.1 Liniendiagramm

Bei einem Liniendiagramm handelt es sich um eine einfach darzustellende und gut verständliche Art von Diagrammen, wobei im zweidimensionalen Fall eine Achse dem Parameter und die zweite Achse dem korrespondierenden Wert entspricht. Ein Liniendiagramm kann auch dreidimensional sein, in diesem Fall kennzeichnen zwei Achsen Parameter.

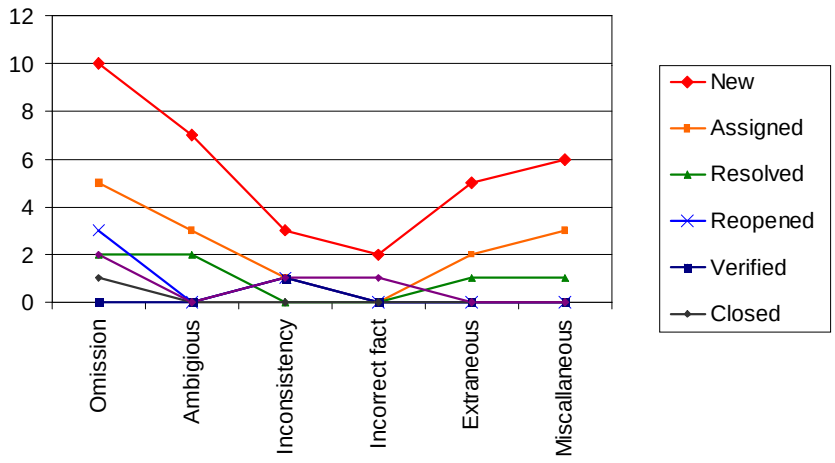


Abbildung 9: Beispiel eines zweidimensionalen Liniendiagramms. Es wird hierbei eine mögliche Verteilung der verschiedenen Fehlerklassen (x-Achse) auf die unterschiedlichen Status (pro Status eine Kurve) dargestellt.

Vorteile:

Die Vorteile eines Liniendiagramms liegen darin, dass dieses eine sehr einfache Art von Diagramm ist und auch mehrere Kurven in einem einzigen Diagramm eingetragen werden können. Diese sind im zweidimensionalen Fall auch gut miteinander zu vergleichen. Zusätzlich kann der Parameter kontinuierlich verändert werden, d.h. entlang einer Achse kann ein kontinuierlicher Bereich aufgetragen werden.

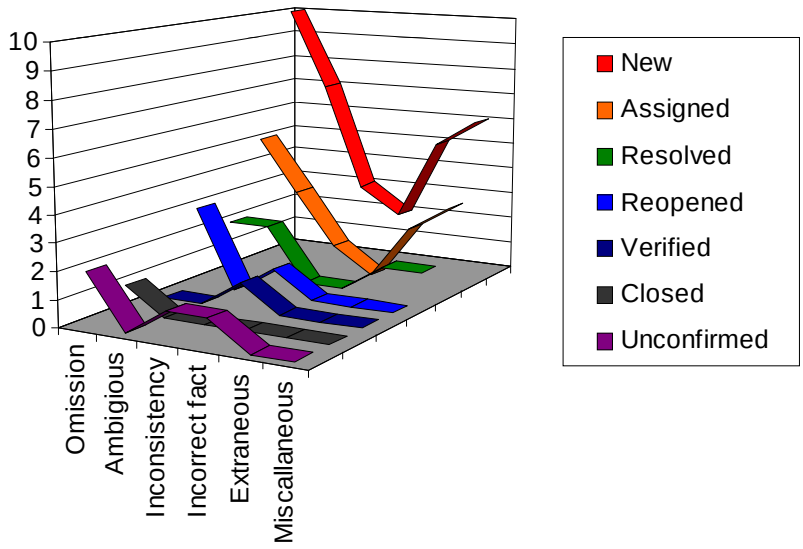


Abbildung 10: Beispiel des Sachverhalts von Abbildung 9 als dreidimensionales Liniendiagramm. Der zweite Parameter wäre hier der Status der Fehler.

Nachteile:

Ein Nachteil eines Liniendiagramms besteht darin, dass im zweidimensionalen Fall nur ein Parameter möglich ist; der zweite muss durch die Verwendung von mehreren Kurven simuliert werden. Dies wird bei einer zu großen Anzahl von Kurven unübersichtlich. Für den zweiten Parameter können außerdem nur diskrete Werte dargestellt werden. Soll auch für den zweiten Parameter ein kontinuierlicher Bereich aufgetragen werden, muss eine dreidimensionale Oberfläche verwendet werden.

1.8.2 Balkendiagramm

Ein Balkendiagramm ist ebenfalls ein einfach darzustellendes Diagramm, welches sowohl zwei- als auch dreidimensional sein kann. Eine oder zwei Achsen entsprechen Parametern. Hierbei können jedoch nur diskrete Werte auf der x-Achse aufgetragen werden, da diese Darstellung ansonsten in eine Liniendiagramm übergeht.

Vorteile:

Ein klarer Vorteil eines Balkendiagramms liegt darin, dass es möglich ist, auch im zweidimensionalen Fall zwei Parameter zu unterscheiden, indem eine Unterteilung nach einem Parameter entlang der Achse stattfindet und der zweite durch verschiedene Balken im gleichen Achsenabschnitt repräsentiert werden kann. Um die Unterscheidung zu vereinfachen, ist es sinnvoll den zweiten Parameter dadurch kenntlich zu machen, dass die zusammengehörigen Balken das gleiche Muster bzw. die gleiche Farbe aufweisen (siehe Abbildung 12).

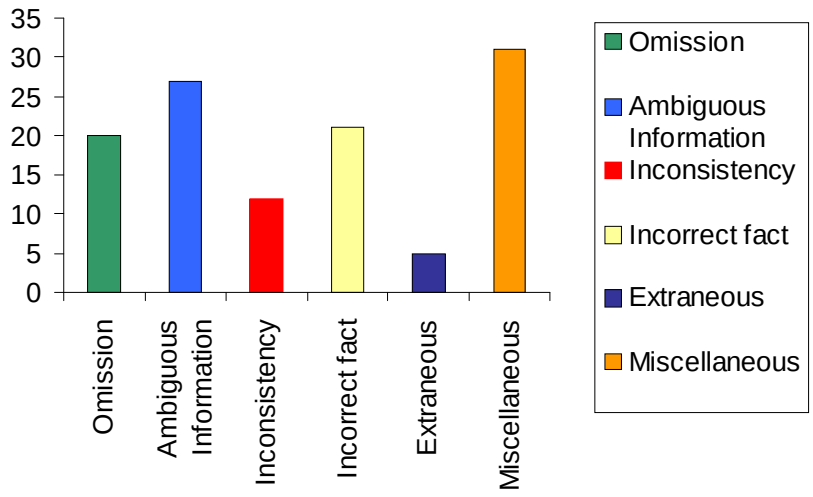


Abbildung 11: Beispiel eines zweidimensionalen Balkendiagramms. Es wird für jede Fehlerklasse die absolute Häufigkeit dargestellt.

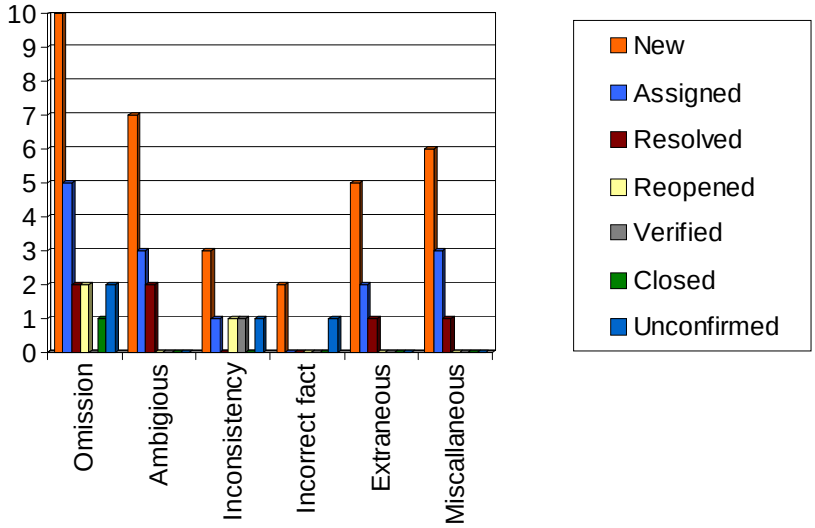


Abbildung 12: Beispiel für ein Balkendiagramm mit zwei Parametern. Der erste Parameter wird entlang der x-Achse aufgetragen und der zweite Parameter wird über die unterschiedlichen Farben der Balken kodiert. Im Gegensatz zum vorangegangenen Diagramm, wird hier nicht nur die Anzahl der Fehler pro Fehlerklasse, sondern auch der Status der jeweiligen Fehler aufgetragen.

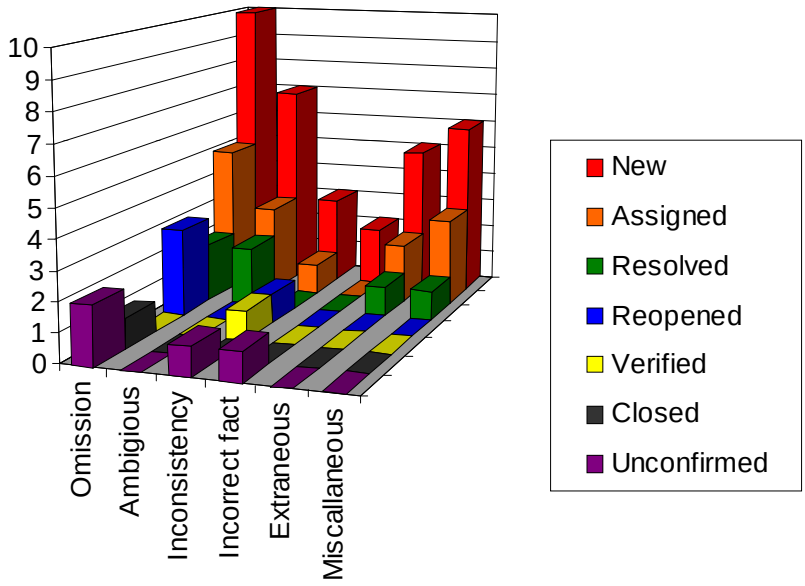


Abbildung 13: Beispiel eines möglichen dreidimensionalen Balkendiagramms. Es werden die gleichen Daten wie im vorangegangen Diagramm dargestellt, mit dem Unterschied, dass dies hier dreidimensional geschieht, indem der Status eines Fehlers entlang einer eigenen Achse aufgetragen werden.

Nachteile:

Der Vorteil kann jedoch auch gleichzeitig ein Nachteil sein, denn bei einer zu großen Anzahl von verschiedenen Werten für die beiden Parameter, können die Balken entweder nicht mehr gleichzeitig dargestellt werden (bei dynamischen Bildern gibt es evtl. die Möglichkeit zum Scrollen) oder die Balkenbreite kann so schmal werden, dass es schwierig wird, die einzelnen Balken zu erkennen und miteinander zu vergleichen. Weiterhin kann ein Balkendiagramm nur verwendet werden, wenn die Parameter diskrete Werte annehmen, z.B. unterschiedliche Phasen eines Projekts.

1.8.3 Kreisdiagramm

Mit einem Kreisdiagramm (oder auch Tortendiagramm) ist es möglich das Verhältnis zwischen Teilen und dem Ganzen darzustellen. Daher ist diese Art besonders geeignet, wenn es um eine Verteilung eines Ganzen geht; wird nur ein Teil betrachtet, ist es wiederum weniger aussagekräftig. Außerdem müssen die einzelnen Teilmengen disjunkt zueinander sein, es darf also nicht vorkommen, dass ein Element zu mehr als einer Teilmenge gehört. In diesem Fall muss eine neue Teilmenge eingeführt werden, welche genau die Elemente umfasst, welche in beide Gruppen fallen.

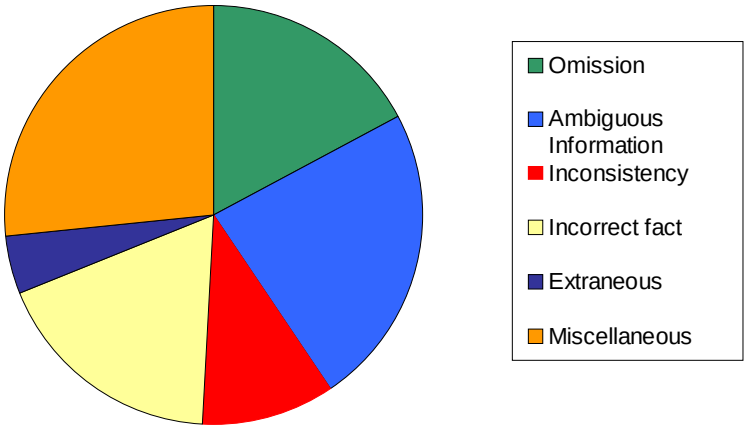


Abbildung 14: Beispiel eines Kreisdiagramms; es wird die relative Häufigkeit der einzelnen Fehlerklassen dargestellt.

Vorteile:

Ein Vorteil eines Kreisdiagramms besteht darin, dass es sich, wie schon bei den beiden zuvor erläuterten Typen, um eine einfach darzustellende und intuitiv verständliche Art eines Diagramms handelt. Außerdem bietet ein Kreisdiagramm eine gute Vergleichsmöglichkeit der verschiedenen Werte miteinander, so dass bereits nach kurzer Zeit eine Aussage über die relative Verteilung getroffen werden kann.

Nachteile:

Ein wesentlicher Nachteil eines Kreisdiagramms besteht darin, dass nur ein Verhältnis abgelesen werden kann und keine Aussagen über die absoluten Werte gemacht werden können, sofern diese nicht angegeben werden. So kann bei einer geringen Gesamtzahl bereits eine Änderung um eine Einheit eine große relative Änderung zur Folge haben. Ein Kreisdiagramm kann zwar dahingehend erweitert werden, dass es dreidimensional wird (= Tortendiagramm), dies erhöht jedoch die Aussagekraft nicht. Weiterhin kann nur die Verteilung von disjunkten Mengen dargestellt werden, oder es muss eine neue Teilmenge eingeführt werden. Dies führt bei einer zu großen Anzahl von Mengen dazu, dass das Diagramm an Aussagekraft verliert, da die Anteile der einzelnen Mengen so gering werden, dass diese nicht mehr in Verhältnis zueinander gesetzt werden können, ohne dass das relative Verhältnis angegeben wird, d.h. zu jedem Kreisabschnitt müsste der zugehörige Wert angegeben werden.

1.8.4 Matrix

Bei einer Matrix handelt es sich um eine Variante des Balkendiagramms, wobei in diesem Fall beide Achsen Parameter repräsentieren und diese wiederum nur diskrete Werte annehmen können. Eine „dritte" Achse liegt parallel zu einer der beiden

Achsen und beginnt für jeden Parameter-Wert wieder bei Null. Wir gehen im Folgenden davon aus, dass dies die waagerechte Achse ist. Auf dieser Achse wird dann der Wert, der einer Kombination von zwei Werten zugeordnet wird, dargestellt, wobei die Breite des Balkens wertabhängig ist.

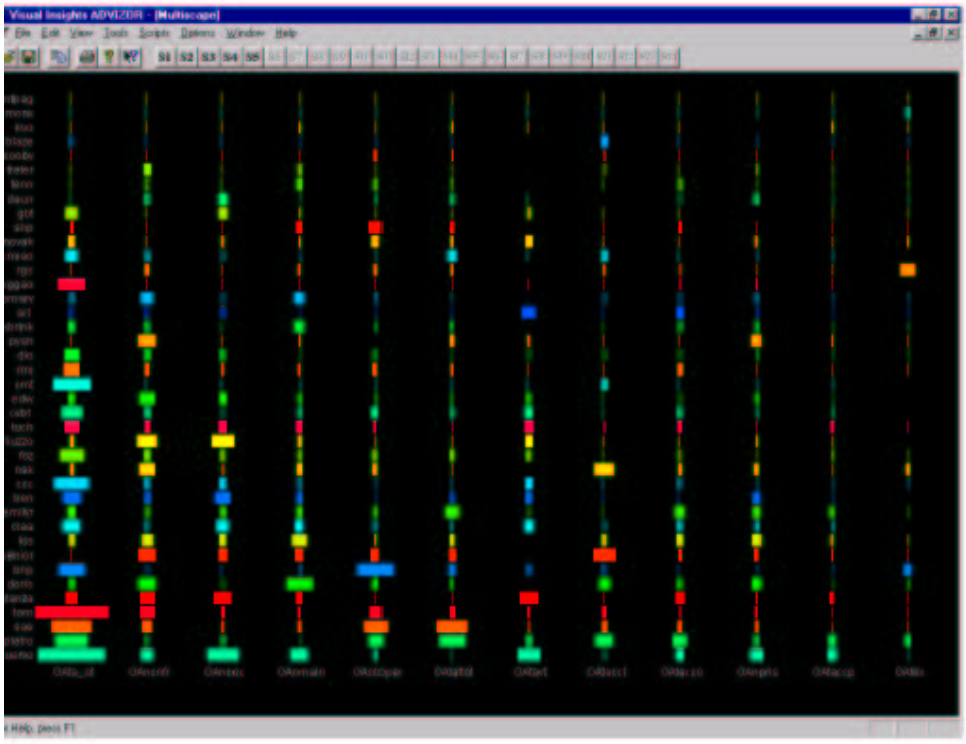


Abbildung 15: Beispiel einer Matrix-Darstellung (nach [4]). Hierbei sind entlang der y-Achse die unterschiedlichen Entwickler aufgetragen und entlang der x-Achse die Module. Die Breite des Balkens kodiert den Umfang des Moduls und die Farbe den Entwickler.

Vorteile:

Es können mit Hilfe einer Matrix-Darstellung auch Daten, welche von zwei Parametern abhängig sind, zweidimensional dargestellt werden, wobei der große Vorteil darin liegt, dass zumindest einer der beiden Parameter einen größeren Bereich umfassen kann, nämlich derjenige, welche auf der ersten Achse abgebildet wird.

Nachteile:

Auch hier ist der Vorteil wieder gleichzeitig ein Nachteil, denn durch die Beschränkung auf Zweidimensionalität muss der Abstand zwischen zwei Werten des Parameters auf der zweiten Achse mindestens so groß sein, dass der maximale Wert, welcher für diesen Parameter auftritt, noch auf der dritten Achse dargestellt werden kann. Dadurch können wiederum auch nur diskrete Werte dargestellt werden. Weiterhin

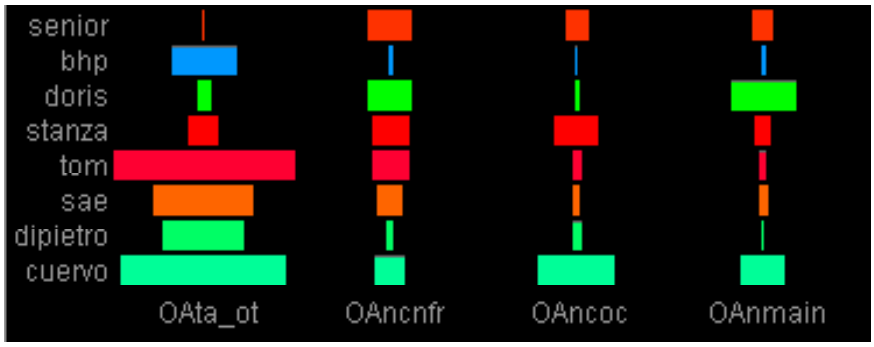


Abbildung 16: Vergrößerter Ausschnitt der vorangegangenen Abbildung einer Matrix.

wird hierdurch auch der gleichzeitig darstellbare Bereich des zweiten Parameters stark eingeschränkt.

1.8.5 Netzwerk

Mit Hilfe eines Netzwerkes ist es möglich, dass z.B. Strukturen innerhalb von Software dargestellt werden können, weitere Anwendungsbeispiele sind die Darstellung eines Produkt- oder Kontrollflusses, d.h. wie der Ablauf eines Programms aussieht (Aufrufbeziehungen usw.) oder wann welche Produkte erstellt werden und welche hierbei verwendet werden. Hierbei entsprechen die Knoten den verschiedenen Softwaremodulen und eine Kante zwischen zwei Knoten bedeutet, dass die Module auch miteinander verbunden sind. Ein Beispiel hierfür sind Module, welche von der gleichen Änderung betroffen sind.

Vorteile:

Ein Vorteil von Netzwerken besteht darin, dass hiermit Strukturen oder Abläufe dargestellt werden können, was mit den anderen Diagrammtypen nicht möglich ist.

Nachteile:

Die Einsatzmöglichkeiten von Netzwerken sind relativ gering, da aufgrund der fehlenden Möglichkeit, Mehrfachzusammenhänge darzustellen, nur einfache Strukturen dargestellt werden können. Dies ergibt sich daraus, dass immer nur zwei Knoten durch eine gemeinsame Kante miteinander verbunden werden können, da ansonsten nicht genau definiert ist, wie z.B. Zusammenhänge, die drei Knoten umfassen, dargestellt werden sollen. Es ist zwar möglich die Knoten immer paarweise miteinander zu verbinden, jedoch ist nicht mehr erkennbar, dass es sich um einen gemeinsamen Zusammenhang handelt, vor allem, wenn mehrere Zusammenhänge im gleichen Diagramm eingetragen werden. Die Alternative, pro Zusammenhang einen zusätzlichen Knoten zu verwenden und alle betroffenen Knoten mit diesem zu verbinden, hat das Problem, dass nicht mehr alle Knoten einem Softwaremodul entsprechen und zusätzlich das Diagramm noch mehr mit Daten gefüllt wird und dadurch noch unübersichtlicher wird.

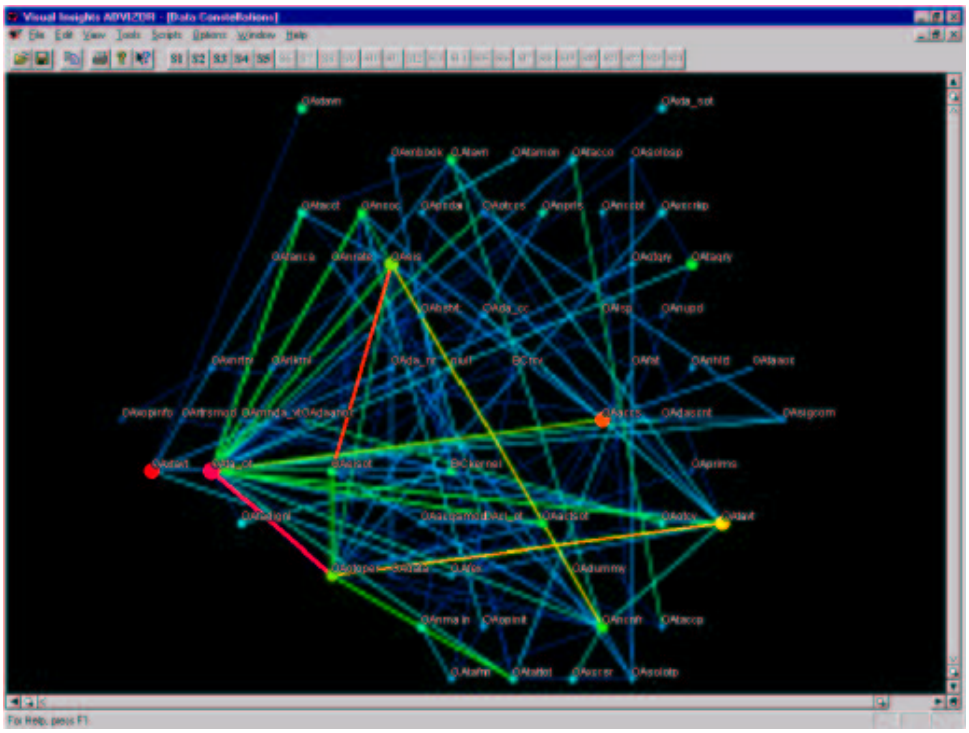


Abbildung 17: Beispiel eines Netzwerks (nach [4]). Es wird dargestellt, welche Dateien von einer Änderungen betroffen sind, d.h. wenn Datei 1 geändert wurde, wie häufig war davon auch Datei 2 betroffen. Die Häufigkeit wird über die Farbe der Verbindung kodiert.

Weiterhin kommt es fast immer zu Überlappungen bei der Darstellung, so dass bei zu vielen Kanten die Verbindungen nicht mehr unterscheidbar sind, und daher keine Informationen mehr daraus gewonnen werden können.

1.8.6 Tabelle

Ein Diagramm kann auch aus einer Tabelle bestehen, welche die Daten ausschließlich textuell und zweidimensional darstellt. Diese kann dazu verwendet werden, wenn entweder der Wertebereich nur wenige Werte enthält (z.B. ist erfüllt, ist nicht erfüllt), oder um möglichst genaue und detaillierte Informationen darzustellen.

Vorteile:

Ein Vorteil liegt darin, dass sehr viele Informationen dargestellt werden können und diese genau und detailiert sein können. Dadurch ist es möglich, die verschiedenen Daten genau miteinander zu vergleichen.

	Funktionale B-Anforderungen	Nicht-Funktionale B-Anforderungen	inverse B-Anforderungen
1	++	-	o
2	+++	+	+
3	++	--	+
4	+	+	
5	++	++	

Anforderung erfüllt

+, -, o Priorität der Anforderung

Abbildung 18: Beispiel einer sinnvollen Anwendung einer Tabelle. Es werden die Anforderungen zusammen mit der zugehörigen Priorität dargestellt und diejenigen markiert, die bereits erfüllt sind.

Nachteile:

Durch den möglicherweise hohen Detailgrad, kann es dazu kommen, dass mehr Informationen dargestellt werden, als benötigt werden. Aufgrund der reinen textuellen Darstellung ist es außerdem schwierig, die benötigten Informationen zu suchen und miteinander zu vergleichen. So müssen erst alle Einträge gelesen werden, bevor der maximale Wert bestimmt werden kann.

1.8.7 Zeitplan

Mit Hilfe eines Zeitplans ist es, wie der Name schon sagt, möglich einen zeitlichen Ablauf darzustellen. Auf einer Achse wird die Zeit aufgetragen (z.B. in Wochen) und auf der anderen Achse können z.B. verschiedene Arbeitspakete aufgetragen werden. Anschließend wird mittels Balken gekennzeichnet, in welchen Wochen die jeweiligen Arbeitspakete bearbeitet werden sollen.

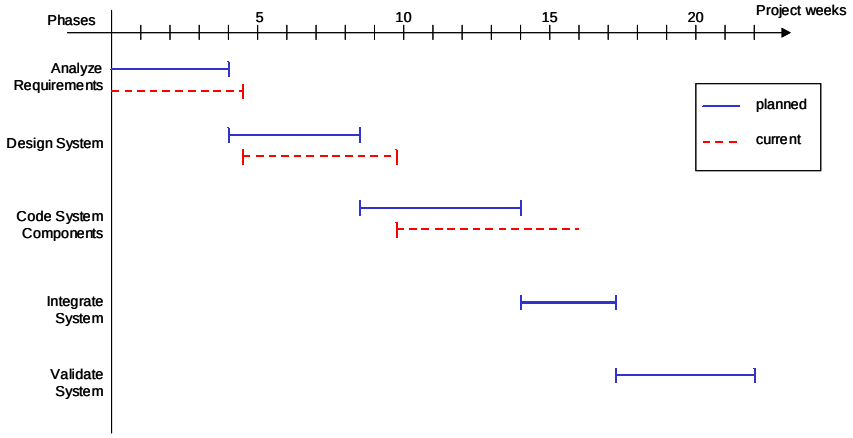


Abbildung 19: Beispiel eines Zeitplans.

Vorteil:

Ein Zeitplan hat den Vorteil, dass er eine Möglichkeit darstellt, eine zeitliche Aufteilung von Arbeitspaketen graphisch darzustellen und nicht nur textuell.

Nachteil:

Der Nachteil eines Zeitplans liegt darin, dass bei zu großer Gesamtdauer, welche dargestellt werden soll, es schwierig wird, die unterschiedlichen Balkenlängen zu erfassen. Daher kann die Dauer der einzelnen Arbeitspakete kaum noch geschätzt werden und somit nur schlecht miteinander verglichen werden.

1.8.8 Box-Plots

Bei Box-Plots handelt es sich um einen Typ von Diagramm, der vor allem in der Statistik recht häufig verwendet wird. Mit Hilfe eines solchen Diagramms können mehrere Fragen auf einmal beantwortet werden. Zu diesen Fragen gehören: „Wo ist das Zentrum der Werte? Innerhalb welcher Grenzen liegen 50% der Werte? Welche Werte weichen extrem ab, und welche sind weit vom Zentrum entfernt, jedoch noch keine Extremwerte."

In Abbildung 20 ist der Aufbau eines Boxplots zu sehen, der hier ausführlicher erläutert wird. Der so genannte Interquartilabstand entspricht dem Abstand zwischen dem ersten und dritten Quartilwert. Der ersten Quartilwert ist so definiert, dass höchstens 25% aller Werte kleiner sind, als dieser Wert. Analog gilt für den dritten Quartilwert, dass höchstens 75% aller Werte kleiner sind. Der Median entspricht dem zweiten Quartilwert: höchstens 50% aller Werte sind kleiner. Die Striche, welche die Extremwerte von den anderen Werten trennen, reichen maximal bis zum anderthalbfachen Interquartilabstand.

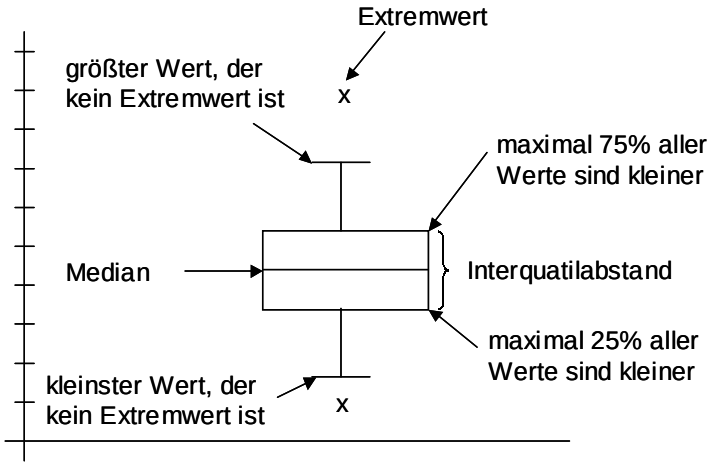


Abbildung 20: Aufbau eines Boxplots.

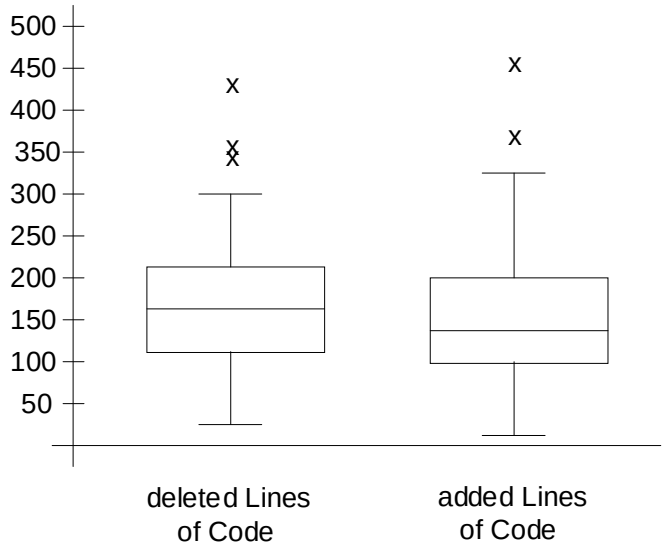


Abbildung 21: Beispiel eines Boxplots, das die Anzahl gelöschter und hinzugefügter Kodezeilen zeigt.

Vorteil:

Ein Vorteil von Boxplots liegt darin, dass mehrere Fragen auf einen Blick beantwortet werden können, welche eine Rolle für statistische Auswertungen spielen.

Nachteil:

Es können einerseits die Fragen einer statischen Auswertung auf einen Blick beantwortet werden, andererseits kann keine Aussage darüber getroffen werden, wie sich die Werte innerhalb des Intervalls (innerhalb des Interquartilabstands) verteilen. Also ob sich die Werte beispielsweise um den Mittelwert verteilen, oder eher an den Grenzen des Intervalls zu finden sind. Wenn z.B. Aussagen über die Qualität eines Software-Projektes getroffen werden sollen und dafür die Fehlerverteilung über alle Klassen dargestellt werden soll, so kann zwar eine Aussage darüber getroffen werden, dass im Mittel die Anzahl der Fehler einen bestimmten Wert hat. Allerdings ist es nicht möglich zu sagen, wie die tatsächliche Verteilung aussieht und ob die Werte gehäuft um den Median oder im gesamten Intervall verteilt sind. So können bei einem Vergleich verschiedener Projekte die gleiche Box-Plot-Darstellung erzeugt werden, obwohl die tatsächlichen Werte stark voneinander abweichen.

2 Literaturrecherche

Das Ziel dieser Literaturrecherche besteht darin, mögliche Sichten auf Messdaten zu finden, um aus der Betrachtung der existierenden Ansätze Anforderungen ableiten zu können. Ein weiteres Ziel ist es, einen Überblick über die verschiedenen Darstellungsarten und deren Vorteile und Nachteile zu gewinnen. Dies dient dann als Grundlage für die Ableitung von Anforderungen an Sichten und für die Entwicklung eines Meta-Modells, welches im Kapitel 22 entwickelt wird und die Zusammenhänge zwischen den verschiedenen Sichten verdeutlichen soll.

2.1 Anmerkungen zur Recherche

Für die Suche nach verwendbaren Quellen wurden neben den Katalogen verschiedener Universitätsbibliotheken auch allgemeine Suchmaschinen und die einiger Zeitschriften der Informatik verwendet. Die wichtigsten Zeitschriften waren: „The Journal of System and Software", „IEEE - Software", „ACM - Computing Surveys", „Software Process Improvement & Practice", „IEEE Transaction on SE" und „Empirical Software Engineering". Für die Zeitschriften von ACM und IEEE gab es jeweils die Möglichkeit eine Online-Suchmaschine zu benutzen, welche die jeweilige Datenbank nach Artikeln durchsucht. Es wurde eine Menge von Schlüsselwörtern ausgewählt, welche eine zentrale Rolle für diese Arbeit spielen, und diese dann dazu benutzt, in den unterschiedlichen Datenbanken zu recherchieren. Diese Begriffe sind: „visualization", „software data", „views", measurement" und „metrics".

Aus den resultierenden Ergebnismengen, wurden diejenigen Artikel in einer Menge zusammengefasst, die sich mit Sichten beschäftigen, da diese als Grundlage für den Vergleich dienen können. Weiterhin wurden die Artikel, welche sich mit Metriken beschäftigen, ausgewählt. Denn aus diesen Metriken können diejenigen ausgewählt werden, welche für die resultierenden Sichten benötigt werden . Zusätzlich wurden auch die Artikel verwendet, die sich mit unterschiedlichen Diagrammtypen beschäftigen, um daraus möglichst geeignete Darstellungsarten für die Sichten zu erhalten.

Diese Menge diente als Grundlage für diese Arbeit, wobei die Artikel in zwei weitere Suchmengen unterteilt wurden, abhängig vom Zweck, zu welchem der Artikel verwendet wurde. Die erste Suchmenge umfasst die Artikel, welche sich mit der Visualisierung von verschiedenen Sichten beschäftigen, um deren Ansätze miteinander zu vergleichen und auch die Vor- und Nachteile der Sichten zu analysieren. Diese Menge umfasst fünf verschiedene Artikel, welche als Grundlage für das folgende Kapitel dienen.

Die restlichen Artikel wurden in der zweiten Menge zusammengefasst, welche aus sechs Artikeln besteht ([6], [7], [14], [23], [30], [32]). Diese behandeln lediglich die allgemeine graphische Repräsentation von unterschiedlichen Daten und gingen auch in die Ableitung der Anforderungen an Sichten mit ein (siehe 3.1).

2.2 Gefundene Literaturquellen

In diesem Kapitel werden die unterschiedlichen Artikel, welche sich mit der Visualisierung von Sichten beschäftigen, betrachtet. Dabei werden die unterschiedlichen Quellen einander gegenübergestellt, indem die Artikel anhand verschiedener Ge-

sichtspunkte betrachtet werden. Es ist teilweise schwierig die unterschiedlichen Ansätze miteinander zu vergleichen, da sie von unterschiedlichen Ansatzpunkten ausgehen. Es gibt zu diesem Thema nicht sehr viel Literatur, da sich die meisten nur mit unterschiedlichen Metriken beschäftigen und weniger mit verschiedenen Sichten, also der rollengebundenen Darstellung von Messdaten.

Die einzelnen Ansätze werden im Anschluss nach folgenden Schema beschrieben:

1. Motivation und Ansatz: beschreiben, aus welchem Grund die unterschiedlichen Unternehmen damit begonnen haben, ein Messprogramm einzuführen. Ebenso wie den Ansatz, den die Unternehmen für die Entwicklung eines Programms zu Grunde gelegt haben, z.B. GQM. Ein Messprogramm bedeutet in diesem Zusammenhang, dass regelmäßig Projektdaten erfasst und visualisiert werden, um ein Projekt zu überwachen und den Prozess zu verbessern.

2. Darstellungsarten: beschreibt, welche unterschiedlichen Diagramm-Typen für die Visualisierung der Daten verwendet wurden.

3. Einteilung der Metriken: beschreibt welche Einteilung in Kategorien vorgenommen wurde, um die unterschiedlichen Metriken zu klassifizieren und einzuordnen, soweit eine solche vorgenommen wurde. Es wird dabei keine Aussage darüber gemacht, ob auch Metriken aus allen Kategorien in dem Programm verwendet werden. Die verwendeten Metriken werden mittels GQM abgeleitet, sofern dieser Ansatz verwendet wurde.

4. Sichten: beschreibt die verwendeten Sichten, wobei die Sichten fortlaufend nummeriert werden, um diese in späteren Abschnitten zu referenzieren. Es wird hierbei beschrieben, welche Daten dargestellt werden. In eine Sichteninstanz können sowohl externe als auch interne Dateneinträge eingehen.

2.2.1 Software Measurement Guidebook

Im „Software Measurement Guidebook" [2] (SMG) wird der Zweck und die wichtige Rolle von Messungen diskutiert, im Besonderen die Aktivitäten und Prozesse eines Messprogramms.

Motivation und Ansatz:

Die Organisation Software Engineering Laboratory (SEL) wurde 1976 gegründet, um die Effektivität von Software Engineering Techniken zu untersuchen und die gewonnen Ergebnisse auch umzusetzen, und wird bei seiner Arbeit von der National Aeronautics und Space Administration (NASA) mit unterstützt und finanziert. Die Erfahrungen von über 100 größeren Projekten wurden vom Office of Safety and Mission zusammengefasst und dokumentiert, um anderen Unternehmen eine Anleitung zu geben, mit Hilfe derer sie ein Messprogramm einrichten können, bzw. auch ein existierendes verbessern können. Den Unternehmen soll vor allem nahe gebracht werden, dass Software Messungen ein Mittel zum Zweck sind und kein Selbstzweck. Es gibt drei Hauptgründe, welche dafür sprechen, dass Messungen durchgeführt werden sollen, diese sind: Verstehen von Software, Durchführen von Verbesserungen von Prozessen und die Leitung von Softwareprojekten.

Um die Sichten zu definieren, wurde der GQM-Ansatz (siehe 1.7) gewählt. Dafür ist es sehr wichtig, dass die Ziele der Organisation, welche ein solches Programm durchführen will, bekannt sind und verstanden werden, um anhand derer Erwartungen bzw. Ziele zu definieren, welche erreicht werden sollen. Zusätzlich müssen Verantwortlichkeiten festgelegt werden, unter anderem wer für die Datenerfassung zuständig ist. Am Ende werden die entsprechenden Metriken ausgewählt, welche helfen, die gesetzten Ziele zu erreichen, und Sichten definiert.

Darstellungsarten:

Um die unterschiedlichen Daten zu visualisieren, wurden nur einfache Diagramm-Typen verwendet. Abhängig von den Metriken wird entweder ein Säulen-Diagramm (Balken-Diagramm, siehe 1.8.2), ein Linien-Diagramm (siehe 1.8.1) oder ein Kreis-Diagramm (siehe 1.8.3) verwendet, um die Daten darzustellen.

Einteilung der Metriken:

Die Metriken lassen sich in folgende fünf Hauptkategorien einteilen: Kosten, Fehler, Prozesscharakteristiken, Projektdynamik und Projektcharakteristiken. Die am meisten genutzten und auch akzeptierten Metriken beschäftigen sich mit Kosten, darunter fällt unter anderem der Aufwand der einzelnen Entwickler. Die zweitwichtigste Kategorie beschäftigt sich mit Daten, die sich mit den Fehlern beschäftigen, hierzu gehört unter anderem die Fehlerklasse und der Aufwand, welcher mit der Fehlerbehebung verbunden ist. Metriken der Kategorie Prozesscharakteristiken beschäftigen sich mit den Eigenschaften eines Prozesses, darunter fallen z.B. die verwendeten Programmiersprachen oder auch verwendete Methoden und Werkzeuge. Die Gruppe der Projektdynamiken befasst sich unter anderem mit den unterschiedlichen Änderungen, sowohl Änderungen in den Anforderungen, als auch Änderungen, welche durch Fehlerbehebung notwendig sind. Außerdem zählen die angepassten Schätzungen der endgültigen Projektcharakteristiken zu dieser Gruppe. Die letzte Kategorie der Projektcharakteristiken enthält Metriken, welche den Zustand und die Eigenschaften des endgültigen Projekts beschreiben, hierzu zählt z.B. der Gesamtaufwand und die Größe des Projekts.

Sichten:

Es wird eine Reihe von unterschiedlichen Sichten definiert, wobei die verwendeten Metriken nicht immer in nur eine einzelne Kategorie fallen und die Sichten auch für unterschiedliche Ziele verwendet werden. Die unterschiedlichen Sichten sollen Beispiele für die Unternehmen sein, welche Sichten möglich und sinnvoll sind. Dies sind aber auf keinen Fall die einzigen Sichten, welche angewendet werden sollen.

- Sicht 1: *Aufwandsverteilung*: Es wird die Verteilung des Gesamtaufwands auf unterschiedliche Phasen bzw. Aktivitäten dargestellt (siehe Anhang A.1, Figure 2-2).
- Sicht 2: *Verteilung auf die Fehlerklassen*: Darstellung der Verteilung der Fehler auf die unterschiedlichen Fehlerklassen, zusätzlich aufgeteilt auf die verwendeten Programmiersprachen (siehe Anhang A.1, Figure 2-3).

- Sicht 3: *Wachstum des Kodes*: Es wird dargestellt, wie der Kode über die Zeit, zusätzlich verteilt auf Phasen, wächst (siehe Anhang A.1, Figure 2-4).
- Sicht 4: *Änderungen im Kode*: Darstellung der Gesamtzahl aller Änderungen normiert auf 1000 Zeilen Kode, dargestellt über die relative Zeit (siehe Anhang A.1, Figure 2-5).
- Sicht 5: *Beobachtung der Wachstumsrate:* Es wird das geplante relative Wachstum des Kodes, zusammen mit oberen und unteren Schranken, mit der tatsächlichen Wachstumsrate verglichen (siehe Anhang A.1, Figure 2-7).
- Sicht 6: *Verwendete Programmiersprachen*: Darstellung, welche Programmiersprachen verwendet werden (siehe Anhang A.1, Figure 6-1).
- Sicht 7: *Entwicklung in der Wiederverwendung von Kode*: Es wird die relative Anzahl von Kodezeilen, welche in anderen Projekten wiederverwendet werden, für jedes Projekt aufgetragen. Hierbei wird für unterschiedliche Programmiersprachen und Zeiträume ein eigenes Diagramm verwendet (siehe Anhang A.1, Figure 6-2).
- Sicht 8: *Kosten gegenüber Größe*: Darstellung der Produktivität (in Lines per Hour) gegenüber der relativen Wiederverwendung von Kodezeilen, sowohl für die Gesamtanzahl der Kodezeilen, als auch für die Zeilen, des neu entwickelten Kodes. Alternativ wird die Summe aus der Anzahl der neu entwickelten Kodezeilen und der gewichteten Anzahl der wiederverwendeten Kodezeilen pro Stunde gegenüber den Projekten und unterschiedlichen Gewichtungsfaktoren aufgetragen (siehe Anhang A.1, Figure 6-3, Figure 6-4).
- Sicht 9: *Kosten der Hauptaktivitäten*: Es wird die Verteilung der Gesamtkosten auf verschiedene Aktivitäten dargestellt (siehe Anhang A.1, Figure 6-7).
- Sicht 10: *Fehlerraten*: Darstellung des Verhältnisses zwischen den Fehlern pro 1000 Zeilen Kode und den unterschiedlichen Phasen (siehe Anhang A.1, Figure 6-8).
- Sicht 11: *Fehler gegenüber Größe und Komplexität*: Es wird das Verhältnis zwischen den Fehlern pro Zeile und der Komplexität bzw. der Anzahl der Zeilen aufgetragen (siehe Anhang A.1, Figure 6-10).
- Sicht 12: *Beobachtung der Software-Änderungen*: Es wird sowohl die zu Beginn eines Projektes geschätzte Anzahl von den aufsummierten Änderungen pro 1000 Zeilen Kode (zusammen mit einem Toleranzbereich) gegenüber den Phasen aufgetragen, als auch die tatsächliche summierte Anzahl der Änderungen (siehe Anhang A.1, Figure 6-14).
- Sicht 13: *Beobachtung des Aufwands des Personals*: Es wird der ursprüngliche Plan des Aufwands an Personal (Anzahl der Vollzeitbeschäftigten mit 40 Stunden pro Woche) zusammen mit evtl. existierenden angepassten Plänen und dem tatsächlichen Aufwand verglichen (siehe Anhang A.1, Figure 6-15).
- Sicht 14: *Beobachtung des Fortschritts beim Testen*: Darstellung der Anzahl aller gefundenen Fehler zusammen mit der Anzahl behobener Fehler und aller noch existierenden Fehler über die Zeit (siehe Anhang A.1, Figure 6-16).

- Sicht 15: *Beobachtung der Fehler*: Es wird die geschätzte Gesamtanzahl aller Fehler zusammen mit einer oberen und unteren Schranke und der tatsächlichen Anzahl über die Zeit dargestellt (siehe Anhang A.1, Figure 6-17).

2.2.2 Collecting and Analyzing Web-Based Project Metrics

Der Artikel „Collecting and Analyzing Web-Based Project Metrics" [5] (Web-Based) beschreibt die Gründe für die Firma Agilent, welche zur Einführung eines Messprogramms geführt haben und die Erfahrungen der Manager und den resultierenden Schritten für die Zukunft. Agilent wurde Ende 1999 als selbständige Firma gegründet und war Teil der strategischen Neuordnung von Hewlett-Packard.

Motivation und Ansatz:

Die Abteilung für Telekommunikationssysteme der Firma Agilent benutzt als Ausgangspunkt zur Beobachtung von Fehlerverteilung und Änderungen ein Werkzeug zum Konfigurationsmanagement. Dadurch ergibt sich die interessante Situation, dass die verschiedenen Messungen bereits durchgeführt werden, und nur noch geringere Investitionen notwendig sind, um ein Messprogramm darauf aufzusetzen. Es gibt verschiedene Gründe, welche dafür sprechen, dass ein solches Programm eine sinnvolle Investition ist. Diese sind die Charakterisierung von Prozessen und Produkten, ebenso wie die Evaluation vom Status eines Projektes, da hierdurch die eigene Organisation besser verstanden werden kann und nur so Verbesserungen möglich sind. Zu Beginn erwartete Agilent, dass dadurch bessere Vorhersagen getroffen werden können, so dass Projekte besser geplant werden können und dies zu Verbesserungen von Software Engineering Prozessen führt.

Für das Messprogramm wurden keine weiteren Messungen und Metriken eingeführt, sondern nur die bereits existierenden weiter verwendet und auch visualisiert. Es wurde erst nach Abschluss einer Testphase beschlossen, weitere Metriken zu verwenden und zu implementieren.

Darstellungsarten:

Für die Repräsentation der Daten werden hauptsächlich zwei Diagramm-Typen verwendet, welche auch die am häufigsten verwendeten sind, nämlich Balken-Diagramme (siehe 1.8.2) und Linien-Diagramme (siehe 1.8.1). Jedoch wird auch eine Tabelle (siehe 1.8.6) verwendet, um gewisse Daten dem Benutzer zu präsentieren.

Einteilung der Metriken:

Die Metriken werden auf der einen Seite danach beurteilt, wie diese zu erfassen sind, d.h. es kann sich um direkt messbare Metriken handeln, wie z.B. die Anzahl der ausstehenden Fehler, als auch um komplexe oder abgeleitete Metriken handeln. Ein Beispiel für eine abgeleitete Metrik ist die Anzahl der unkommentierten Ausdrücke im Kode abhängig von Entwickler und Monat. Auf der anderen Seite wird die gleiche Einteilung vorgenommen wie bereits von SEL (siehe 2.2.1), also eine Einteilung in die fünf Kategorien: Kosten, Fehler, Prozesscharakteristiken Projektdynamik und Projektcharakteristiken. Obwohl eine grundsätzliche Einteilung der Metriken stattfindet, werden keine Metriken aus allen Kategorien verwendet.

Sichten:

Es wurden drei Sichten verwendet und implementiert; dies ergibt sich bereits dadurch, dass die bereits vorhandenen Messungen weiter benutzt werden sollen. Es sollte nur ein einfacher Prototyp ohne großen Aufwand entwickelt werden, erst im Anschluss sollte anhand der Erfahrungen über eine Erweiterung entschieden werden. Seit dem Abschluss der Erprobung wird ein robustes Management-Werkzeug entwickelt, welches um weitere Metriken erweitert werden soll (beispielsweise Kode-Metriken). Weitere Ergebnisse sind noch nicht vorhanden. Die verwendeten Sichten werden im Folgenden aufgelistet.

- Sicht 16: *Fehler über die Zeit*: Es wird die Gesamtzahl aller gefundenen Fehler, aller behobenen Fehler und der noch übrigbleibenden Fehler pro Projekt über die Zeit dargestellt (siehe Anhang A.2, Figure 4).

- Sicht 17: *Fehlerursachen*: Darstellung der Anzahl der Fehler verteilt auf die unterschiedlichen Fehlerklassen pro Projekt (siehe Anhang A.2, Figure 5).

- Sicht 18: *Dauer bis zur Fehlerbehebung*: Es wird die minimale, maximale und mittlere Dauer bis zur Fehlerbehebung für verschiedene Projekte dargestellt, wobei eine zusätzliche Unterteilung der Fehler nach Priorität („Wie schnell muss dieser Fehler behoben werden?") stattfindet (siehe Anhang A.2, Table 1 und Table 2).

2.2.3 A Practical View of Software Measurement and Implementation Experiences Within Motorola

„A Practical View of Software Measurement and Implementation Experiences Within Motorola" [1] (Motorola) beschreibt das Messprogramm von Motorola, hierfür werden auch die verwendeten Grundlagen erläutert.

Motivation und Ansatz:

Motorola entschloss sich aus mehreren Gründen, Metriken und Sichten einzusetzen. Hierzu gehört einerseits, dass dadurch ein besseres Verständnis für den Entwicklungsprozess entsteht. Andererseits sollen notwendige Änderungen festgelegt werden, damit eine Verbesserung der Qualität und der Produktivität stattfindet, und dadurch die Konkurrenzfähigkeit zu anderen Firmen erhalten bleibt. Um dieses Ziel zu erreichen, wurde der GQM Ansatz (siehe 1.7), wie auch von SEL, verwendet. Hierzu wurden sieben Ziele definiert, welche Motorola erreichen wollte: Verbesserung der Projektplanung, Verbesserung der Beherrschung von Fehlern, Verbesserung der Zuverlässigkeit von Software, Verringerung der Fehlerdichte von Software, Verbesserung des Kundenservices, Kostenreduzierung von Nichtübereinstimmungen und Erhöhung der Produktivität. Diese dienen als Ausgangspunkt für die Ableitung der Metriken, werden jedoch nicht anhand der fünf Parameter von GQM-Zielen definiert. Es wurde außerdem zu Beginn eine eigene Arbeitsgruppe gegründet, deren Aufgabe darin bestand mittels GQM eine minimale Menge von Metriken zu definieren und auszuwählen.

Darstellungsarten:

Neben der beiden Diagramm-Typen Linien-Diagramm (siehe 1.8.1) und Balken-Diagramm (siehe 1.8.2) wird auch eine Art Tabelle verwendet. Die Tabelle wird jedoch nur einmal verwendet, um den Fortschritt der Aktivitäten im Vergleich zur Planung eines Projektes darzustellen.

Einteilung der Metriken:

Motorola verwendet eine andere Einteilung für die Metriken, als die bereits zuvor erwähnten. Die Metriken werden abhängig von ihrem Hauptzweck in folgende drei Kategorien eingeteilt: Prozess-, Projekt- und Produktmetriken. Prozessmetriken werden dazu benutzt, den Prozess der Softwareentwicklung und Wartung zu verbessern, hierzu zählt z.B. die Effektivität von Inspektionen und Tests. Projektmetriken dienen dazu, ein Projekt zu beobachten und zu verbessern, wie z.B. die Anzahl der Entwickler oder die Verteilung des Aufwands auf die unterschiedlichen Projektphasen. Die letzte Kategorie der Produktmetriken soll helfen, Verbesserung des Produktes zu erreichen, hierzu gehören z.B. Metriken, welche die Komplexität des Entwurfs beschreiben.

Sichten:

Im Folgenden werden die unterschiedlichen Sichten, welche von Motorola verwendet werden, aufgelistet und erläutert.

- Sicht 19: *Genauigkeit von Schätzungen*: Es wird das Verhältnis zwischen den geschätzten und tatsächlichen Werten des Aufwands und der Projektdauer über die Zeit dargestellt (siehe Anhang A.3, Chart 9).
- Sicht 20: *Effektivität der Fehlerbeherrschung*: Darstellung des Verhältnis zwischen der Anzahl der gefundenen Fehler (relativ zur Gesamtzahl der Fehler) und der geplanten Zielkurve über die Zeit (siehe Anhang A.3, Chart 7).
- Sicht 21: *Effektivität der Fehlerbeherrschung pro Phase*: Darstellung des Verhältnis zwischen der Anzahl der gefundenen Fehler (relativ zur Gesamtzahl der Fehler) verteilt auf die einzelnen Phasen (siehe Anhang A.3, Chart 8).
- Sicht 22: *Qualität des Softwareentwicklungsprozesses*: Es wird das Verhältnis von Fehlern, welche in der gleichen Phase gefunden werden, in der sie verursacht wurden, zur entsprechenden Kodegröße und auch das Verhältnis der restlichen Fehler zur entsprechenden Kodegröße, dargestellt (siehe Anhang A.3, Chart 1).
- Sicht 23: *Qualität der freigegebenen Software*: Darstellung des Verhältnis zwischen der Anzahl der Fehler in der freigegebenen Software und der entsprechenden Kodegröße, und das Verhältnis von Fehlern, welche durch inkrementale Entwicklung verursacht wurden, zur entsprechenden Kodegröße (siehe Anhang A.3, Chart 2).
- Sicht 24: *Fehler gefunden durch Benutzer*: Darstellung des Verhältnis zwischen der Anzahl der durch den Benutzer gefundenen Fehler und der entsprechenden

Kodegröße, und das Verhältnis von Fehlern, welche sowohl durch inkrementale Entwicklung verursacht wurden als auch durch den Benutzer gefunden wurden, zur entsprechenden Kodegröße (siehe Anhang A.3, Chart 3).

- Sicht 25: *Gemeldete Fehler nach der Freigabe von Software*: Es wird die Anzahl neuer gemeldeter Fehler innerhalb eines Monats und die Gesamtanzahl aller noch nicht gelösten Fehler am Ende eines Monats über die Zeit dargestellt (siehe Anhang A.3, Chart 4).
- Sicht 26: *Alter der gemeldeten Fehler nach der Freigabe von Software*: Darstellung des mittleren Alter von noch offenen Fehlern, bzw. des mittleren Alter von bereits behobenen Fehlern, wobei jeweils nur die Fehler betrachtet werden, welche nach der Freigabe der Software aufgetreten sind, über die Zeit (siehe Anhang A.3, Chart 5).
- Sicht 27: *Kosten zur Behebung von Fehlern freigegebener Software*: Es werden nur die Kosten zur Behebung von Fehlern, welche nach der Freigabe noch vorhanden sind, dargestellt (siehe Anhang A.3, Chart 6).
- Sicht 28: *Software Produktivität*: Darstellung des Verhältnis zwischen der Kodegröße und dem Aufwand (siehe Anhang A.3, Chart 10).
- Sicht 29: *Fortschritt des Projektes*: Es wird der ursprüngliche Zeitplan (Dauer der einzelnen Phasen/Aktivitäten) zusammen mit der tatsächlichen und der weiterhin geplanten Dauer dargestellt (siehe Anhang A.3, Fig. 7).
- Sicht 30: *Kosten und Verdienst*: Darstellung der geschätzten Gesamtkosten, des Budgets, der tatsächlichen Kosten und des Verdienstes (siehe Anhang A.3, Fig. 8).
- Sicht 31: *Änderungen der Anforderungen*: Es wird die Anzahl der Änderungen in den Anforderungen über die Zeit aufgetragen, sowohl die Anzahl der neuen Änderungen, als auch die Summe der Änderungen der bereits vergangenen Zeit und der Zukunft (geplant) dargestellt (siehe Anhang A.3, Fig. 9).
- Sicht 32: *Ursachen der Änderungen in den Anforderungen*: Es wird die Anzahl der Änderungen in den Anforderungen, verteilt auf die unterschiedlichen Ursachen, dargestellt (siehe Anhang A.3, Fig. 10).
- Sicht 33: *Abgedeckte Anforderungen*: Es wird sowohl die Summe aller Anforderungen über die Zeit, als auch die Summe der implementierten Anforderungen dargestellt (siehe Anhang A.3, Fig. 11).
- Sicht 34: *Verteilung auf die Fehlerklassen*: Darstellung der absoluten Häufigkeit der Fehler aufgeteilt auf die unterschiedlichen Fehlerklassen (siehe Anhang A.3, Fig. 12).
- Sicht 35: *Verbleibende Fehler*: Darstellung der vorhandenen Fehler über die Zeit (sowohl Vergangenheit, als auch Zukunft) (siehe Anhang A.3, Fig. 13).
- Sicht 36: *Effektivität des Review*: Darstellung der Fehlerdichte im Quellkode innerhalb des Projektes pro Review zusätzlich zu oberen und unteren Schranken (siehe Anhang A.3, Fig. 14).

- Sicht 37: *Unterteilung der nichtbehobenen Fehler in Prioritäten bzw. Gewichtung*: Es wird die Anzahl von noch nichtbehobenen Fehlern gegenüber der unterschiedlichen Gewichtung und der Zeit dargestellt (siehe Anhang A.3, Fig. 15).
- Sicht 38: *Alter der nichtbehobenden Fehler*: Es wird das Alter der noch nichtbehobenen Fehlern gegenüber der unterschiedlichen Priorität und der Zeit dargestellt (siehe Anhang A.3, Fig. 16).

2.2.4 Visualizing Software Changes

Der Artikel „Visualizing Software Changes" [4] (VSC) stellt verschiedene Arten vor, wie Änderungen visualisiert werden können, hierzu werden viele verschiedene Diagrammtypen benutzt, um die Vorteile der verschiedenen Diagrammtypen miteinander zu kombinieren. Außerdem werden weitere Mittel zur Darstellung von Zusammenhängen über verschiedene Diagramme hinweg benutzt, wie z.B. gleiche Farben für gleiche Daten, um damit den Benutzer zu unterstützen, dass er die Daten in den Diagrammen ohne lange Suche einander zuordnen kann.

Motivation und Ansatz:

Ein großes Problem beim Software Engineering besteht darin, Änderungen des Kodes zu verfolgen, da häufig Änderungen notwendig sind und diese sowohl komplex und teuer sind, als auch schwer zu verfolgen. Ursachen für Änderungen gibt es mehrere, dazu gehört sowohl das Hinzufügen neuer Funktionalität (perfective), auch verursacht durch wachsende Benutzeranforderungen, als auch die Anpassung an neue Hardware und an neue Betriebssysteme (adaptive) und natürlich die Fehlerbehebung (corrective). Im idealen Fall sollte es einfach sein, zukünftige Änderungen durchzuführen, in der Realität ist oft bereits die Architektur fehlerhaft, da durch einen engen Zeitplan und hohe Kosten Kompromisse erzwungen werden. Dies führt dazu, dass viel Zeit und Aufwand auf die Änderungen entfällt, denn diese sind komplex, kostenintensiv sowie schwer zu beherrschen und zu verstehen.

Das Ziel ist, die Daten, welche sich aus Änderungen in der Software ergeben, auf verschiedene Arten darzustellen und dem Benutzer zu präsentieren, so dass Strukturen erkannt werden können und dadurch ein erhöhtes Verständnis für Änderungen entsteht.

Darstellungsarten:

Im Gegensatz zu den restlichen Ansätzen wird eine Vielzahl von Visualisierungsarten benutzt, um die Daten darzustellen, d.h. ein und dieselben Daten werden unterschiedlich visualisiert. Es handelt sich hierbei nicht nur um zweidimensionale Diagramme, sondern auch um dreidimensionale. Im Folgenden werden die unterschiedlichen Diagrammtypen, welche neben Balken- und Kreisdiagrammen verwendet werden, näher erläutert.

- Matrix: Eine Matrix-Darstellung (siehe 1.8.4) bietet sich an, wenn Daten anhand von zwei Parametern dargestellt werden sollen (ein Parameter entspricht der Zeile und der zweite der Spalte). Eine zusätzliche Größe kann durch die Farbe oder durch die Balkenbreite dargestellt werden.

- Cityscape: Eine Cityscape-Darstellung ist im Grunde genommen ein dreidimensionales Balkendiagramm oder auch eine Erweiterung in die dritte Dimension einer Matrix. Neben den beiden Parameter in der Ebene, gibt es noch zusätzlich die Höhe und Farbe als weitere Parameter.
- Datenblätter (Tabelle, siehe 1.8.6): Die einfachste Art Daten darzustellen, besteht darin, diese in Tabellenform textuell aufzulisten.
- Netzwerk: Bei einem Netzwerk (siehe 1.8.5) handelt es sich um eine Möglichkeit, wie Strukturen innerhalb von Software dargestellt werden können, wobei dies jedoch nur für einfache Zusammenhänge gilt.

Neben den verschiedenen Diagrammtypen, werden noch weitere Mittel benutzt, um Zusammenhänge hervorzuheben. Hierzu gehört u.a. die Verwendung von gleichen Farben über verschiedene Diagramme hinweg, und die Benennung von verschiedenen Punkten.

Einteilung der Metriken:

Es findet keine explizite Einteilung der Metriken statt, da bereits von Beginn an nur Metriken benutzt werden, welche sich mit den Änderungen beschäftigen. Es wurde mehr auf die Ausnutzung vieler Möglichkeiten zur Visualisierung von Daten Wert gelegt. Die einzige Einteilung welche vorgenommen wurde, betrifft die Art der Änderung: Es wird unterschieden in adaptive, korrigierende und Änderungen zur Unterstützung der Wartung. Unter adaptiven Änderungen versteht man die Änderungen, welche durchgeführt werden, um neue Funktionalität zum System hinzuzufügen, oder um die Software an neue Hardware anzupassen. Korrigierende Änderungen umfassen diejenigen Änderungen, welche zur Fehlerbehebung dienen. Die letzte Art der Änderung werden dadurch verursacht, dass die Wartung der Software vereinfacht werden soll, wichtig dabei ist, dass die Änderungen nicht dazu dienen, neue Funktionalität hinzuzufügen, oder um Fehler zu beheben.

Hierbei ist zu beachten, dass diese Definition der Arten von Änderungen abweicht, von der Definition aus [9]. Dort werden als adaptive Änderungen nur diejenigen bezeichnet, die auf Änderungen der Hardware zurückzuführen sind. Als korrigierende Änderungen werden die aufgefasst, welche zur Fehlerbehebung und Wartung durchgeführt werden. Die Anpassung an neue Anforderungen sind perfektive Änderungen.

Sichten:

Es werden nur Metriken und Sichten verwendet, um die Änderungen von Software zu verstehen, diese werden nun im Anschluss aufgezählt. Die aufgelisteten Sichten, werden nicht alle in einem Diagramm dargestellt, sondern in einer Menge von Diagrammen und unterschiedlichen Typen, jedoch gleichzeitig.

- Sicht 39: *Statistische Angaben*: Es werden die verschiedenen statistischen Daten einer Änderung dargestellt, hierzu gehört die Anzahl der Änderungen, die durch Bearbeitung entstanden sind, in verschiedenen Dateien, die zugehörige Versionsnummer, Anzahl der hinzugefügten bzw. gelöschten Kodezeilen und die Anzahl der aktiven Entwickler (siehe Anhang A.4, Figure 1).

- Sicht 40: *Änderungen zeitlich geordnet*: Darstellung der Änderungen über die Zeit, Änderungen über den Dateityp und Jahr und die durch Bearbeitung entstandenen Änderungen geordnet nach Modulen und Subsystemen (siehe Anhang A.4, Figure 2).

- Sicht 41: *Änderungen pro Entwickler*: Darstellung der Anzahl der Änderungen aufgeteilt auf die Entwickler, zusammen mit der Anzahl der hinzugefügten bzw. gelöschten Kodezeilen (siehe Anhang A.4, Figure 3).

- Sicht 42: *Größe der Änderungen sortiert nach Freigabe*: Es wird die Anzahl der hinzugefügten und gelöschten Zeilen von Änderungen für die unterschiedlichen freigegebenen Versionen dargestellt, einmal nach Größe und einmal nach Versionen sortiert (siehe Anhang A.4, Figure 4).

- Sicht 43: *Aktivitäten sortiert nach Entwickler und Softwaremodule*: Darstellung der Anzahl der Änderungen pro Entwickler und pro Softwaremodul und Darstellung der Gesamtzahl der Änderungen pro Entwickler über alle Module (siehe Anhang A.4, Figure 5 und Figure 6).

- Sicht 44: *Umfang der Änderungen*: Es wird dargestellt, wie viele Softwareeinheiten (z.B. Module, Dateien) von einer Änderungen betroffen sind, wobei zusätzlich noch dargestellt wird, wie häufig die gleichen Einheiten von ein und derselben Änderungen betroffen sind (siehe Anhang A.4, Figure 7).

2.2.5 Software Management Environment (SME) - Components and Algorithms

Im „Software Management Environment" [3] werden die Komponenten und Algorithmen des Software Management Environment erläutert. Es handelt sich hierbei um ein Management-Werkzeug, welches die Funktionen, die dem Projekt zur Projektbeobachtung, Schätzung und Planung zu Grunde liegen beschreibt.

Motivation und Ansatz:

Das interaktive Werkzeug Software Management Environment (SME) wurde unter anderem von SEL gesponsort, um die Möglichkeit zu haben, Projekte verfolgen zu können und deren Status zu evaluieren. Im Gegensatz zu dem Artikel SMG (siehe 2.2.1) handelt es sich um ein Werkzeug, während im Rahmen des SEL eine Anleitung beschrieben wird, anhand derer jedes Unternehmen sein eigenes Messprogramm entwickeln kann. In diese Anleitung gehen unter anderem die Erfahrungen, die mit dem SME gemacht wurden, mit ein. Das Werkzeug soll in erster Linie Software-Manager bei ihrer Aufgabe unterstützen, hierzu gehört, dass der Fortschritt des Projektes beobachtet und mit anderen Projekten verglichen werden soll. Weitere Aufgaben sind der Aufwandsvergleich, ebenso wie der Vergleich mit einem so genannten normalen Verhalten von Projekten, d.h. es werden Projekte mit dem Verhalten von ähnlichen Projekten verglichen. Außerdem sollen Vorhersagen getroffen werden und Aussagen über das Projekt gemacht werden, hierfür müssen die Schwachstellen und Stärken analysiert werden.

Darstellungsarten:

Es werden drei Typen von Diagrammen verwendet, wobei es sich ausschließlich um zweidimensionale Diagramme handelt. Es werden Balken- (siehe 1.8.2) und Linien-Diagramme (siehe 1.8.1) verwendet, abhängig von der Art der Daten und welcher Diagrammtyp geeigneter ist, bzw. als geeigneter angesehen wird. Zusätzlich werden die Daten in einer Tabelle (siehe 1.8.6) dargeboten, wenn die Daten geeignet sind für diese Art der Darstellung.

Einteilung der Metriken:

Es wurde keine explizite Einteilung der Metriken vorgenommen, sondern allein eine Menge von acht Metriken ausgewählt, welche verwendet wurden. Weiterhin wurden vier so genannte Profile verwendet, welche unter anderem dazu dienen sollen, die Stabiliät und Zuverlässigkeit eines Projekts zu untersuchen.

Sichten:

Der Artikel befasst sich hauptsächlich mit dem Werkzeug und weniger mit unterschiedlichen Sichten, d.h. die einzelnen Komponenten werden ausführlich beschrieben. Lediglich für die einzelnen Metriken und Profile werden sowohl Daten als auch Modelle graphisch dargestellt. Diese werden im Folgenden aufgelistet, wobei keine getrennte Auflistung für Daten und Modelle stattfindet. Vielmehr dient eine Sicht nicht nur zur Darstellung von den realen Daten, sondern auch zur Darstellung des zugehörigen Modells, welches den geplanten theoretischen Verlauf repräsentiert (z.B. Baseline), wobei beide nicht gleichzeitig in einem Diagramm dargestellt werden.

- Sicht 45: *Aufwand*: Es wird der Gesamtaufwand eines Projektes über die Zeit dargestellt (siehe Anhang A.5, Figure 2-8).
- Sicht 46: *Kodezeilen*: Darstellung der Gesamtanzahl von Kodezeilen eines Projektes über die Zeit (siehe Anhang A.5, Figure 2-9).
- Sicht 47: *Anzahl der Module*: Es wird die Gesamtanzahl aller Module eines Projektes über die Zeit dargestellt (siehe Anhang A.5, Figure 2-10).
- Sicht 48: *CPU-Nutzung*: Es wird die Gesamtanzahl der Computerbenutzung in CPU Stunden eines Projektes, über die Zeit dargestellt (siehe Anhang A.5, Figure 2-11).
- Sicht 49: *Computer-Durchläufe*: Darstellung der Nutzung des Computers in Hinblick auf die Anzahl aller Aufgaben eines Projekts, welche auf allen Rechnern erledigt werden, über die Zeit (siehe Anhang A.5, Figure 2-12).
- Sicht 50: *Änderungen von Modulen*: Es wird die Gesamtzahl aller Änderungen in Modulen eines Projekts dargestellt. Diese Anzahl kann automatisch erfasst werden, indem die Anzahl der Versionen abzüglich der initialen Anzahl gezählt wird (siehe Anhang A.5, Figure 2-13).
- Sicht 51: *Gemeldete Änderungen*: Im Gegensatz zur zuvor genannten Sicht, werden in dieser Sicht die gemeldeten Änderungen eines Projekts über die

Zeit dargestellt, wobei diese nicht automatisch erfasst werden können, da eine Änderung mehrere Module betreffen kann (siehe Anhang A.5, Figure 2-14).

- Sicht 52: *Gemeldete Fehler*: Darstellung der gemeldeten Fehler eines Projektes über die Zeit (siehe Anhang A.5, Figure 2-15).

- Sicht 53: *Aufwand, um Änderungen zu isolieren*: Es wird der summierte Aufwand, welcher benötigt wird, um Änderungen zu isolieren, dargestellt, wobei eine zusätzliche Unterteilung in fünf Kategorien, abhängig von der Dauer für eine Änderung, stattfindet. Diese fünf Kategorien sind: weniger als eine Stunde, zwischen einer Stunde und einem Tag, zwischen einem Tag und drei Tagen, mehr als drei Tage und unbekannte Dauer (siehe Anhang A.5, Figure 2-17).

- Sicht 54: *Aufwand, um Änderungen zu implementieren*: Darstellung des summierten Aufwands, welcher benötigt wird, um Änderungen zu implementieren. Wobei eine zusätzliche Unterteilung in fünf Kategorien, abhängig von der Dauer für eine Änderung, stattfindet. Diese fünf Kategorien sind: weniger als eine Stunde, zwischen einer Stunde und einem Tag, zwischen einem Tag und drei Tagen, mehr als drei Tage und unbekannte Dauer (siehe Anhang A.5, Figure 2-18).

- Sicht 55: *Aufwand, um Fehler zu isolieren*: Es wird der summierte Aufwand, welcher benötigt wird, um Fehler zu isolieren, dargestellt, wobei eine zusätzliche Unterteilung in fünf Kategorien, abhängig von der Dauer für eine Fehler, stattfindet. Diese fünf Kategorien sind: weniger als eine Stunde, zwischen einer Stunde und einem Tag, zwischen einem Tag und drei Tagen, mehr als drei Tage und unbekannte Dauer (siehe Anhang A.5, Figure 2-19).

- Sicht 56: *Aufwand, um Fehler zu korrigieren*: Darstellung des summierten Aufwands, welcher benötigt wird, um Fehler zu korrigieren, wobei eine zusätzliche Unterteilung in fünf Kategorien, abhängig von der Dauer für eine Korrektur, stattfindet. Diese fünf Kategorien sind: weniger als eine Stunde, zwischen einer Stunde und einem Tag, zwischen einem Tag und drei Tagen, mehr als drei Tage und unbekannte Dauer (siehe Anhang A.5, Figure 2-20).

2.3 Zusammenfassung der Recherche-Ergebnisse

In diesem Abschnitt wird eine zusammenfassende Übersicht über die verschiedenen Ansätze und der jeweiligen Menge an Sichten und Metriken gegeben.

Der erste Artikel gibt eine Anleitung für Unternehmen, die ein eigenes Messprogramm einführen wollen, daher sind die Sichten und Metriken allgemein und damit auch einsetzbar. Es sind auch mehrere Sichten und Metriken definiert, da das Ziel des Artikels auch darin liegt, eine Menge von Metriken und Sichten zu definieren, die benutzt werden können. Erst in einem weiteren Schritt, wenn die Unternehmen ihre Erfahrungen gesammelt haben, können sie weitere Metriken und Sichten definieren, die ihren eigenen Zielen entsprechen.

Der nächste Artikel ist ein Erfahrungsbericht der Firma Agilent, die erst das bereits vorhandene Management zur Beobachtung der Fehlerverteilung und Änderungen benutzt und nach positiven Erfahrungen beschließt, ihr Messprogramm auszubauen. Daher werden nur wenige Metriken und Sichten verwendet.

	Ansatz	Anzahl der Sichten	Anzahl der Metriken
SMG	GQM	15	15
Web-Based	Verwendung bereits vorhandener Metriken	3	3
Motorola	GQM	20	26
VSC	Betrachtung der Eigenschaften von Änderungen aus verschiedenen Blickwinkeln	6	11
SME	Metriken durch Expertenwissen bestimmt	12	12

Tabelle 1: Übersicht über die unterschiedlichen Quellen.

Motorola entwickelte ein eigenes Messprogramm, das auch eingesetzt wurde, und leitete anhand der eigenen Ziele Metriken und Sichten mit dem GQM-Ansatz ab. Durch die systematische Herleitung ergeben sich relativ viele Metriken (im Vergleich zu den übrigen Artikeln), außerdem wurden mehrere Metriken in einer Sicht zusammengefasst, d.h. es gibt mehr Metriken als Sichten.

Im nächsten Artikel, der sich mit der Visualisierung von Änderungen beschäftigt, werden bedeutend weniger Sichten und Metriken verwendet, als bei Motorola. Dies ergibt sich bereits dadurch, dass nur Metriken aus einer Kategorie verwendet werden. Außerdem werden die Daten in einem Diagramm, jedoch auf unterschiedliche Art und Weise, z.B. gleichzeitig als Balkendiagramm und als Tabelle, dargestellt. Hierdurch werden auch mehrere Metriken in einer Sicht verwendet, und es sind weniger Sichten notwendig.

Der letzte Artikel stellt ein Management-Werkzeug vor, welches von anderen Firmen eingesetzt werden kann. Die Metriken werden als die wichtigsten Metriken beschrieben, jedoch ohne Angabe einer Grundlage. Für jede dieser Metriken wird eine eigene Sicht zur Visualisierung verwendet.

Anhand der Tabelle kann man erkennen, dass die Anzahl der Metriken und damit auch der Sichten abhängig vom Ansatz ist. Durch einen allgemeinen Ansatz (z.B. GQM) werden mehr Metriken betrachtet, und damit auch mehr Sichten zur Darstellung benötigt. Außerdem ist die Anzahl der Sichten abhängig von der Anzahl der unterschiedlichen Kategorien der Metriken. So deckt der Ansatz von Motorola mehrere Kategorien ab, z.B. Aufwand, Fehler usw. und daher gibt es auch 20 Sichten.

3 Sichten-Metamodell

In diesem Kapitel soll ein Meta-Modell für Sichten entwickelt werden, mit welchem alle Zusammenhänge zwischen Sichten dargestellt werden können. Zur Erinnerung: eine Sicht ist die graphische Darstellung von Messdaten. Hierfür werden die Erkenntnisse, welche aus den gefunden Sichten und allgemeinen Artikeln über Visualisierung abgeleitet wurden, zusammengefasst, wobei die Erkenntnisse in unterschiedliche Kategorien eingeteilt werden. Zuerst werden einige Anforderungen an Sichten abgeleitet und erläutert und in einem weiteren Abschnitt wird eine Klassifikation der Sichten vorgenommen. Anschließend wird ein UML-Modell der Zusammenhänge entwickelt.

3.1 Anforderungen an Sichten

Anhand von einigen allgemeinen Artikeln und auch bereits vorhandenen Sichteninstanzen, konnten Anforderungen an Sichten abgeleitet werden. Diese werden im Folgenden unter jeweils einem Schlagwort zusammengefasst und ausführlich beschrieben.

3.1.1 Übersichtlichkeit

In einer Sicht sollten nur wenige Informationen dargestellt werden, da es ansonsten für den Betrachter der Sichteninstanz schwierig wird, zu erkennen, welche Informationen wichtig sind, um eine Entscheidung über weiteres Vorgehen zu treffen. Wird der Aufwandsverlauf mehrerer Projekte dargestellt, zusätzlich zu Ober- und Unterschranken für den Aufwand, so ist es bereits bei zwei Projekten schwierig, die Kurven und jeweiligen Schranken einander zu zuordnen. Bei noch mehr Kurven ist es sogar unmöglich, die richtige Zuordnung vorzunehmen. Soll mit einer solchen Sicht beispielsweise überprüft werden, ob die Grenzen des Aufwands überschritten werden und wie der Vergleich mit anderen Projekten ausfällt, so besteht allein schon die Schwierigkeit darin, die richtige Kurve und zugehörigen Grenzen zu bestimmen. Daher ist es auch sehr schwierig eine Aussage über die Einhaltung der Grenzen zu treffen. So ist es sinnvoller, nur eine Vergleichskurve darzustellen, z.B. der Mittelwert über alle Projekte, zusätzlich zu den Schranken der aktuellen Kurve.

Eine Sicht sollte daher übersichtlich sein.

3.1.2 Verteilung

Aus der vorangegangenen Anforderung kann eine weitere abgeleitet werden, nämlich die Verteilung der Daten auf mehrere Diagramme, wenn ansonsten eine zu große Anzahl von Informationen zugleich dargestellt werden würde. Ein Beispiel hierfür ist es, die Darstellung des Aufwands auf unterschiedliche Phasen und Aktivitäten zu verteilen. Würden diese Daten alle in einer Sichteninstanz dargestellt werden, so besteht die Schwierigkeit einerseits darin, eine Achse mit den Aktivitäten aller Phasen zu beschriften, und andererseits darin, dass der Aufwand einer der Phasen zu einem derart großen Wert führen könnte, dass die Werte für eine Aktivität im Vergleich zu den Werten von Phasen zu niedrig sind, um noch erkennbar zu sein. Werden jedoch in einer Instanz der Aufwand über die Phasen und in einer anderen

der Aufwand gegenüber den Aktivitäten einer einzigen Phase aufgetragen, so werden beide Probleme gelöst. Werden diese noch untereinander verknüpft, dass es möglich ist zu navigieren, ist die Vergleichsmöglichkeit ohne großen Aufwand gegeben.

Daher sollten die Daten gegebenenfalls auf mehrere Diagramme verteilt werden.

3.1.3 Vergleichsmöglichkeit

Um den Betrachter bei der Entscheidungsfindung zu unterstützen, ist es sinnvoll, diesem eine geeignete Basis anzubieten, anhand derer er die Werte beurteilen kann. Dies kann auf unterschiedliche Art und Weise geschehen. So ist es möglich, ältere Werte zum Vergleich mit einzutragen oder auch den geplanten Verlauf eines Wertes. Hierbei ist jedoch zu beachten, dass bei den älteren Werten auch wirklich ein Zusammenhang zu den aktuellen derart besteht, dass die Projekte Gemeinsamkeiten haben, also dass sie auch über eine gemeinsame Basis verfügen. Nur in diesem Fall sind Aussagen aufgrund eines Vergleichs auch begründet. Eine weitere Art, dem Betrachter eine Unterstützung zu geben, bietet sich vor allem bei der Aufschlüsselung eines Ganzen an, z.B. kann man die Gesamtheit der Fehler auf unterschiedliche Fehlerklassen verteilen. Bei dieser Vergleichsmöglichkeit verwendet man meist Kreisdiagramme, da dem Betrachter sofort klar ist, dass es sich hierbei um eine vollständige Verteilung handelt, und dass es keine weiteren Einteilungen gibt, welche nicht aufgelistet sind. In diesem Fall ist auch weniger die Angabe von absoluten Werten interessant, sondern vielmehr die Angabe von Prozentzahlen, da in diesem Fall nicht die Gesamtzahl betrachtet wird, sondern nur die Verteilung auf die unterschiedlichen Klassen.

Eine Sicht sollte daher Möglichkeiten zum Vergleich von Daten bieten.

3.1.4 Hervorhebung

Bei vielen Sichten soll untersucht werden, ob ein Wert oder Verlauf mit einem anderen übereinstimmt, oder ob zu große Abweichungen existieren. Um den Betrachter bei seiner Aufgabe zu unterstützen, sollten diese Punkte bereits in der Sicht hervorgehoben werden, damit nach kurzer Zeit zumindest die problematischen Punkte erkannt werden können. Dies heißt für den Fall von oberen und unteren Schranken, dass diejenigen Punkte hervorgehoben werden, die nicht innerhalb dieser Schranken liegen, also entweder zu niedrig oder zu hoch sind. Für den Fall, dass nur eine Vergleichskurve existiert, z.B. der Verlauf eines vergangenen Projektes, ist es sehr unwahrscheinlich, dass die Kurven exakt übereinstimmen und von daher ist es nicht sinnvoll alle Punkte zu markieren, welche nicht exakt mit den anderen übereinstimmen. Vielmehr sollten nur die Punkte hervorgehoben werden, deren Abweichung größer als ein zu Beginn festgelegter Wert ist.

In einer Sicht sollten Abweichungen hervorgehoben werden.

3.1.5 Skalierung

Bei der Darstellung sollte keine feste Größe angenommen werden, sondern vielmehr sollte diese abhängig von der Anzahl der jeweiligen Messdaten sein. Ansonsten kann dies dazu führen, dass die Daten nicht alle darstellbar sind, oder gestaucht werden müssen, damit der Platz ausreicht. Letzteres kann dazu führen, dass die einzelnen Daten nicht mehr erkennbar sind. Wird die Größe an die Anzahl der darzustellenden Daten angepasst, kann die Sicht sowohl wenige Daten als auch viele visualisieren, so dass diese jeweils noch durch den Benutzer interpretiert werden können. Die einfachste Art zu gewährleisten, dass die Sicht mit verschiedenen Mengen von Daten zurechtkommt, ist die Verwendung eines Kreisdiagramms, da sich dort nur die Gesamtanzahl und Verhältnisse ändern. Eine andere Art der Skalierung kann auf die Anforderung der „Verteilung" (siehe 3.1.2) zurückgeführt werden, indem bei einer zu großen Anzahl von Daten diese auf mehrere Diagramme verteilt werden.

Eine Sicht sollte mit den Daten skalieren.

3.1.6 Robustheit

Vor allem zu Beginn eines Projektes stehen noch nicht alle Daten zur Verfügung. So stehen die konkreten Aufwandsdaten einer Phase erst zur Verfügung, wenn das Projekt in die Phase eingetreten ist und bereits Aufwände erfasst wurden. Somit stehen noch nicht alle Daten zur Verfügung, welche in die Sicht eingetragen werden sollen. Allerdings sollte die Sicht trotzdem dargestellt werden, da diese bereits bei der Durchführung des Projektes benötigt werden, um dieses zu kontrollieren und nicht erst nach Abschluss des Projektes.

Eine Sicht sollte auch darstellbar sein, wenn Informationen fehlen.

3.1.7 Konsistenz

Ein wichtiger Punkt ist, dass die verschiedenen Sichten konsistent sein sollten. Dies beinhaltet sowohl die Konsistenz innerhalb einer Sicht (interne Konsistenz), als auch die Konsistenz verschiedener Sichten zueinander (externe Konsistenz). Zu der internen Konsistenz gehört, dass die Daten nicht widersprüchlich sind, z.B. darf die Summe einzelner Werte nicht kleiner sein, als ein einzelner Wert. Es ist jedoch auch wichtig, dass nicht nur eine Sicht in sich konsistent ist, sondern auch dass verschiedene Sichten sich nicht gegenseitig widersprechen. Ein Beispiel hierfür wäre einerseits die Darstellung der Aufwände für die Aktivitäten und die Darstellung der akkumulierten Aufwände für die einzelnen Phasen andererseits.

Eine Sicht sollte sowohl zu anderen Sichten konsistent sein, als auch zu sich selbst.

3.1.8 Zielorientierte Darstellung

Die Darstellung von Messdaten sollte zielorientiert, also abhängig vom Objekt, vom Qualitätsaspekt, vom Zweck (zweckorientiert), Kontext (kontextorientiert) und von der Rolle (rollenorientiert) sein. Zweckorientiert bedeutet in diesem Zusammenhang, dass berücksichtigt wird, welche Aufgabe mit Hilfe der Sicht unterstützt werden soll. Wenn z.B. die Projektüberwachung unterstützt werden soll, sollten unter

anderem Abweichungen vom geplanten Aufwand hervorgehoben werden. Was die rollenorientierte Darstellung betrifft, werden nur die Informationen berücksichtigt, welche für die Rolle wichtig sind. Ein Projektmanager benötigt beispielsweise nicht die Daten jedes einzelnen Entwicklers, sondern eher einen Überblick über alle Daten eines Projektes. Dagegen benötigt ein einzelner Entwickler, um seine eigene Arbeit einzuschätzen, auch seine eigenen Daten im Vergleich zu allen Daten. Zusätzlich ist noch der Kontext des Projekts zu beachten, also ob das Projekt beispielsweise an der Universität oder in einem Unternehmen durchgeführt wird.

Eine Sicht sollte die Messdaten zielorientiert darstellen.

3.1.9 Variabilität/Anpassbarkeit

Um eine Sicht in vielen Bereichen einsetzbar zu machen, ist es notwendig, dass diese variabel ist und an die jeweiligen Anforderungen des Benutzers angepasst werden kann. Dabei kann es sich sowohl um Anpassungen an die Organisation handeln, als auch an den Projektkontext. Dies ist wichtig, da verschiedene Organisation eine unterschiedliche Art und Weise haben, bestimmte Sachverhalte darzustellen. Auch in der gleichen Organisation werden für unterschiedliche Projekte unterschiedliche Darstellungsweisen verwendet. Damit diese bereit sind, ein Werkzeug wie den SP-CC einzusetzen, darf dieses nicht eine eigene Darstellungsform verwenden, ohne die Möglichkeit, diese zu verändern. Unternehmen haben ihre eigene Darstellungsform schon über längere Zeit erprobt und sind im Allgemeinen damit zufrieden.

Eine Sicht sollte anpassbar sein.

3.2 Klassifikation von Sichten

Um die Sichten genauer zu beschreiben und zu bewerten, wird eine Klassifikation für Sichten vorgestellt. Hierbei werden die wichtigsten Punkte, anhand welcher die Sichten zu unterscheiden sind, aufgelistet und erläutert. Diese Punkte umfassen z.T. die Anforderungen aus dem vorangegangenen Abschnitt. So ist es wichtig, die Rollen zu kennen, für welche die Sicht von Interesse ist, um zu beurteilen, ob die Darstellung der Sicht zielorientiert ist, oder nicht. Auch die möglichen Darstellungsarten können für diese Aussage benutzt werden. In der Auflistung der möglichen Vergleichsdaten findet sich auch die Anforderung der „Vergleichsmöglichkeit", und auch die Anforderung der „Hervorhebung", da ohne Vergleichsdaten keine Hervorhebung von abweichenden Werten möglich ist. Wenn keine Vergleichsdaten aufgelistet sind, ist auch die Vergleichsmöglichkeit nicht gegeben. Mit dem Punkt der Anpassbarkeit wird die Anforderung der Variabilität abgedeckt.

Die Frage nach der Übersichtlichkeit einer Sicht kann bis zu einem gewissen Grad beantwortet werden, indem man die verwendeten Metriken betrachtet. Die Metriken geben an, welche Daten und wieviele dargestellt werden, um daraus zu schätzen wie übersichtlich die resultierende Sicht sein kann: bei sehr vielen Metriken mit vielen verschiedenen Daten, ist es wahrscheinlich, dass diese weniger übersichtlich ist. Es ist jedoch auch eine allgemeine Beschreibung zur Beantwortung der Frage notwendig. Ob die Daten auf mehreren Diagrammen verteilt dargestellt werden, wenn deren Anzahl zu groß ist, kann fast nur anhand der Implementierung entschieden werden, wobei die allgemeine Beschreibung einen Anhaltspunkt liefern kann.

Die Anforderungen „Skalierung” und „Robustheit” können nicht ohne die genaue Implementierung überprüft werden, da nur die resultierenden Diagramme Auskunft darüber geben, ob diese beispielsweise skaliert werden.

Ob die Sichten konsistent sind, kann für die interne Konsistenz noch anhand des Diagramms überprüft werden, jedoch für die externe Konsistenz fehlt das Wissen, ob den unterschiedlichen Sichten die gleichen Daten zu Grunde liegen.

Als zusätzlicher Punkt, mit welchem Sichten beschrieben werden können, ist die Auflistung der referenzierten Sichten. Dies dient unter anderem dazu, die Zusammenhänge genauer zu beschreiben und bereits miteinander verwandte Sichten zu kennzeichnen. Nun folgt die Klassifikation, deren Herleitung erläutert wurde.

Rollen:

Für jede Sicht ist es wichtig zu wissen, welche Rollen einen Nutzen von dieser Sicht haben können, daher werden in diesem Abschnitt die jeweiligen Rollen aufgelistet.

Metriken:

Wichtig bei der Klassifikation von Sichten sind die verwendeten Metriken, welche der Sicht zu Grunde liegen, d.h. welche durch die Sicht visualisiert werden. Eine Metrik kann dabei von verschiedenen Sichten unterschiedlich visualisiert werden und umgekehrt kann eine Sicht mehrere Metriken visualisieren.

Vergleichsdaten:

Eine Anforderung an Sichten ist die Vergleichsmöglichkeit der Daten. Mit diesem Punkt kann beschrieben werden, ob in dieser Sicht Vergleichsdaten verwendet werden und auf welcher Basis diese beruhen. So gibt es die Möglichkeit, die bereits vorhandenen Daten von vorangegangenen Projekten zu verwenden, oder diese zu schätzen.

Darstellungsarten:

Für jede Sicht sind nicht alle Diagrammtypen geeignet, daher werden hier alle diejenigen Typen aufgelistet, die grundsätzlich geeignet sind, die Daten darzustellen, dies macht jedoch keine Aussage darüber, welcher Typ auch eingesetzt wird. Es kann für die Darstellung einer dieser Diagrammtypen verwendet werden.

Anpassbarkeit:

Hier wird beschrieben, ob die Sicht angepasst werden kann. Das beste Beispiel für eine Anpassung besteht darin, Parameter zu verwenden und abhängig vom Wert des Parameters die Darstellung anzupassen.

Referenzierte Sichten:

Um den Zusammenhang zwischen verschiedenen Sichten zu verdeutlichen, werden alle Sichten, welche mit der aktuellen verwandt sind, aufgelistet. Hierzu zählen auch Sichten, welche eine andere verfeinern. Diese Sichten sollten sich gegenseitig in der Implementierung referenzieren.

3.3 Meta-Modell

Das Meta-Modell dient dazu, die Zusammenhänge, welche zwischen den verschiedenen Sichten bestehen können, zu verdeutlichen und auch, um auf einen Blick zu erläutern, aus welchen Bestandteilen eine Sicht besteht.

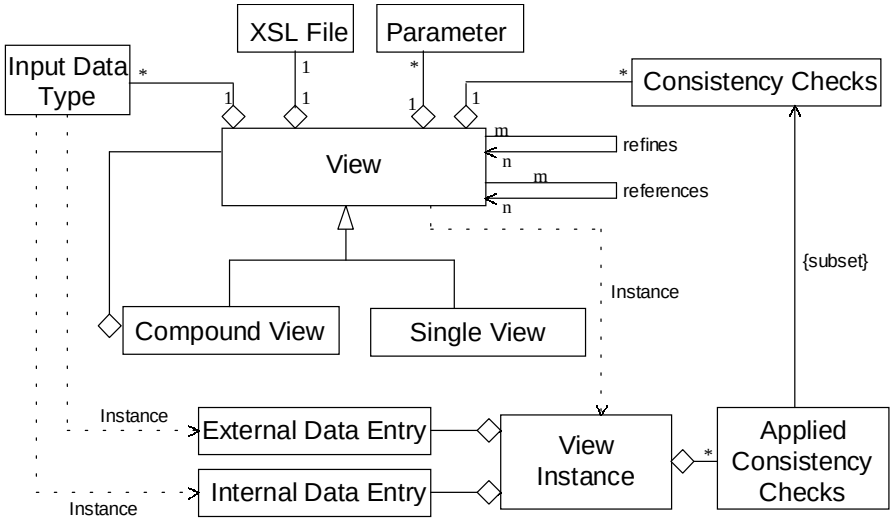


Abbildung 22: Sichten-Meta-Modell als UML-Diagramm.

Abbildung 22 zeigt das Sichten-Meta-Modell als ein UML-Diagramm, wobei das Composite Pattern benutzt wurde, um darzustellen, dass eine Sicht einerseits aus einer einzelnen Sicht bestehen kann (*Single View*) und andererseits aus mehreren Sichten zusammengesetzt sein kann (*Compound View*).

Jede Sicht besteht aus einer Menge von Eingaben (*Input Data* Type), Parametern (*Parameter*) und Konsistenzchecks (*Consistency Checks*). Letztere beschreiben Anforderungen, die an eine Sicht gestellt werden. Weiterhin besteht eine Sicht aus einer XSL-Datei (*XSL File*), in welcher die Implementierung stattfindet.

Eine Sicht kann andere Sichten sowohl verfeinern als auch referenzieren. Auch wenn hier eine Unterscheidung stattfindet, geschieht dies im SPCC über den gleichen Mechanismus.

In die Instanz einer Sicht (*View Instance*) gehen verschiedene Dateneinträge (*External Data Entry* bzw. *Internal Data Entry*) ein. Diese entsprechen den Eingaben der Sicht. Außerdem wird ein Teil der Konsistenztests in einer Instanz angewendet (*Applied Consistency Checks*).

3.4 Bewertung der gefundenen Sichten

In diesem Abschnitt werden die Sichten aus der Literatur bewertet. Die Sichten wurden als Anhang eingefügt, sortiert nach den Artikeln und entsprechend der Nummerierung. Die Bewertung geschieht anhand der Anforderungen, die in Abschnitt 3.1 erläutert wurden. In der Bewertung wird jedoch komplett auf einen Teil der Anforderungen verzichtet, zu diesen Anforderungen zählen: Variabilität, zielorientierte Darstellung, Konsistenz und Robustheit. Der Grund hierfür liegt darin, dass zur Beantwortung dieser Fragen die Artikel nicht ausreichen; die Sichten müssten im Einsatz untersucht werden. Auch wenn die Frage nach der Skalierung nicht immer beantwortet werden kann, wird diese Anforderung mit in die Bewertung aufgenommen, da es Fälle gibt, in denen die Skalierung dadurch gegeben ist, dass ein Kreisdiagramm verwendet wird und somit die Anzahl der Daten keine Rolle mehr spielen. Außerdem ist zu beachten, dass sich das Vorhandensein einer Vergleichsmöglichkeit und die Hervorhebung von abweichenden Werten bis zu einem gewissen Grad beeinflussen. Wenn keine Vergleichsmöglichkeit gegeben ist, können auch keine Werte bestimmt werden, die zu stark abweichen.

Die Tabellen geben jeweils nur einen einfachen Überblick über die Eigenschaften der verschiedenen Sichten. Es wird in jedoch noch einmal in einem zusätzlichen erklärenden Abschnitt pro Artikel auf die Besonderheiten eingegangen und diese genauer erläutert.

3.4.1 Bewertung der Sichten: „Software Measurement Guidebook”

Wie in Tabelle 2 zu sehen ist, wurde in den Sichten komplett auf die Hervorhebung von Abweichungen verzichtet, obwohl in den meisten Fällen Vergleichsdaten vorliegen. Als Vergleichsdaten werden häufig geplante Verläufe (oder auch Mittelwert oder Schranken) benutzt. Ansonsten werden auch häufig Kreisdiagramme verwendet, welche zumindest einen guten Vergleich der Teilmengen unterstützen. Fast alle Sichten sind sehr übersichtlich, mit zwei Ausnahmen (Sicht 11 und 13), in denen die Daten in den Diagrammen durch Punkte repräsentiert werden und die Punkte nicht mehr den Werten zugeordnet werden können. Sofern notwendig, wurden die Daten auf mehrere Diagramme verteilt. Die Verteilung auf mehrere Diagramme war häufig nicht notwendig, da nur wenig Daten gleichzeitig dargestellt werden. Über die Skalierung kann entweder keine Aussage gemacht werden, oder es handelt sich um ein Kreisdiagramm (Sicht1, Sicht 2, Sicht 6, Sicht 9), welches sowohl mit vielen als auch wenigen Daten zurechtkommt, da sich nur die Gesamtanzahl und die Verhältnisse ändern.

3.4.2 Bewertung der Sichten: „Collecting and Analyzing Web-Based Project Metrics”

Wie in Tabelle 3 zu sehen ist, wurde - wie bereits zuvor - komplett auf die Hervorhebung von abweichenden Werten verzichtet, obwohl zumindest für alle Sichten Vergleichsdaten existieren. Hierbei ist jedoch festzuhalten, dass für die Sicht 16 zwar Vergleichsdaten existieren, es jedoch nicht sinnvoll ist hierbei von Abweichungen oder ähnlichem zu sprechen, da nur die Fehler nach „gefunden”, „behoben” und „verbleibend” unterteilt werden. Es ist nicht möglich Aussagen über die Skalierung

Sicht	Übersicht-lichkeit	Verteilung	Vergleich	Hervor-hebung
1	ja	nicht nötig, da nur wenige Daten	Teilmengenvergleich	nein
2	ja	Programmiersprache	Teilmengenvergleich	nein
3	ja	nicht nötig	nein	nein
4	ja	nicht nötig	nein	nein
5	ja	nicht nötig	Plankurve und Bereich	nein
6	ja	zwei Kreisdiagramme	Wartung ↔Entwicklung	nein
7	ja	Programmiersprache	Jahre; Mittelwert, Sprachen	nein
8	ja	nicht nötig	verschiedene Projekte	nein
9	ja	nicht nötig	Teilmengenvergleich	nein
10	ja	nicht nötig	Vorhersage	nein
11	+/-	nicht nötig	Mittelwert	nein
12	ja	nicht nötig	Schranken	nein
13	+/-	nicht nötig	initial, angepasste Pläne	nein
14	ja	nicht nötig	gefunden, behoben, verbleibend	nein
15	ja	nicht nötig	Oberschranke, Unterschranke	nein

Tabelle 2: Bewertung der Sichten „Software Measurement Guidebook".

Sicht	Übersicht-lichkeit	Verteilung	Vergleich	Hervor-hebung
16	ja	nicht nötig	gefunden, behoben, verbleibend	nein
17	ja	nicht nötig	versch. Fehlerklassen; Projekte	nein
18	ja	zwei Tabellen	versch. Projekte	nein

Tabelle 3: Bewertung der Sichten „Collecting and Analyzing Web-Based Project Metrics".

der Sichten zu machen. Allerdings sind die Sichten alle übersichtlich und auf mehrere Diagramme bzw. Tabellen verteilt, sofern dies notwendig ist, also die Anzahl der Daten zu groß ist, um in einer Sicht dargestellt zu werden (Sicht 18).

3.4.3 Bewertung der Sichten „A Practical View of Software Measurement and Implementation Experiences Within Motorola"

Auch Motorola verzichtete in ihren Sichten komplett auf die Hervorhebung von abweichenden Werten (siehe Tabelle 4), obwohl fast immer Vergleichskurven vorhanden sind und somit die Grundvoraussetzung gegeben ist. Ebenso wird auf die Verteilung auf mehrere Diagramme verzichtet. Letzteres ist jedoch auch für keine der Sichten notwendig, da die meisten sehr übersichtlich sind und nur so viele Da-

Sicht	Übersicht-lichkeit	Verteilung	Vergleich	Hervor-hebung
19	ja	nicht nötig	durch Verhältnis, Zielkurve	nein
20	ja	nicht nötig	Zielkurve	nein
21	ja	nicht nötig	verschiedene Projekte	nein
22	ja	nicht nötig	sofort oder später gefunden	nein
23	ja	nicht nötig	Fehler, durch Wartung verursachte Fehler	nein
24	ja	nicht nötig	wie 23, nur durch Benutzer gefunden	nein
25	ja	nicht nötig	neue Fehler, Gesamtzahl Fehler	nein
26	ja	nicht nötig	gelöst, bestehend	nein
27	ja	nicht nötig	nein	nein
28	ja	nicht nötig	Gesamt-, Teilaufwand	nein
29	ja	nicht nötig	geplant, tatsächlich	nein
30	ja	nicht nötig	geschätzt, tatsächliche Kosten; Verdienst	nein
31	ja	nicht nötig	Summe, neu	nein
32	ja	nicht nötig	verschiedene Ursachen	nein
33	ja	nicht nötig	Summe, erfüllt	nein
34	ja	nicht nötig	verschiedene Fehlerklassen	nein
35	ja	nicht nötig	innerhalb des Projekts	nein
36	ja	nicht nötig	Ober- und Unterschranke	nein
37	+/-	nicht nötig	verschiedene Projektwochen	nein
38	+/-	nicht nötig	verschiedene Projektwochen	nein

Tabelle 4: Bewertung der Sichten „A Practical View of Software Measurement and Implementation Experiences Within Motorola".

ten darstellen, wie gut in einem Diagramm zu erkennen sind. Die letzten beiden Diagramme sind zwar weniger übersichtlich, der Grund hierfür liegt jedoch nicht an der Menge der Daten, sondern daran, dass es schwierig ist, die Daten den jeweiligen Wochen zu zuordnen. Das Problem würde sich bereits dadurch lösen lassen, dass die Balken, die die gleichen Wochen repräsentieren, die gleiche Farbe oder das gleiche Muster aufweisen, statt allen Balken die gleiche Farbe zu geben. Es kann für keine der Sichten eine Aussage über die Skalierung getroffen werden, da die Daten weder als Kreisdiagramm dargestellt werden, noch die Sichten im praktischen Einsatz untersucht werden konnten.

3.4.4 Bewertung der Sichten: „Visualizing Software Changes"

Die Sichten des Artikels „Visualizing Software Changes" sind fast alle sehr unübersichtlich (siehe auch Tabelle 5), da sie vollgepackt mit Daten sind, und diese dadurch überladen wirken. Die Darstellung der Daten in einer Matrix (siehe auch 1.8.4) führt auch nicht zu einer Erhöhung der Übersichtlichkeit. Die Daten wurden jedoch fast immer auf mehrere Diagramme verteilt und in unterschiedlichen Arten von Diagrammen dargestellt, wobei durch die Menge der Daten keine Erhöhung der Übersichtlichkeit erreicht wird. Dies sollte jedoch das Ziel sein, wenn die Daten auf

Sicht	Übersichtlichkeit	Verteilung	Vergleich	Hervor-hebung
39	ja	2 Diagramme	neu; gelöscht	nein
40	zu viele Daten	mehrere Diagramme	verschiedene Jahre	nein
41	zu viele Daten	mehrere Diagramme	verschiedene Entwickler	nein
42	zu viele Daten	mehrere Diagramme	verschiedene Versionen	nein
43	zu viele Daten	mehrere Diagramme	verschiedene Module, Entwickler	nein
44	+/-	nein	nein	nein

Tabelle 5: Bewertung der Sichten „Visualizing Software Changes".

verschiedene Diagramme verteilt werden.

Jede der ersten fünf Sichten beinhaltet eine Darstellung der Daten in einer Tabelle. Durch diese Tabelle ist eine Unterstützung gegeben, um wenigstens die genauen Werte zu erfahren, wenn diese nicht aus den anderen Diagrammen abgelesen werden können. Die Daten der letzten Sicht werden nur in einem Netzwerk (siehe 1.8.5) dargestellt, wobei die Anzahl der Verbindungen noch zu überschauen ist, jedoch können allein die Verbindungen, die eine andere Farbe besitzen, gut identifiziert werden.

Über die Skalierung kann für keine Sicht eine allgemeine Aussage getroffen werden, wobei auch Kreisdiagramme verwendet werden und für diese die Skalierung kein Problem ist.

3.4.5 Bewertung der Sichten: „Software Management Environment (SME) - Components and Algorithms"

Die Sichten des Artikels sind relativ einfach gehalten und verzichten komplett auf eine Hervorhebung (siehe Tabelle 6), da jedoch für die Sichten 45 bis 52 keine Vergleichsdaten verwendet werden ist die Möglichkeit der Hervorhebung gar nicht erst gegeben. Durch die Einfachheit sind die Sichten auch sehr übersichtlich und eine Verteilung auf mehrere Diagramme ist durch die wenigen darzustellenden Daten auch gar nicht nötig.

Es gibt nur für die letzten vier Sichten eine Möglichkeit zum Vergleich, wobei sich dieser Vergleich darauf beschränkt, die Daten in fünf Gruppen einzuteilen, diese sind: Aufwand beträgt bis zu einer Stunde, zwischen einer Stunde und einem Tag, zwischen einem und drei Tagen, mehr als drei Tage oder unbekannt. Daher ist auch für diese Sichten eine Hervorhebung von abweichenden Daten nicht sinnvoll. Außerdem kann für keine der Sichten eine Aussage über die Skalierung gemacht werden.

Sicht	Übersicht-lichkeit	Verteilung	Vergleich	Hervor-hebung
45	ja	nicht nötig	nein	nein
46	ja	nicht nötig	nein	nein
47	ja	nicht nötig	nein	nein
48	ja	nicht nötig	nein	nein
49	ja	nicht nötig	nein	nein
50	ja	nicht nötig	nein	nein
51	ja	nicht nötig	nein	nein
52	ja	nicht nötig	nein	nein
53	ja	nicht nötig	$\leq$1h, $\leq$1d, $\leq$3d, >3d, unbekannt	nein
54	ja	nicht nötig	$\leq$1h, $\leq$1d, $\leq$3d, >3d, unbekannt	nein
55	ja	nicht nötig	$\leq$1h, $\leq$1d, $\leq$3d, >3d, unbekannt	nein
56	ja	nicht nötig	$\leq$1h, $\leq$1d, $\leq$3d, >3d, unbekannt	nein

Tabelle 6: Bewertung der Sichten „Software Management Environment (SME) - Components and Algorithms".

4 Herleitung möglicher Sichten

Für die Herleitung von Sichten werden zuerst die unterschiedlichen Rollen eines Software-Entwicklungsprozesses erläutert, eine Auswahl getroffen, welche Rollen im Weiteren betrachtet werden und bei welchen Aufgaben diese unterstützt werden. Im nächsten Abschnitt werden die Metriken beschrieben, welche für die Auswahl der Sichten eine Rolle spielen und von den jeweiligen Sichten benutzt werden. Im letzten Abschnitt werden die Sichten beschrieben, welche in den anderen Abschnitten hergeleitet wurden, und erläutert, welche Rollen und Metriken davon jeweils betroffen sind.

4.1 Rollen

In einem Software-Entwicklungsprozess fallen eine Vielzahl von Aufgaben an, die auf so genannte Rollen logisch aufgeteilt werden (siehe Skript zur Vorlesung Software Engineering 1 [9]). Die unterschiedlichen Rollen werden in verschiedene Kategorien eingeteilt. Auf der einen Seite stehen die technischen Entwicklerrollen und auf der anderen Seite die organisatorischen und managementorientierten Rollen. Feinere Einteilung sind möglich (siehe Abbildung 23). Im Folgenden werden zuerst die organisatorischen und managementorientierten Rollen und anschließend die technischen Rollen betrachtet.

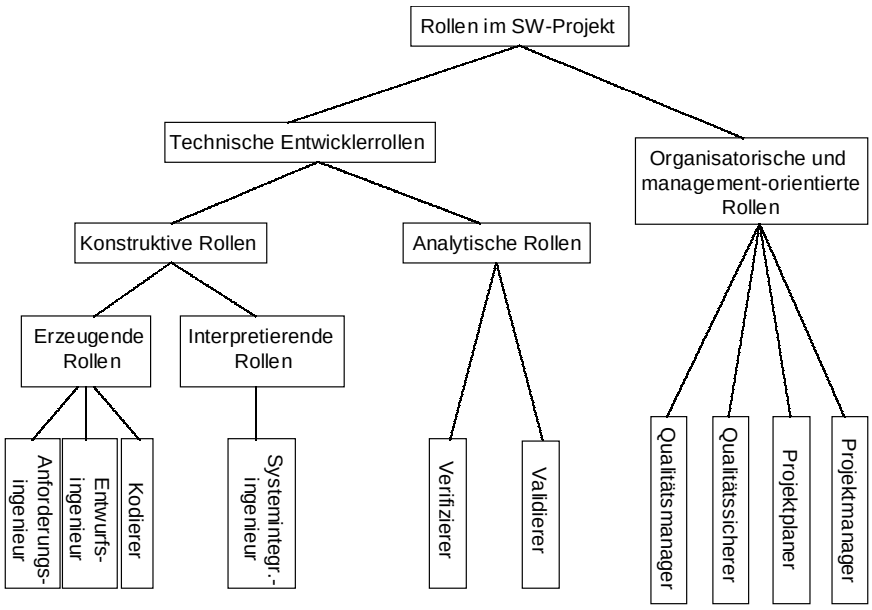


Abbildung 23: Klassifikation der Rollen eines Software-Entwicklungsprozesses.

Ein *Projektplaner* erstellt aus einer Menge von Projektzielen und Projekteigenschaften einen Projektplan, der während der Entwicklung gegebenenfalls korrigiert und angepasst werden muss.

Ein *Projektmanager* muss sicherstellen, dass das Projekt entsprechend dem Projektplan und der Menge der zur Verfügung stehenden Ressourcen durchgeführt wird. Sollte dies nicht mehr sichergestellt sein, muss er außerdem Gegenmaßnahmen einleiten, um dies zu korrigieren und damit wieder die Einhaltung des Projektplans zu erreichen.

Ein *Qualitätssicherer* hat die Aufgabe, die fertigen Produkte und Prozesse zu kontrollieren. Dadurch soll die Einhaltung der Ziele bezüglich der Qualität, die im Projektplan definiert wurden, sichergestellt werden, um dadurch das Vertrauen in deren Erfüllung zu festigen.

Ein *Qualitätsmanager* verwaltet die Erfahrungen und das Wissen von verschiedenen Projekten, um dies für laufende Projekte zu nutzen, sowohl in deren Planung als auch deren Ausführung. Das Ziel hierbei ist, dass eine organisationsweite Verbesserung erreicht wird.

Neben den organisatorischen und managementorientierten Rollen gibt es die bereits erwähnten technischen Entwicklerrollen, wobei auch hier noch eine Einteilung nach unterschiedlichen Aufgaben stattfindet. Alle Produkte, die von den Entwicklern erstellt werden, müssen dabei vorgegebenen Dokumentationsrichtlinien entsprechen.

Der *Anforderungsingenieur* erstellt aus einer vom Kunden gegebenen Problembeschreibung eine so genannte Problem-Lösungsdokumentation, welche u.a. die Benutzeranforderungen, die Entwickleranforderungen und die Verfolgbarkeitsmatrix umfasst.

Eine weitere technische Entwicklerrolle ist der *Entwurfsingenieur der System-Architektur*. Dieser erstellt ausgehend von der Problem-Lösungsdokumentation eine Software-Lösungsdokumentation. Hierzu zählt nicht nur der Systementwurf und die Verfolgbarkeitsmatrix für den Entwurf, sondern auch die Komponentenanforderungen und die Testfälle für die Komponenten und die Integration.

Weiterhin gibt es den *Entwurfsingenieur für Daten und Algorithmen*, welcher anhand der Komponentenanforderungen eine Beschreibung für die Komponenten erstellt, die gewisse Qualitätsanforderungen erfüllen müssen.

Anhand der Komponentenbeschreibung erstellt der *Kodierer* den Komponentenkode und die Software-Komponenten-Dokumentation. Dabei müssen vorgegebene Qualitätsanforderungen, wie z.B. *Information Hiding* oder *Kommentierungsgüte* eingehalten werden.

Die Aufgabe eines *Verifizierers* besteht darin, für jede Abstraktionsstufe zwischen der Problembeschreibung und dem Komponentenkode, die Übereinstimmung mit der nächsthöheren Abstraktionsstufe (Ausnahme Problembeschreibung), ebenso wie die Einhaltung der vorgegebenen Dokumentationsrichtlinien und die Qualität zu überprüfen.

Der *Systemintegrationsingenieur* konstruiert ausgehend von der Software-System-Beschreibung ein ablauffähiges Software-System, welches die Qualitätsanforderungen, die in den nicht-funktionalen Benutzer-Anforderungen beschrieben wurden, einhält.

Der *Validierer* überprüft für jede Integrationsstufe des ablauffähigen Systems, ob die Anforderungen, welche in der entsprechenden Stufe der Software-System-Beschreibung aufgelistet wurden, mit dem Verhalten übereinstimmen.

Ausgewählte Rollen:

Es wird der Schwerpunkt mehr auf die managementorientierten und organisatorischen Rollen des Projektplaners, des Projektmanagers und des Qualitätssicherers gelegt und damit auf eine weitere Unterscheidung des Entwickler verzichtet. Für jede dieser betrachteten Rollen soll nur eine Auswahl der Aufgaben unterstützt werden, welche nun erläutert werden.

Der Projektplaner soll beim Erstellen und Anpassen eines Projektplans unterstützt werden, indem einerseits am Ende eines Projekts eine Zusammenfassung dargestellt wird, und andererseits die Möglichkeit besteht, den aktuellen Zustand eines Projekts zu erfragen.

Ein Schwerpunkt des Projektmanagers liegt darin, den Plan zu überwachen und die Erkenntnisse für die Zukunft zu nutzen. Außerdem muss er die Häufigkeit der Fehler überwachen und Verfahren zu deren Reduktion einführen.

Der Qualitätssicherer soll darin unterstützt werden, Aussagen über die Qualität zu machen, ebenso wie den Fortschritt der Dokumentation zu überwachen und geeignete Maßnahmen zur Erhöhung der Qualität einzuleiten.

Dem Entwickler soll durch den SPCC die Möglichkeit gegeben werden, sich selbst im Vergleich zum Gesamtbild zu beurteilen, darunter fällt auch die Überprüfung, ob seine verwendeten Techniken und Methoden angemessen sind.

4.2 Ausgewählte Metriken

In diesem Abschnitt werden die verschiedenen Metriken erläutert, die den Sichten des nächsten Kapitels zu Grunde liegen. Es wurden Metriken aus verschiedenen Kategorien verwendet. Diese Kategorien sind: Aufwand, Zeit, Fehler, Änderungen, Kosten, Priorität und Kode-Metriken. Hierbei handelt es sich um die gleichen Kategorien, in welche sich auch die Metriken, die den Sichten aus der Literatur zu Grunde liegen, einteilen lassen. Aus jeder Kategorie wird mindestens eine Metrik verwendet, wobei es jedoch möglich ist, dass die Metriken in mehr als eine Kategorie fallen, jedoch nur einer zugeordnet werden.

Jede Metrik wird nummeriert, um sie später referenzieren zu können, wenn die einzelnen Sichten klassifiziert werden.

4.2.1 Zeit

- Metrik 1: Dauer der einzelnen Phasen: Diese Metrik ordnet jeder Phase eine Dauer (in Wochen) zu. Mit dieser Metrik kann ein Zeitplan eines Projektes erstellt und überwacht werden, indem zuerst mit Hilfe dieser Metrik den Phasen die geplante Dauer zugeordnet wird und nach Abschluss der Phase die tatsächliche Dauer und diese miteinander verglichen werden.

4.2.2 Aufwand

- Metrik 2: Aufwand pro Phase: Es wird jeder Phase ein entsprechender Aufwand zugeordnet, wobei der Aufwand in Personenstunden gemessen wird. Es kann überprüft werden, ob ein Aufwand gewisse Schranken überschreitet. Dies kann dadurch geschehen, dass jeder Phase sowohl ein Wert für eine obere als auch für eine untere Schranke zugeordnet wird. Diese Werte werden mit dem

tatsächlichen Aufwand verglichen, welcher ebenfalls wieder der Phase zugeordnet wird.

- Metrik 3: Aufwand pro Aktivität: Jeder einzelnen Aktivität wird ein Aufwand zugeordnet, wobei es sich hierbei entweder um eine Schätzung des Aufwandes oder den tatsächlichen Aufwand handeln kann. Der Aufwand wird in Personenstunden gemessen. Mit Hilfe dieser Metrik kann überprüft werden, ob der Aufwand für die einzelnen Aktivitäten innerhalb gewisser Schranken liegt. Hierfür wird jeder Aktivität sowohl eine obere und untere Schranke für den Aufwand zugeordnet, als auch der tatsächliche Aufwand und anschließend miteinander verglichen.

- Metrik 4: Aufwand pro Testart und pro Klasse: Jeder Klasse und Testart wird ein Aufwandswert zugeordnet, welcher in Personenstunden gemessen wird. Es ist möglich, die Klassen zu bestimmen, welche noch getestet werden müssen, und wie aufwändig es ist, die jeweiligen Klassen zu testen.

- Metrik 5: Aufwand pro Testart und pro Package: Jedem Package und jeder Testart wird ein Aufwandswert zugeordnet. Hierfür wird der Aufwandswert der entsprechenden Testart von jeder Klasse und jedem Package innerhalb des Packages summiert. Dies wird rekursiv durchgeführt, solange bis ein Package nur noch Klassen enthält. Der Aufwand wird in Personenstunden gemessen. Analog zur vorangegangenen Metrik können die Packages bestimmt werden, welche intensiver getestet worden sind, und welche kaum bzw. noch nicht getestet wurden.

4.2.3 Fehler

- Metrik 6: Fehler pro Phase und Affiliation: Jeder Kombination von Phase und Affiliation wird die Summe aller Fehler zugeordnet, welche die entsprechende Phase und Affiliation besitzen. Hiermit ist es möglich, einen Überblick über alle Phasen zu gewinnen, und zu beurteilen, wie effektiv der Fehlerfindungsprozess war.

- Metrik 7: Aufwand für die Nacharbeit (*Rework*), verursacht durch einen Fehler: Jedem Fehler wird ein Aufwand zugeordnet, der notwendig ist, um diesen zu korrigieren. Mit Hilfe dieser Metrik ist es möglich, die Fehler zu identifizieren, deren Behebung mit einem großen Aufwand verbunden sind.

- Metrik 8: Aufwand pro Phase und Affiliation (Ursprung des Fehlers; siehe Metrik 11): Für jede Kombination von Phase und Affiliation wird der Aufwand der zugehörigen Fehler aufaddiert. Hiermit ist es möglich, die Phasen zu identifizieren, in denen Fehler einen großen Aufwand zu Folge haben.

- Metrik 9: Fehler pro Fehlerklasse: Jeder Fehler wird einer bestimmten Fehlerklasse zugeordnet und für jede Klasse die Fehler aufaddiert. Die Metrik kann dazu benutzt werden, einen Überblick über die relative Häufigkeit der einzelnen Fehlerklassen zu erhalten.

- Metrik 10: Fehlerstatus: Für jeden Status werden diejenigen Fehler aufaddiert, welche diesen zu diesem Zeitpunkt haben. Mit Hilfe dieser Sicht kann der

Stand der Fehlerbehebung überblickt werden, da jeder Fehler einen gewissen Status hat. Damit kann erkannt werden, welche Fehler bereits behoben sind, oder welche noch lokalisiert werden müssen.

- Metrik 11: Ursprung des Fehlers (*Source/Affiliation*): Jedem Fehler wird eine Phase zugeordnet, in welcher das Dokument, das den Fehler beinhaltet entstanden ist. Anhand dieser Metrik können die Phasen identifiziert werden, in welchen viele Fehler ihren Ursprung haben. In diesen kann mehr Zeit für das Review eingeplant werden.

- Metrik 12: Zeitpunkt der Entdeckung eines Fehlers: Jedem Fehler wird die Phase zugeordnet, in der Fehler gefunden wurde. Zusammen mit der vorangegangenen Metrik ist es möglich, Aussagen über die Qualität des Fehlerfindungsprozesses zu machen. Wenn viele Fehler erst in einer späteren Phase gefunden werden, ist es notwendig, dass für das Review mehr Zeit reserviert wird und auch, dass andere Techniken und Methoden verwendet werden.

4.2.4 Änderungen

- Metrik 13: Aufwand für Änderungen pro Phase: Es wird der Aufwand, welcher für die Änderungen, die in der gleichen Phase durchgeführt wurden, aufaddiert und dieser Phase zugeordnet. Es kann der Aufwand pro Phase, der durch Änderungen verursacht wird, dargestellt werden und damit überprüft werden, ob der Prozess geeignet ist.

- Metrik 14: Änderungen pro Phase: Es wird für jede Phase die Anzahl der Änderungen, welche in dieser durchgeführt wurden, addiert. Es ist möglich für alle Phasen die Anzahl der Änderungen darzustellen. Zusammen mit Metrik 13 ist es außerdem möglich, abzuschätzen welcher Anteil des Aufwands der Phasen auf die Änderungen entfällt und ob der Prozess geeignet ist.

- Metrik 15: Anzahl der Änderungen in einer Kategorie: Die durchgeführten Änderungen werden eine der drei Kategorien *perfective*, *adaptive* und *corrective* zugeordnet - abhängig von der Ursache der Änderung - und die Anzahl jeweils addiert. Anhand dieser Metrik ist es möglich zu erkennen, wie sich die Änderungen auf die einzelnen Kategorien verteilen.

- Metrik 16: Anzahl der Änderung pro Klasse: Für jede Klasse werden die Änderungen, welche durchgeführt wurden, aufaddiert. Mit Hilfe dieser Metrik ist es möglich, die Klassen zu identifizieren, die von vielen Änderungen betroffen sind.

- Metrik 17: Anzahl der Änderungen pro Package: In jedem Package wird die Anzahl der Änderungen jeder Klasse und jedes Packages von diesem Package summiert und diesem Package zugeordnet. Dies wird rekursiv angewendet, bis nur noch Klassen in einem Package enthalten sind. Analog zu der vorangegangenen Metrik ist es möglich, diejenigen Packages zu identifizieren, welche von vielen Änderungen betroffen sind.

4.2.5 Priorität

- Metrik 18: Priorität der Anforderungen: Für jede einzelne Anforderung an ein System, wird eine Priorität (+, -, 0) bestimmt. Werden diejenigen Anforderungen markiert, welche bereits erfüllt sind, ist es möglich, den ungefähren Stand eines Projekts zu erhalten.

4.2.6 Kode-Metriken

- Metrik 19: Verhältnis zwischen Dokumentations- und Kodezeilen pro Klasse: Es wird für jede Klasse die Anzahl aller Zeilen mit Dokumentation durch die Gesamtzahl aller Kodezeilen dividiert und dieser Klasse zugeordnet. Es können diejenigen Klassen identifiziert werden, welche wenig oder sehr viel dokumentiert sind.

- Metrik 20: Verhältnis zwischen Dokumentations- und Kodezeilen pro Package: Für jedes Package wird ein Mittelwert aus allen Verhältnissen zwischen Kode- und Dokumentationszeilen der Klassen und Packages, die zu diesem Package gehören, gebildet und diesem Package zugeordnet. Dies wird rekursiv angewendet, bis ein Package nur noch Klassen enthält. Analog zur vorangegangenen Sicht können die Packages identifiziert werden, die wenig oder überdurchschnittlich viel dokumentiert sind.

4.2.7 Kosten

- Metrik 21: geschätzte Kosten pro Phase: Es wird jeder Phase eines Prozesses ein Wert für die geschätzten Kosten (z.B. in Mio. Euro), welche mit dieser Phase verknüpft werden, zugeordnet. Mit Hilfe der Metrik kann die Verteilung der Kosten auf die einzelnen Phasen dargestellt werden und zusammen mit Metrik 23 es ist auch möglich, die realen Kosten mit den geplanten zu vergleichen, indem jeweils die Kosten den Phasen zugeordnet werden.

- Metrik 22: geschätzte Kosten pro Aktivität: Es wird jeder Aktivität einer Phase ein Wert für die geschätzten Kosten, welche mit dieser Aktivität verknüpft werden, zugeordnet. Es ist mit Hilfe dieser Metrik möglich, die Kostenverteilung auf die Aktivitäten darzustellen und zusammen mit Metrik 24 diese auch mit geplanten Werten, oder einer oberen und unteren Schranke, zu vergleichen.

- Metrik 23: reale Kosten pro Phase: Es wird jeder Phase eines Prozesses ein Wert für die realen Kosten (z.B. in Mio. Euro), welche mit dieser Phase verknüpft werden, zugeordnet. Mit Hilfe der Metrik kann die Verteilung der Kosten auf die einzelnen Phasen dargestellt werden und zusammen mit Metrik 21 es ist auch möglich, die realen Kosten mit den geplanten zu vergleichen, indem jeweils die Kosten den Phasen zugeordnet werden.

- Metrik 22: reale Kosten pro Aktivität: Es wird jeder Aktivität einer Phase ein Wert für die realen Kosten, welche mit dieser Aktivität verknüpft werden, zugeordnet. Es ist mit Hilfe dieser Metrik möglich, die Kostenverteilung auf die Aktivitäten darzustellen und zusammen mit Metrik 22 diese auch mit geplanten Werten, oder einer oberen und unteren Schranke, zu vergleichen.

4.3 Resultierende Sichten

In Kapitel 4.1 wurde ein Menge von Rollen bestimmt, die mit Hilfe des SPCC unterstützt werden sollen, außerdem wurde für jede Rolle eine Menge von Aufgaben ausgewählt, da nicht alle Aufgaben geeignet sind, um unterstützt zu werden. Diese wurden genutzt, um eine Menge von Meta-Sichten abzuleiten, die in diesem Kapitel erläutert werden. Aus diesen Meta-Sichten können Sichten hergestellt und implementiert werden. Jede dieser Meta-Sichten wird sowohl allgemein beschrieben als auch anhand der Klassifikation von Kapitel 3.2. Die Metriken, auf welche hier verwiesen wird, wurden im vorangegangenen Kapitel erläutert und nummeriert.

4.3.1 Aufwandsverteilung auf Phasen bzw. Aktivitäten

Es wird die Verteilung des Aufwands auf die verschiedenen Phasen eines Projekts dargestellt. Eine sinnvolle Verfeinerung hiervon besteht darin, den Aufwand verteilt auf die Aktivitäten einer Phase darzustellen. Hierbei sind zwei Möglichkeiten denkbar, wie eine Vergleichsmöglichkeit aussehen könnte. Die erste besteht darin, zusätzlich zum tatsächlichen Verlauf auch eine Referenzkurve, die entweder den geplanten Verlauf oder den Mittelwert aus vergangenen Projekten repräsentiert, einzutragen. Die Alternative ist eng damit verwandt und besteht darin, dass nicht nur eine, sondern zwei zusätzliche Kurven eingetragen werden. Diese Kurven repräsentieren Schranken für den Verlauf des Aufwands. Im letzteren Fall ist es einfach, abweichende Werte zu bestimmen, nämlich die Werte, die außerhalb des durch die Schranken eingeschlossenen Bereichs liegen. Im anderen Fall ist die Angabe eines zusätzlichen Parameters notwendig, der die maximale prozentuale Abweichung angibt.

Diese Sicht ist ein Hilfsmittel, um den Projektplan gegebenenfalls anzupassen. Und es ist auch möglich, über die Notwendigkeit der Umverteilung des Personals zu entscheiden.

Rollen	Projektplaner, Projektmanager
Metriken	Metrik 2, Metrik 3
Vergleichsdaten	Referenzkurve; obere und untere Schranke
Darstellungsarten	Liniendiagramm
Anpassung	maximale prozentuale Abweichung
ref. Sichten	-

Tabelle 7: Klassifikation der Sicht: Aufwandsverteilung auf Aktivitäten.

4.3.2 Zeitlicher Projektverlauf

Es wird die Dauer der verschiedenen Phasen aufgetragen, wobei die Dauer erst ab Beginn der Phase aufgetragen wird. Dabei wird nicht nur der tatsächliche Verlauf eingetragen, sondern auch der geplante zeitliche Verlauf (Projektwochen). Diese Sicht kann zur Anpassung des Projektplans benutzt werden, sowie zur Überwachung des Plans.

Rollen	Projektplaner, Projektmanager
Metriken	Metrik 1
Vergleichsdaten	geschätzte Daten (können aus vergangenen Projekten abgeleitet werden)
Darstellungsarten	Zeitplan
Anpassung	-
ref. Sichten	-

Tabelle 8: Klassifikation der Sicht: zeitlicher Projektverlauf.

4.3.3 Fehlerverteilung auf Phasen und Affiliation und zugehörigem Aufwand

Für jede Kombination von Phasen und Affiliations wird die Anzahl der Fehler zusammen mit dem summierten Aufwand dargestellt. Hierbei entspricht die Phase derjenigen Phase, in welcher der Fehler gefunden wurde und die Affiliation der Phase, in der das Dokument erstellt wurde, das den Fehler enthält. Dies hat zur Folge, dass es Kombinationen gibt, für welche keine Fehler existieren können, da ein Fehler frühestens in der Phase gefunden werden kann, in der er verursacht wurde. Der Aufwand wird entlang der y-Achse aufgetragen und die Anzahl der Fehler oberhalb des entsprechenden Balkens angegeben. Mit Hilfe dieser Sicht ist es möglich Aussagen über die Qualität des Reviews verschiedener Phasen zu machen, da erkennbar ist, in welchen Phasen viele Fehler verursacht wurden, die erst später gefunden wurden, bzw. einen hohen Aufwand zur Folge haben. Außerdem kann entschieden werden, für welche Phasen mehr Zeit für das Review eingeplant werden muss, oder ob andere Methoden und Werkzeuge eingesetzt werden müssen.

Rollen	Projektmanager, Qualitätssicherer
Metriken	Metrik 6, Metrik 8, Metrik 11, Metrik 12
Vergleichsdaten	-
Darstellungsarten	Balkendiagramm
Anpassung	-
ref. Sichten	-

Tabelle 9: Klassifikation der Sicht: Fehlerverteilung auf Phasen und Affiliation und zugehörigem Aufwand.

4.3.4 Fehlerverteilung auf Fehlerklassen

Es werden alle Fehler verteilt auf die verschiedenen Fehlerklassen dargestellt, wobei es sich im Normalfall um eine totale Verteilung handelt und es gegebenenfalls sinnvoll ist, die absolute Anzahl anzugeben. Die Sicht kann dazu herangezogen werden, zu entscheiden, ob mehr Zeit für Fehlerfindung und Fehlerkorrektur eingeplant werden muss, und auch, ob weitere Maßnahmen zur Reduzierung von Fehlern notwendig sind.

Rollen	Projektmanager, Qualitätssicherer
Metriken	Metrik 9
Vergleichsdaten	-
Darstellungsarten	Kreisdiagramm, Balkendiagramm
Anpassung	Diagrammtyp
ref. Sichten	Fehlerverteilung nach Status Fehlerverteilung nach Status und Fehlerklasse

Tabelle 10: Klassifikation der Sicht: Fehlerverteilung auf Fehlerklassen.

4.3.5 Fehlerverteilung nach Status

Für jeden Status wird die Anzahl der Fehler aufgetragen. Diese Sicht wird durch die Sicht 4.3.6 verfeinert und dient dazu, einen Überblick über die Anzahl der noch zu behebenden Fehler zu gewinnen, und über die Qualität der bereits behobenen Fehler.

Rollen	Qualitätssicherer, Entwickler
Metriken	Metrik 10
Vergleichsdaten	-
Darstellungsarten	Kreisdiagramm, Balkendiagramm
Anpassung	Diagrammtyp
ref. Sichten	Fehlerverteilung auf Fehlerklassen Fehlerverteilung nach Status und Fehlerklasse

Tabelle 11: Klassifikation der Sicht: Fehlerverteilung auf Status.

4.3.6 Fehlerverteilung nach Status und Fehlerklasse

Es wird die Anzahl der Fehler verteilt auf Status und Fehlerklasse dargestellt. Die Gruppierung (Einteilung der x-Achse) ist variabel entweder nach Status oder Fehlerklasse und kann anhand eines Parameters eingestellt werden. Diese Sicht fasst die beiden vorangegangenen Sichten zusammen und dient wie jene auch dazu, die Lösungen der Fehler zu beurteilen, und einen Überblick über die noch zu behebenden Fehler zu erlangen. Außerdem kann sie benutzt werden, um über die Zuteilung von mehr Zeit für das Finden und Beheben von Fehlern zu entscheiden.

Rollen	Projektmanager, Qualitätssicherer, Entwickler
Metriken	Metrik 10, Metrik 11
Vergleichsdaten	-
Darstellungsarten	Balkendiagramm
Anpassung	Einteilung der x-Achse (Status oder Klasse)
ref. Sichten	Fehlerverteilung nach Status Fehlerverteilung auf Fehlerklassen

Tabelle 12: Klassifikation der Sicht: Fehlerverteilung nach Status und Fehlerklasse.

4.3.7 Auflistung der Anforderungen*:*

Es werden alle verschiedenen Anforderungen (z.B. Entwickleranforderungen, Benutzeranforderungen) zusammen mit der jeweiligen Priorität aufgelistet und diejenigen hervorgehoben, welche bereits implementiert wurden, um damit den Fortschritt des Projekts zu beurteilen.

Rollen	Projektmanager, Entwickler
Metriken	Metrik 18
Vergleichsdaten	-
Darstellungsarten	Tabelle
Anpassung	-
ref. Sichten	-

Tabelle 13: Klassifikation der Sicht: Auflistung der Anforderungen.

4.3.8 Ort der Änderungen

Diese Sicht stellt den Ort einer Änderungen dar, wobei es sowohl denkbar ist, dass hierfür verschiedene Klassen (oder Packages) des Systems als Grundlage dienen oder auch die unterschiedlichen Dokumente, welche während des Entwicklungsprozesses erstellt werden. Werden die Klassen und Packages als mögliche Orte gewählt, so ist es je nach Größe des Projektes sinnvoll, verschiedene Abstraktionsgrade zu wählen. Die Sicht kann durch sich selbst verfeinert werden, indem durch Auswählen eines Packages nur die Klassen dieses Packages angezeigt werden, bzw. immer nur die Packages und Klassen einer „Ebene" angezeigt werden. Dies kann mittels eines Parameters gesteuert werden. Durch Einbeziehen der Sicht, kann entschieden werden, ob eine Umgruppierung des Personals notwendig ist, wenn z.B. in einer Gruppe zu erkennen ist, dass alle Entwickler zu unerfahren sind, um die Notwendigkeit von Änderungen zu vermeiden. Außerdem können Aussagen über die Qualität getroffen werden.

Rollen	Projektmanager, Qualitätssicherer, Entwickler
Metriken	Metrik 16, Metrik 17
Vergleichsdaten	-
Darstellungsarten	Balkendiagramm
Anpassung	Package-Auswahl
ref. Sichten	Sicht für übergeordnetes Package (sofern vorhanden) Sicht für Sub-Packages (sofern vorhanden)

Tabelle 14: Klassifikation der Sicht: Ort der Änderungen.

4.3.9 Kategorie der Änderungen

In dieser Sicht werden die Änderungen anhand ihrer Ursache eingeteilt, wobei zuerst einmal eine Einteilung in drei Hauptgruppen stattfindet: Anpassen (*adaptive*),

Fehlerbehebung (*corrective*) und Wartungspflege (*perfective*). Es ist jedoch denkbar, diese Einteilung zu verfeinern, vor allem unter dem Gesichtspunkt, dass während der Entwicklung hauptsächlich Änderungen zur Fehlerbehebung durchgeführt werden. So wären weitere mögliche Einteilungen: Änderungen in den Anforderungen (noch während der Entwicklung) oder Erhöhung der Robustheit. Auch hierbei handelt es sich um eine Sicht, die zur Prozessverbesserung und weiteren Verbesserungsmaßnahmen benutzt werden kann. Zusätzlich kann daraus abgeleitet werden, ob die Entwicklung so durchgeführt werden muss, dass Änderungen einfacher durchzuführen sind.

Rollen	Projektmanager, Qualitätssicherer
Metriken	Metrik 15
Vergleichsdaten	-
Darstellungsarten	Balkendiagramm, Kreisdiagramm
Anpassung	Diagrammtyp
ref. Sichten	-

Tabelle 15: Klassifikation der Sicht: Kategorie der Änderungen.

4.3.10 Anzahl der Änderungen pro Phase

Die Anzahl der Änderungen wird, unterteilt auf die verschiedenen Phasen, dargestellt, um daraus für den Prozess zukünftiger Projekte mögliche Verbesserungen abzuleiten, indem z.B. ein Prototyp entwickelt wird.

Rollen	Projektmanager, Qualitätssicherer
Metriken	Metrik 14
Vergleichsdaten	möglicher Vergleich mit alten Daten
Darstellungsarten	Liniendiagramm, Balkendiagramm
Anpassung	Diagrammtyp
ref. Sichten	- Aufwand für die Änderungen pro Phase - Änderungen und damit verbundener Aufwand pro Phase

Tabelle 16: Klassifikation der Sicht: Anzahl der Änderungen pro Phase.

4.3.11 Aufwand für die Änderungen pro Phase

Für jede Phase wird der Aufwand, welcher durch die Änderungen in dieser Phase verursacht wurde, eingetragen. Dies kann zusammen mit anderen Sichten dazu benutzt werden, einen Projektplan zu erstellen oder anzupassen, und zusätzlich, um Verbesserungen des Prozesses ableiten zu können.

4.3.12 Änderungen und damit verbundener Aufwand pro Phase

Die Sicht fasst die beiden vorangegangenen Sichten zusammen und stellt sowohl die Anzahl der Änderungen als auch den Aufwand für jede Phase dar und kann dazu

Rollen	Projektplaner, Projektmanager, Qualitätssicherer
Metriken	Metrik 13
Vergleichsdaten	möglicher Vergleich mit alten Daten
Darstellungsarten	Liniendiagramm, Balkendiagramm
Anpassung	Diagrammtyp
ref. Sichten	- Anzahl der Änderungen pro Phase - Änderungen und damit verbundener Aufwand pro Phase

Tabelle 17: Klassifikation der Sicht: Aufwand für die Änderungen pro Phase.

benutzt werden, den Prozess zu verbessern, und weitere Verbesserungsmaßnahmen einzuleiten. Dabei ist zu beachten, dass aufgrund der unterschiedlichen Einheiten für Aufwand und Anzahl der Änderungen, zwei Achsen zum Auftragen notwendig sind und daher eine Mischung aus verschiedenen Diagrammtypen sich am besten eignet, um die Daten der richtigen Achse zuzuordnen.

Rollen	Projektmanager, Qualitätssicherer
Metriken	Metrik 13, Metrik 14
Vergleichsdaten	-
Darstellungsarten	Mischung aus Linien- und Balkendiagramm
Anpassung	-
ref. Sichten	- Anzahl der Änderungen pro Phase - Aufwand für die Änderungen pro Phase

Tabelle 18: Klassifikation der Sicht: Änderungen und damit verbundener Aufwand pro Phase.

4.3.13 Prozentuales Verhältnis zwischen Kodezeilen und Dokumentationszeilen

Diese Sicht stellt das relative Verhältnis zwischen den Zeilen zur Dokumentation und Kodezeilen, zusammen mit einem Mittelwert aus vergangenen Projekten, dar. Abhängig von der Anzahl der Klassen und Packages kann es sinnvoll sein, eine Hierarchie aufzubauen, indem zuerst das Verhältnis auf oberster Package-Ebene dargestellt wird (z.B. gemittelt über alle Klassen) und dann für jedes Package die Möglichkeit besteht, sich das Verhältnis innerhalb von diesen anzuschauen. Dies kann so lange fortgesetzt werden, bis nur noch Klassen vorhanden sind. Der Nutzen dieser Sicht besteht darin, den Stand der Dokumentation zu messen und daraus resultierend die Qualität des Produktes.

4.3.14 Aufwand für Tests pro Klasse

Es wird für jede Klasse dargestellt, wie groß der Aufwand ist, der für das Testen aufgewendet wurde, wobei eine Unterscheidung nach Art des Test (z.B. white box

Rollen	Qualitätssicherer, Entwickler
Metriken	Metrik 19, Metrik 20
Vergleichsdaten	Mittelwert aus vergangenen Projekten
Darstellungsarten	Balkendiagramm mit Mittelwertskurve
Anpassung	Package-Auswahl
ref. Sichten	-

Tabelle 19: Klassifikation der Sicht: Prozentuales Verhältnis zwischen Kodezeilen und Dokumentationszeilen.

und black box) stattfindet. Weiterhin ist es von Vorteil, bei größeren Projekten eine Hierarchie einzuführen und nicht alle Klassen im gleichen Diagramm darzustellen. Mit Hilfe dieser Sicht kann ein Überblick über den Fortschritt der Tests gewonnen werden und damit verbunden können Aussagen über die Qualität des Produkts gemacht werden.

Rollen	Qualitätssicherer, Entwickler
Metriken	Metrik 4, Metrik 5
Vergleichsdaten	direkter Vergleich durch Gegenüberstellung; möglich: durchschnittlicher Aufwand aus vergangenen Projekten
Darstellungsarten	Balkendiagramm mit Mittelwertskurve
Anpassung	Package-Auswahl
ref. Sichten	Sicht für übergeordnetes Package (soweit vorhanden) Sicht für Sub-Packages (soweit vorhanden)

Tabelle 20: Klassifikation der Sicht: Aufwand für Tests pro Klasse.

4.3.15 Kostenvergleich pro Phase

Neben der Darstellung des Aufwands für jede Phase, ist es außerdem sinnvoll, die Kosten für jede Phase und als Verfeinerung die Kosten für jede Aktivität, darzustellen. Hierbei werden jeweils die geplanten und tatsächlichen Kosten einander gegenüber gestellt, um daraus ableiten zu können, ob es zu Budgetüberschreitungen kommen wird und Gegenmaßnahmen notwendig sind, oder ob Anpassungen des Projektplans notwendig sind. Weiterhin kann nicht nur ein Überblick über das gegenwärtige Projekt gewonnen werden, sondern es können auch gesammelte Erfahrungen für zukünftige Projekte benutzt werden.

4.3.16 Verteilung der Gesamtkosten

Diese Sicht hängt eng mit der vorherigen zusammen, wobei diese erst am Ende eines Projekts sinnvoll ist, da die Verteilung der Kosten auf die einzelnen Phasen erst am Ende des Projekts bekannt ist. Auch hier besteht die Möglichkeit, diese Sicht zu verfeinern, indem für eine Phase die Verteilung der Kosten auf die Aktivitäten dargestellt wird. Eine sinnvolle Ergänzung der Phasen besteht darin, noch die Wartung zu berücksichtigen, vor allem da Wartung einen Großteil der Kosten

Rollen	Projektmanager, Projektplaner
Metriken	Metrik 21, Metrik 22
Vergleichsdaten	geschätzte Kosten (können aus vergangenen Projekten abgeleitet werden)
Darstellungsarten	Balkendiagramm, Liniendiagramm
Anpassung	Diagrammtyp
ref. Sichten	Kostenvergleich pro Aktivität

Tabelle 21: Klassifikation der Sicht: Kostenvergleich pro Phase.

ausmachen kann. Da diese Sicht erst am Ende des Projektes sinnvoll ist, kann sie als Unterstützung bei der Planerstellung von späteren Projekten benutzt werden. Bei einem zu großen Anteil der Wartungskosten ist es sinnvoll, die Ursachen hierfür zu analysieren und daraus resultierende Erkenntnisse für spätere Projekte zu nutzen, damit dann weniger Wartung notwendig wird, bzw. diese kostengünstiger durchzuführen ist.

Rollen	Projektmanager, Projektplaner
Metriken	Metrik 21
Vergleichsdaten	-
Darstellungsarten	Kreisdiagramm, Balkendiagramm
Anpassung	Diagrammtyp
ref. Sichten	Verteilung der Kosten auf Aktivitäten

Tabelle 22: Klassifikation der Sicht: Verteilung der Gesamtkosten.

5 Implementierung und Evaluation

Nachdem im vorangegangenen Kapitel Meta-Sichten abgeleitet wurden, soll nun ein Teil als Sichten implementiert werden. Diese werden in diesem Kapitel noch einmal genauer mit allen Eingaben und Parametern beschrieben. Außerdem werden die einzelnen Sicht anhand den Anforderungen von 3.1 bewertet und dadurch hervorgehoben, welche Unterschiede zwischen den Sichten aus der Literatur und den implementierten bestehen. Schließlich werden am Ende die Erfahrungen, die sich aus dem Einsatz der Sichten ergeben zusammengefasst.

5.1 Einleitung

An dieser Stelle wird zuerst der Transformationsprozess, d.h. die Verarbeitung der Daten und anschließende Generierung der SVG-Grafiken, beschrieben, bevor die Implementierung der Sichten genauer erläutert wird.

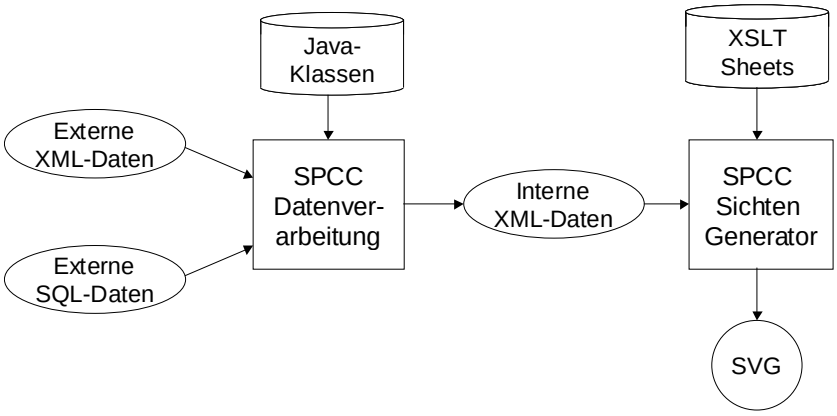


Abbildung 24: Darstellung des Transformationsprozesses; ausgehend von externen Daten über deren Verarbeitung zu internen Daten, die vom Generator mit Hilfe eines XSLT-Stylesheets in SVG-Grafiken umgewandelt werden.

In Abbildung 24 ist der Ablauf des Transformationsprozesses zu sehen. Ausgehend von externen Daten, die entweder in XML oder SQL vorliegen, werden interne XML-Daten erzeugt. Diese Erzeugung geschieht durch die Verarbeitung innerhalb des SPCCs, in welcher neben den externen Daten auch Java-Klassen mit eingehen. Letztere repräsentieren die verschiedenen Funktionen. Die erzeugten internen XML-Daten gehen in den SPCC-Sichten-Generator ein, welcher XSLT-Stylesheets benutzt, um die Daten in SVG-Grafiken zu transformieren.

In Abbildung 25 ist ein Screenshot des SPCC-Clients zu sehen, der alle zur Verfügung stehenden Funktionen zeigt, welche Daten verarbeiten können. Beim Bearbeiten einer solchen Funktion kann jeweils die entsprechende Java-Klasse, die die gewünschte Funktionalität bereitstellt, ausgewählt werden. Hier kann auch die Konsistenz der Funktionen überprüft werden.

Weiterhin ist in Abbildung 26 ein Screenshot aller zur Verfügung stehenden Sichten und in Abbildung 27 einer aller erzeugten SVG-Grafiken der jeweiligen Visua-

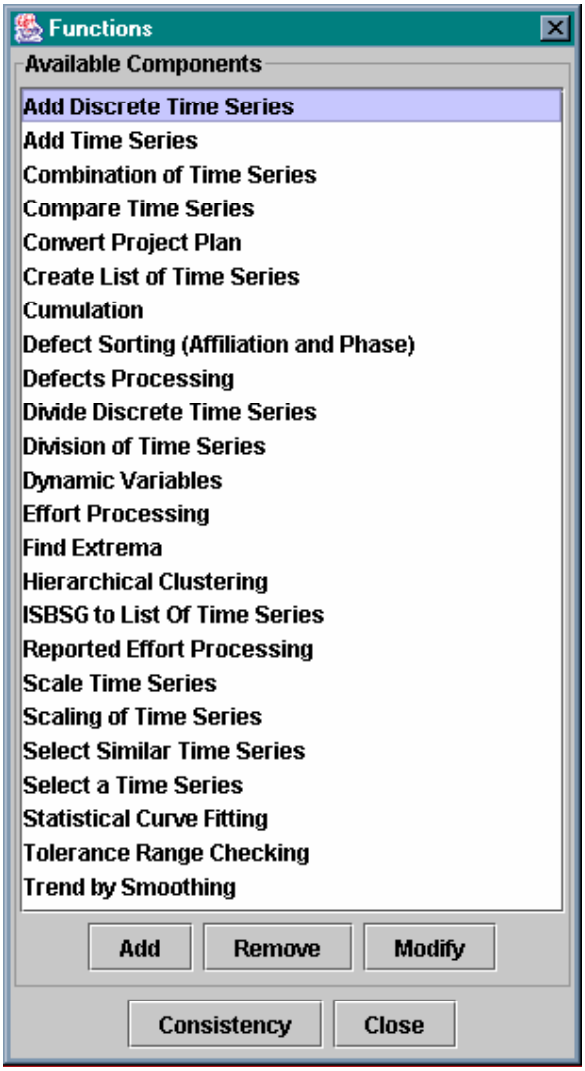


Abbildung 25: Fenster, welches die zur Verfügung stehenden Funktionen zeigt.

lisierungskette zu sehen. In letzterem ist auch zu erkennen, dass der Client die Möglichkeit bietet, die erzeugten SVG-Grafiken zu betrachten.

Die Implementierung der Sichten geschieht in XSL (*eXtensible Stylesheet Language*), wobei es sich hierbei um eine Programmiersprache handelt, die in der Lage ist, XML-Dateien in ein anderes Format, in diesem Fall SVG (*Scalable Vector Graphics*), umzuwandeln. Ein wichtiges Konzept bei der Programmierung in XSL ist die Rekursion.

Bei SVG handelt es sich um ein Beschreibungsformat für Vektorgrafiken, welche mit einem speziellen SVG-Viewer oder Plugin betrachtet werden können. SVG basiert wiederum auf XML. Vorteile von SVG liegen unter anderem darin, dass es zunächst ein offener Standard zur Darstellung von skalierbaren Graphiken ist und das

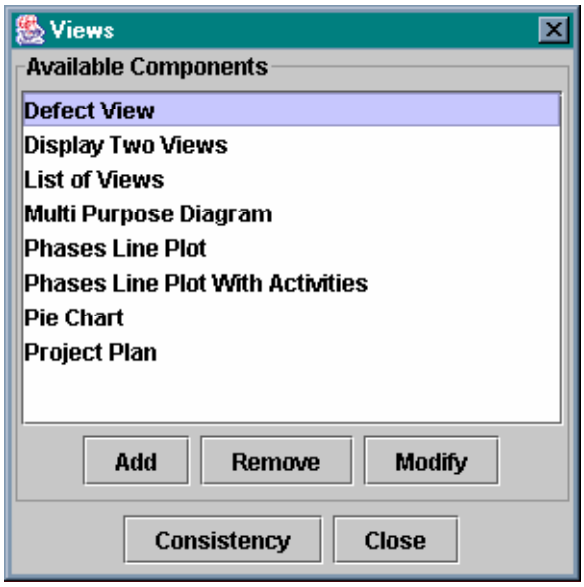


Abbildung 26: Fenster, welches die zur Verfügung stehenden Sichten zeigt.

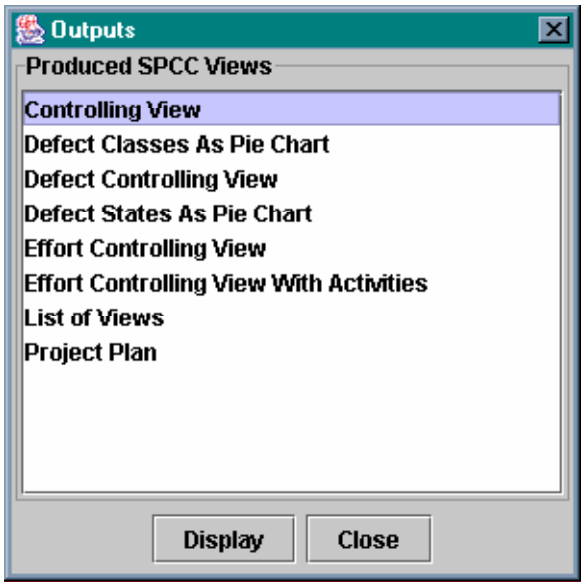


Abbildung 27: Fenster, welches die zur Verfügung stehenden SVG-Grafiken zeigt.

Dokument selbst wieder XML ist, so dass es sich problemlos in andere XML-basierte Dokumente integrieren lässt. Außerdem bietet SVG verschiedene Möglichkeiten zur Animation, Interaktivität und unterstützt auch die Verwendung von Java-Skripten (ECMA-Script).

Momentan gibt es unter anderem ein Plugin von Adobe für verschiedene Webbrowser und ein SVG-Viewer von Apache auf Java-Basis (*Batik SVG Toolkit*). Beide können die hier realisierten Sichten darstellen, wobei die Darstellung mit dem Adobe-Plugin Version 3.0 und mit dem Batik SVG-Viewer für die Versionen 1.1.1 und 1.5beta4b getestet wurde.

5.2 Implementierung der Sichten

Es wurden fünf Sichten implementiert, hierzu zählen: Aufwandsverteilung auf die Phasen (siehe 4.3.1), Aufwandsverteilung auf die Phasen und Aktivitäten (siehe 4.3.1), Fehlerverteilung auf Phasen und Affiliation mit zugehörigem Aufwand (siehe 4.3.3), sowie der zeitliche Projektverlauf (siehe 4.3.2).

Die fünfte Sicht wurde oben noch nicht aufgelistet, da diese Sicht allgemeiner gefasst ist und bereits zwei konkrete Instanzen besitzt. Es handelt sich hierbei um ein Kreisdiagramm, welches - wie bereits in Abschnitt 1.8.3 erläutert wurde - geeignet ist für die Darstellung einer Verteilung. Damit eignet sich diese Sicht sowohl für die Fehlerverteilung sortiert nach Status (siehe 4.3.5) als auch für die Fehlerverteilung auf die einzelnen Fehlerklassen (siehe 4.3.3).

Wie bereits angedeutet wurde, gibt es sehr spezialisierte Sichten, welche nur eine Art von Instanz haben; hierzu zählen die ersten vier Sichten. Es ist zwar möglich mehrere Instanzen dieser Sichten mit unterschiedlichen Daten zu haben, aber der Unterschied liegt einzig in den realen Daten, die den Sichten zugeordnet werden. Am Beispiel heißt dies, dass zwar unterschiedliche Aufwandsverläufe verwendet werden, es sich jedoch immer um den Typ *Aufwandsverlauf* handelt. Außerdem gibt es allgemeinere Sichten, in der verschiedene Daten dargestellt werden können, wobei nur eine implementiert ist (Kreisdiagramm). Am Beispiel der implementierten Sicht bedeutet dies, dass verschiedene Verteilungen dargestellt werden können, unabhängig davon, um welche Art von Daten es sich handelt.

Die beiden ersten Sichten wurden getrennt realisiert, da sie zwar viele Gemeinsamkeiten aufweisen, jedoch auch einige entscheidende Unterschiede. Da es einen erheblichen Mehraufwand bedeutet, diese Unterschiede auch in der Implementierung zu berücksichtigen, wurden diese Sichten getrennt realisiert und referenzieren sich nur noch gegenseitig. Auf die Unterschiede wird später noch genauer eingegangen, wenn die Sichten mit all ihren Parametern beschrieben werden.

Diese fünf Sichten werden jetzt zusammen mit den notwendigen Eingaben und Parametern genauer beschrieben. Außerdem werden die Funktionen kurz erläutert, die benutzt werden, um die Eingaben zu berechnen. Die genaue Implementierung der Sichten in XSL kann dem Anhang entnommen werden, an dieser Stelle werden nur die Grundlagen und Besonderheiten beschrieben.

5.2.1 Aufwandsverteilung auf Phasen (*Phases Line Plot*)

Diese Sicht ist als Liniendiagramm realisiert worden, da diese Darstellung die geeignetste ist. Neben dem tatsächlichen Verlauf sollen noch Referenzkurven eingetragen werden und auf diese Art sind die Kurven am besten miteinander zu vergleichen.

In die Sicht gehen eine Menge von optionalen und nicht-optionalen Daten ein. Die erste nicht-optionale Eingabe besteht natürlich aus den aktuellen Aufwandsdaten, bereits sortiert nach Phasen und addiert. Um diese Sicht später wiederverwen-

den zu können, wird eine Zeitreihe (*Time Series*) für die Eingabe gewählt. Dies bedeutet auch, dass eine Funktion (*Reported Effort Processing*) notwendig ist, die die Aufwandsdaten entsprechend konvertiert. Der Zeitstempel entspricht dabei der Phase, zu der der Eintrag gehört.

Eine weitere Eingabe besteht aus den Phasen, mit denen die x-Achse beschriftet werden soll. Dies ist zugleich auch die letzte Eingabe, die nicht optional ist.

Optionale Eingaben sind die beiden Referenzkurven. Dadurch dass die beiden Kurven optional sind, ist es auch möglich nur eine Referenzkurve als Eingabe eingehen zu lassen, wenn der Verlauf beispielsweise mit dem geplanten Verlauf verglichen werden soll. Auch diese Daten gehen als eine Zeitreihe (*Time Series*) ein und müssen daher zuerst durch eine Funktion konvertiert werden (*Effort Processing*), wobei auch hier der Zeitstempel der Nummer der Phase entspricht. Für die Konvertierung werden zwei Funktionen verwendet, da der erfasste Aufwand von einem anderen Datentyp ist, als die Referenzdaten. Es wurden verschiedene Datentypen benutzt, da für den erfassten Aufwand noch mehr Daten mitverwaltet werden, wie z.B. der Entwickler, der den Eintrag vorgenommen hat.

Da die Punkte, welche außerhalb der Schranken liegen, hervorgehoben werden sollen, müssen diese als Eingabe in die Sicht mit eingehen. Hierbei wird unterschieden zwischen den Werten, die zu groß sind, und denjenigen, die zu klein sind. Da es möglich ist, keine Referenzkurve anzugeben, existieren in diesem Fall auch keine abweichenden Punkte und somit muss diese Eingabe optional sein. Um diese Punkte zu berechnen, wird eine Funktion benötigt (*Tolerance Range Checking*), die sowohl den aktuellen Verlauf, als auch eine obere und untere Schranke als Eingabe hat, alle als Zeitreihe. Diese Funktion gibt jeweils eine Liste von Zeitreihen (*List of Time Series*) für die Punkte oberhalb und unterhalb aus, wobei eine Zeitreihe einem Wert, der außerhalb der Schranken liegt, entspricht.

Wenn nur eine Referenzkurve existiert, soll es möglich sein, die Punkte zu markieren, die zu stark von der Referenzkurve abweichen, wobei die Abweichung prozentual angegeben wird. Um diese Punkte zu berechnen, muss diese Referenzkurve zuerst entsprechend der maximalen Abweichung nach oben und unten skaliert werden, wofür eine Funktion existiert (*Scale Time Series*) und diese beiden skalierten Funktionen als Eingabe für die andere Funktion zu benutzen.

Damit ist die Sicht schon fast vollständig bestimmt, es fehlen nur noch die Parameter, mit welchen es möglich ist, die Sicht an die speziellen Wünsche anzupassen. Hierzu zählt unter anderem die Überschrift (*Title*) der Sicht und die Beschriftung (Einheiten) der x- und y-Achse (*x-Axis Title*, *y-Axis Title*).

In Abbildung 28 sind noch einmal die Eingaben und Parameter mit dem jeweiligen Typ auf einen Blick zusammengefasst. Die Abbildung 29 zeigt den Datenfluss einer möglichen Instanz, wobei die Abbildung so vereinfacht wurde, dass keine speziellen Funktionsinstanzen angegeben werden, sondern nur der Typ der Funktionsinstanz.

Das erzeugte Diagramm besteht zum Teil aus statischen Elementen, die für alle Diagramme gleich sind, und zum Teil aus dynamischen Elementen, die abhängig von den konkreten Daten sind. Zu den statischen Elementen zählen beispielsweise die Achsen des Koordinatensystems. Diese sind für jede Instanz an der gleichen Position und haben die gleiche Länge. Ebenso ist die Legende statisch. Schließlich befinden sich die Diagramm-Überschrift und die Einheiten der gleichen Position, wobei der Text abhängig von den Daten der jeweiligen Instanz ist. Alle weiteren Elemente sind

Phases Line Plot
Controlled Series : Time Series Phases And Activities : Phase And Activities Lower Border : Time Series (optional) Upper Border : Time Series (optional) Values (above) : List of Time Series (optional) Values (below) : List of Time Series (optional)
Title : String x-Axis Title: String y-Axis Title: String

Abbildung 28: Die Sicht „Phases Line Plot” mit den zugehörigen Eingaben und Parametern.

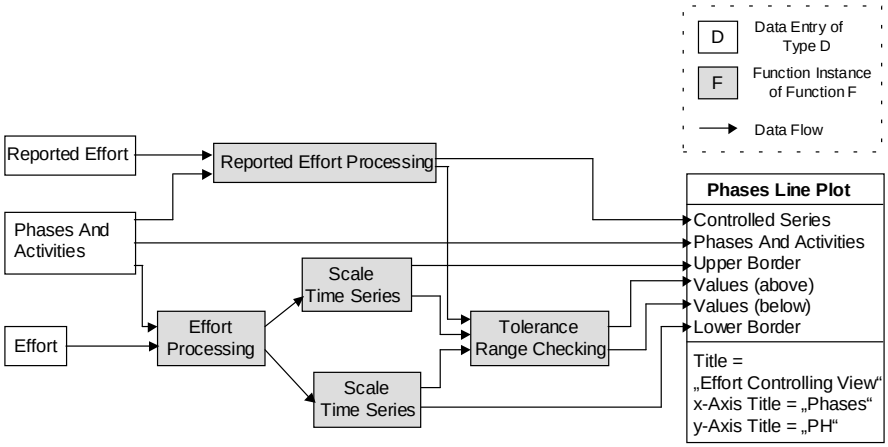


Abbildung 29: Datenfluss der Daten einer möglichen Instanz der Sicht „Phases Line Plot”.

abhängig von den konkreten Daten. So wird die Skalierung der y-Achse so berechnet, dass der maximal darstellbare Wert zehn Prozent größer ist, als der Maximal-Wert der Eingaben. Daraufhin wird rekursiv ein so genanntes Template (entspricht einer Prozedur oder Methode) aufgerufen, welches die Einteilung und Beschriftung der y-Achse in zehn Abschnitte übernimmt.

Ein ähnliches Verfahren wird für die Beschriftung der x-Achse mit den Phasen angewendet. So wird die Achse in so viele Abschnitte eingeteilt, wie Phasen vorhanden sind. Beide Werte (Skalierung der x-Achse und Skalierung der y-Achse) werden benötigt, um für die Werte der Kurven die Position im Koordinatensystem zu berechnen. In Abbildung 30 ist ein Beispiel einer mit dieser Sicht erzeugten SVG-Grafik zu sehen.

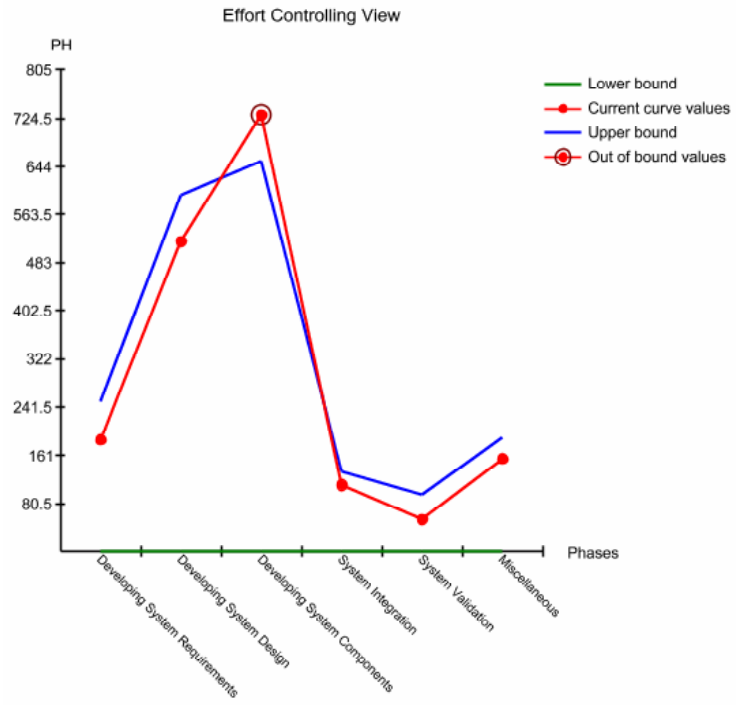


Abbildung 30: Mittels der Sicht „Phases Line Plot" erzeugte SVG-Grafik.

5.2.2 Aufwandsverteilung auf Phasen und Aktivitäten (*Phases Line Plot With Activities*)

Wie bereits in der Einleitung erwähnt wurde, hat diese Sicht einige Gemeinsamkeiten mit der vorangegangenen. Daher wird auf diese Teile nur kurz eingegangen und mehr Wert auf die Unterschiede gelegt und auch begründet, aus welchem Grund die beiden Sichten geteilt wurden.

Diese Sicht wurde ebenfalls als Liniendiagramm realisiert, aus den gleichen Gründen wie zuvor. Es gibt jedoch einen entscheidenden Unterschied: während die vorangegangene Sicht nur aus einem Diagramm besteht, gibt es in dieser Sicht ein Diagramm, das den Aufwand verteilt auf die Phasen zeigt, und außerdem für jede Phase ein Diagramm, welches die Verteilung des Aufwandes auf die Aktivitäten zeigt.

Die Eingaben der vorangegangenen Sicht, gehen auch in diese Sicht ein. Hierbei ist jedoch zu beachten, dass der tatsächliche Verlauf der Kurve (und der Referenzkurven) nicht nur die Werte für die Phasen enthält, sondern auch auf die Aktivitäten der Phasen aufgeschlüsselt werden muss. Da die Eingaben als Zeitreihe in die Sicht eingehen, muss der Zeitstempel die Phase und Aktivität kodieren. Da es möglich sein soll, anhand des Zeitstempels die Phase und Aktivität auf einfache Art und Weise zu bestimmen, wird die Nummer der Phase mit einem geeigneten Faktor

multipliziert und anschließend die Nummer der Aktivität hinzuaddiert. Dieser Faktor (*Phases Factor*) geht sowohl in die Konvertierungsfunktionen, als auch in die Sicht als Parameter ein und kann individuell festgelegt werden. Dieser Faktor muss größer sein, als die maximale Anzahl der Aktivitäten einer Phase.

Zusätzlich hat diese Sicht eine weitere Eingabe, die die maximalen Werte des erfassten Aufwands über alle Phasen und für jede einzelnen Phase enthält. Es wird der maximale Wert von den erfassten Aufwänden bestimmt, da diese Kurve, die einzige ist, die nicht optional ist, genausowenig wie die Maxima. Später werden in XSL die maximalen Werte mit dem jeweiligen Maximum der oberen Referenzkurve verglichen und der größere der beiden weiterhin als Maximum verwendet. Diese Eingabe geht wiederum als Zeitreihe (*Time Series*) ein, wobei anhand des Zeitstempels unterschieden wird, welches Maximum durch den Wert repräsentiert wird.

Diese Maxima sind auch ein Grund, aus welchem eine unabhängige Sicht notwendig ist. Es kann nicht unterschieden werden, ob in die Sicht nur die Daten für die Phasen eingehen oder auch diejenigen für die Aktivitäten. Bei der Transformation der Sicht muss unterschieden werden, ob das Diagramm für die Phasenübersicht generiert wird, oder das einer Phase. Der Grund für die Unterscheidung liegt darin, dass die beiden Typen unterschiedliche Fähigkeiten haben müssen. So muss es möglich sein von der Phasenübersicht zu den jeweiligen Phasendiagrammen zu gelangen und wieder zurück zur Phasenübersicht; dies ist auch nicht notwendig, wenn nur die Aufwandsverteilung auf die Phasen dargestellt werden soll.

Es wird jedoch immer versucht ein Diagramm für jede Phase zu generieren und damit auch auf die Maxima für die einzelnen Phasen zugegriffen, um die y-Achse zu skalieren. Diese sind jedoch nicht vorhanden, wenn nur die Aufwände der Phasen betrachtet werden. Ebensowenig gibt es für die Phasen ein Maximum für die obere Kurve.

Außerdem ist noch ein weiterer Parameter notwendig, da die Einheit der x-Achse für die Diagramme der einzelnen Phase eine andere ist als für die Gesamt-Übersicht und damit auch mittels einem Parameter anpassbar sein sollte.

Phases Line Plot With Activities
Controlled Series : Time Series Phases And Activities : Phase And Activities Lower Border : Time Series (optional) Upper Border : Time Series (optional) Values (above) : List of Time Series (optional) Values (below) : List of Time Series (optional) Maxima : Time Series
Title : String Phases Factor : int x-Axis Title 1: String x-Axis Title 2: String y-Axis Title: String

Abbildung 31: Die Sicht „Phases Line Plot With Activities" mit den zugehörigen Eingaben und Parametern.

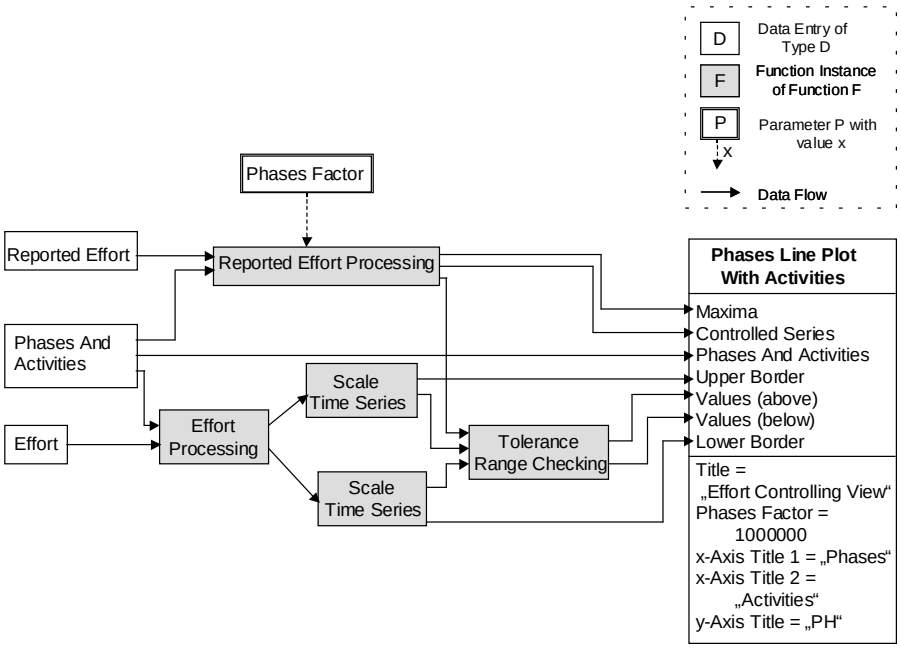


Abbildung 32: Datenfluss einer möglichen Instanz der Sicht „Phases Line Plot With Activities”.

In Abbildung 31 kann man noch einmal die Eingaben und Parameter der Sicht auf einen Blick sehen. Abbildung 32 zeigt einen möglichen Kontrollfluss für die Daten einer Instanz der Sicht. Auch diese wurde wieder vereinfacht, wie bereits die Sicht zuvor, so dass keine speziellen Funktionsinstanzen angegeben werden, sondern nur der Typ der Funktion.

Es wurde bereits zu Beginn erwähnt, dass eine Menge von Diagrammen erzeugt wird. Diese Diagramme werden in zwei Spalten angeordnet, wobei erst ab der zweiten Reihe die Diagramme für die einzelnen Phasen angeordnet werden. Der freie Platz in der ersten Reihe wird benutzt, um einen Link zu platzieren, mit dessen Hilfe die Sicht „gestartet” werden kann, wenn das automatische Skript, welches diese Aufgabe eigentlich übernehmen sollte, nicht durch den SVG-Viewer unterstützt wird (gilt für Batik 1.1.1). Jedes einzelne Diagramm wird entsprechend verkleinert, so dass zunächst alle Diagramme angezeigt werden (siehe Abbildung 33). Dies ist notwendig, da die Sicht ansonsten in Batik 1.1.1 nicht korrekt angezeigt werden kann. Der Verkleinerungsfaktor wird automatisch berechnet, abhängig von der Anzahl der Phasen. Nach dem Erzeugen der SVG-Datei wird die so genannte ViewBox so gesetzt, dass allein der Ausschnitt mit der Phasenübersicht vergrößert angezeigt wird. Hierzu wird automatisch nach dem Laden - falls möglich - ein Skript gestartet, welches, wie bereits erwähnt, nicht unter Batik 1.1.1 funktioniert. Im letzteren Fall muss der Benutzer einen Link anwählen, welcher speziell für diesen Zweck erzeugt wird und auch nur in diesem Fall zu sehen ist.

Die einzelnen Phasen dieser Übersicht sind jeweils mit einem Link versehen, so

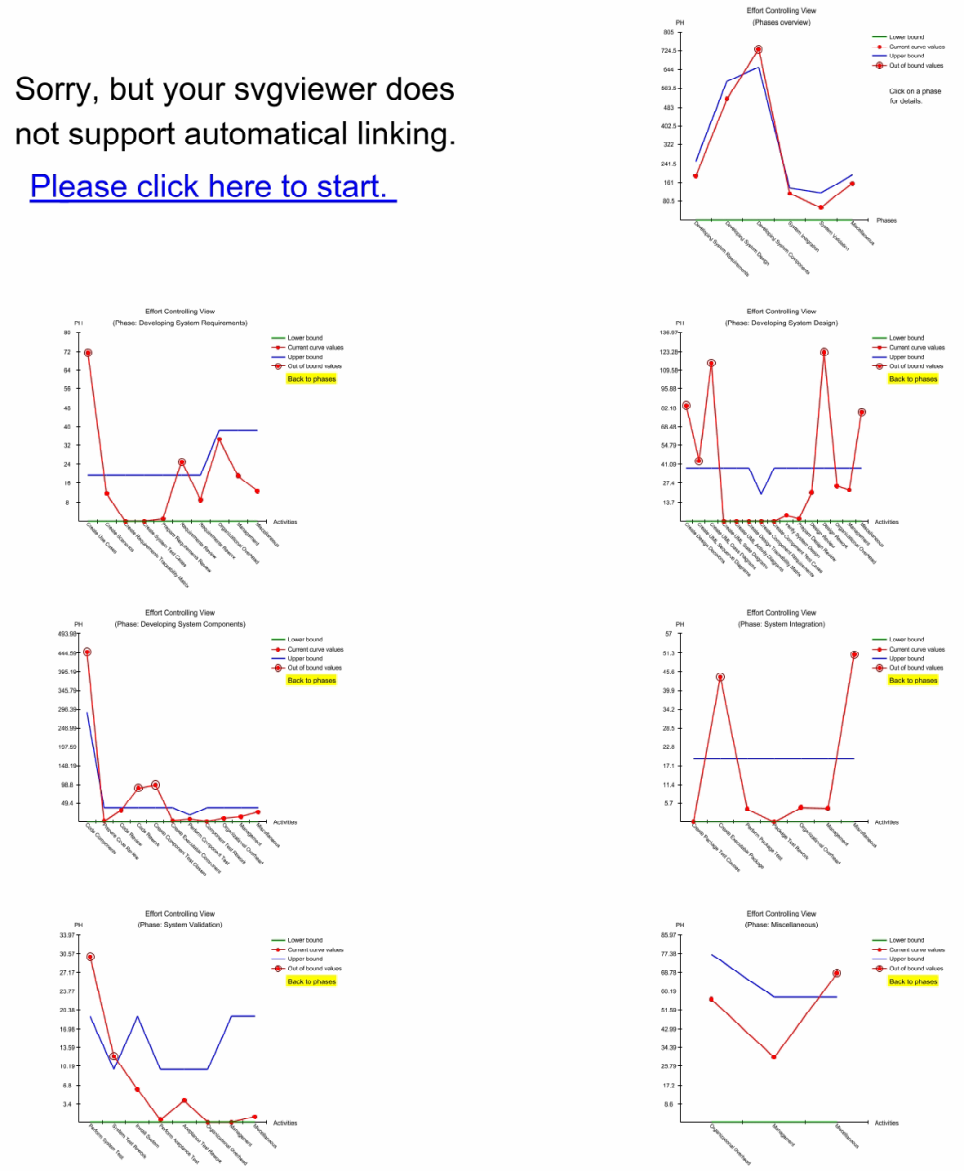


Abbildung 33: Anordnung der Phasen-Diagramme, welche durch die Sicht „Phases Line Plot With Activities" erzeugt werden.

dass beim Anklicken der Phasen, das jeweilige Diagramm, das die Aufwandsverteilung auf die Aktivitäten dieser Phase zeigt, angezeigt wird. Dieses Diagramm enthält auch wieder einen Link, mit dem man zurück zur Phasenübersicht gelangt.

Da viele Elemente häufig verwendet werden, werden diese als Symbole definiert und für jedes Diagramm nur noch an der entsprechenden Stelle platziert. Für die Skalierung der y-Achse wird die zusätzliche Eingabe der Maxima benutzt, welche

mit dem größten Wert der Oberschranke verglichen wird und der größere weiter verwendet wird. Das Verfahren ist ansonsten das gleiche wie bei der vorangegangenen Sicht.

Ein Beispiel für die resultierende SVG-Grafik ist in Abbildung 33 zu sehen. Außerdem wird jeweils die Phasenübersicht und das Diagramm einer Phase zusätzlich in den Abbildungen 34 und 35 gezeigt.

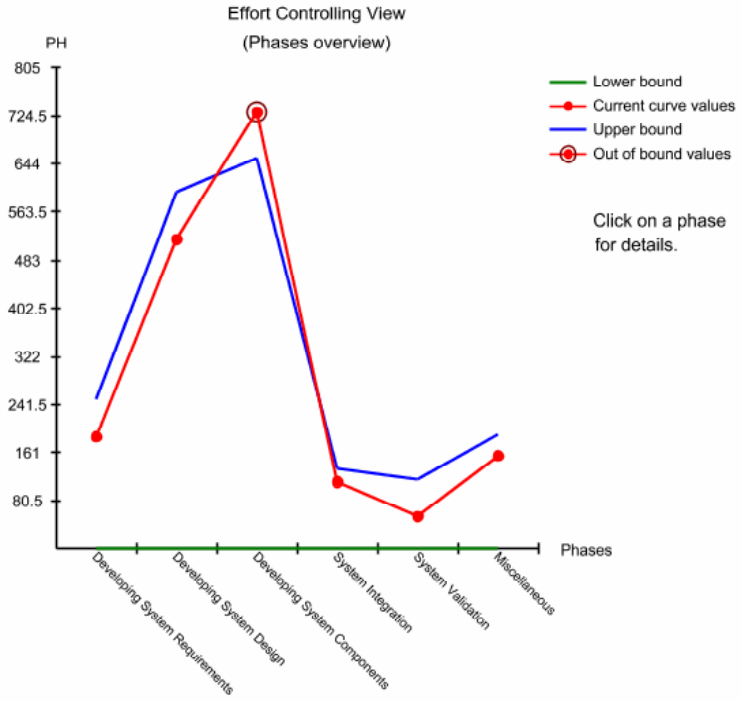


Abbildung 34: Ausschnitt der durch die Sicht „Phases Line Plot With Activities" erzeugten SVG-Grafik, Phasenübersicht.

5.2.3 Fehlerverteilung auf Phasen und Affiliation mit zugehörigem Aufwand (*Defect View*)

Eine weitere Sicht zeigt die Verteilung der Fehler auf Phasen (Phase, in der der Fehler gefunden wurde) und Affiliation (Phase, in der das Dokument erstellt wurde, in dem der Fehler gefunden wurde). Für diese ist die Darstellung als Balkendiagramm am Besten geeignet, da die Daten nach zwei Einteilungen sortiert werden müssen (siehe auch Abbildung 12).

Als Eingabe benötigt die Sicht sowohl die Phasen (*Phases and Activities*) als auch die Affiliation (*Categories*), welche direkt in die Sicht eingehen. Außerdem müssen die Fehler, nachdem sie durch eine Funktion (*Defect Sorting (Affiliation and Phases)*) sortiert, der Aufwand addiert und die Fehler gezählt wurden, als Eingabe in

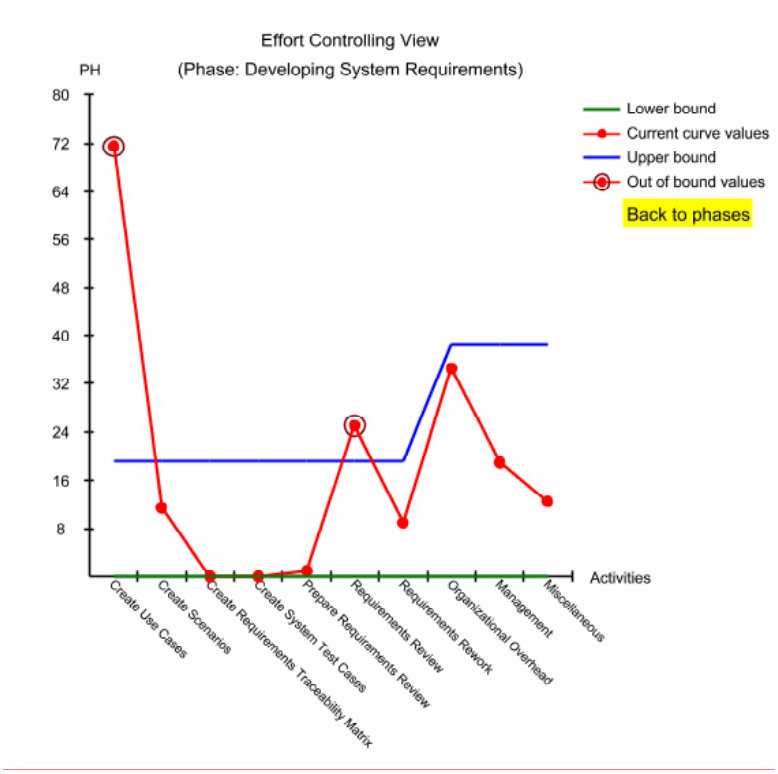


Abbildung 35: Ausschnitt der durch die Sicht „Phases Line Plot With Activities" erzeugten SVG-Grafik, der ersten Phase.

die Sicht eingehen. Diese Eingabe ist vom Typ *Defects*, welcher jeder Kombination von Phasen und Affiliation einen Eintrag mit der Summe der Fehler und der Summe des Aufwands, der benötigt wird, um diese Fehler zu beheben, zuordnet. Alle Eingaben sind nicht optional, da ansonsten wichtige Informationen zur Darstellung fehlen.

Neben den Eingaben, werden noch einige Parameter benötigt, um die Sicht an die jeweilige Anwendung anzupassen. Die Parameter sind die gleichen wie schon bei der Sicht „Aufwandsverteilung auf Phasen" (siehe 5.2.1), nämlich der Titel (*Title*), und die Einheiten der beiden Achsen (*x-Axis Title, y-Axis Title*).

Auch für diese Sicht ist eine Übersicht mit allen Eingaben und Parametern gegeben (siehe Abbildung 36) und ein möglicher vereinfachter Kontrollfluss (siehe Abbildung 37).

Das erzeugte Diagramm besteht wieder aus statischen und dynamischen Elementen. Statische Elemente sind das Koordinatensystem sowie wie die Legende. Die Skalierung der y-Achse und auch der x-Achse wird abhängig von den realen Eingabe-Daten durchgeführt. Ebenso wird die Balkenbreite automatisch berechnet, so dass in jeder Phase für alle Affiliation ein Balken eingezeichnet werden könnte. In der Realität kann dies nicht passieren, da nicht jede Kombination von Phasen und Affiliation Sinn ergibt (ein Fehler kann nicht früher entdeckt werden, als er

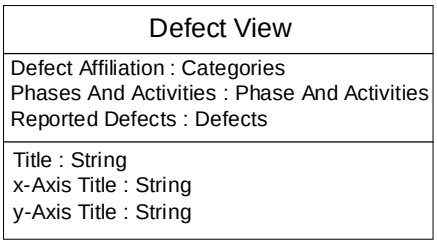


Abbildung 36: Die Sicht „Defect View" mit den zugehörigen Eingaben und Parametern.

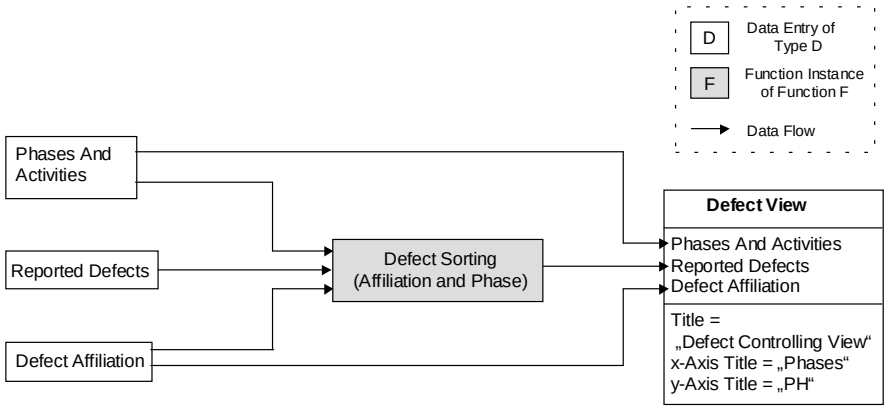


Abbildung 37: Datenfluss einer möglichen Instanz der Sicht „Defect View".

verursacht wurde).

Für die primäre Einteilung der x-Achse werden die Phasen gewählt und auch benutzt, um die Achse zu beschriften. Die Kodierung der Affiliation wird über verschiedene Farben durchgeführt. Bis jetzt ist die Anzahl der Affiliation noch auf vier beschränkt. Anhand einer der Affiliation zugeordneten Nummer wird eine von vier Farben ausgewählt.

In der Abbildung 38 ist eine SVG-Grafik zu sehen, die durch diese Sicht erzeugt wurde.

5.2.4 Zeitlicher Projektverlauf (*Project Plan*)

Die vorletzte Sicht, die im Laufe meiner Arbeit implementiert wurde, zeigt den zeitlichen Projektverlauf, also Beginn und Ende, der einzelnen Phasen bzw. Aktivitäten. Für diese Sicht bietet sich offensichtlich ein Zeitplan zur Darstellung an (siehe 1.8.7).

Diese Sicht hat wieder optionale und nicht-optionale Eingaben. Nicht-optional sind z.B. die Phasen und Aktivitäten, welche auf der y-Achse aufgetragen werden. Da nicht für jede Phase und Aktivitäten Einträge existieren, wird eine Funktion (*Convert Project Plan*) benötigt, die diejenigen Phasen und Aktivitäten herausfil-

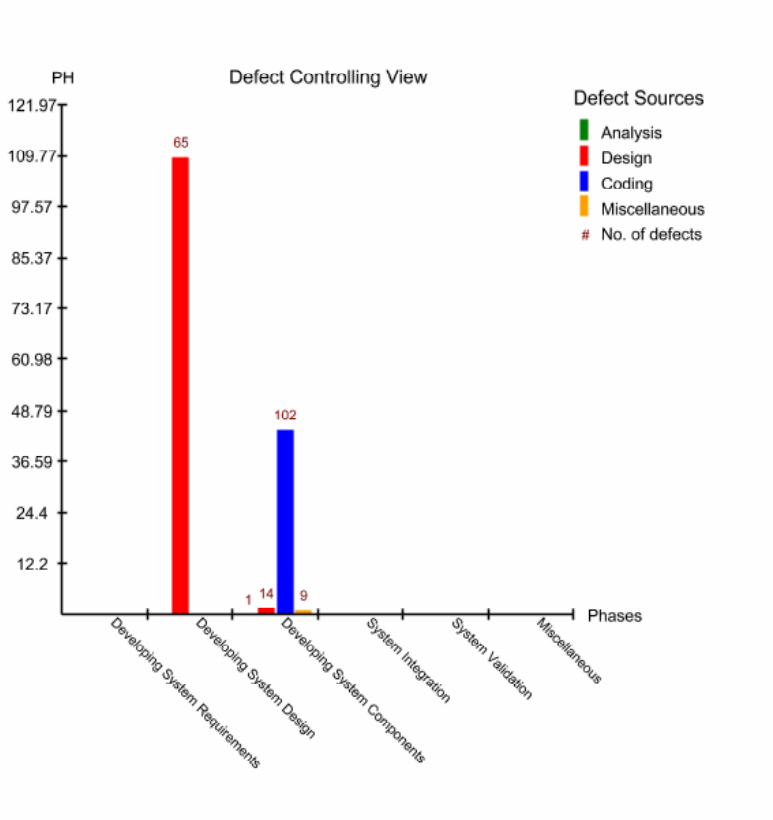


Abbildung 38: Mittels der Sicht „Defect View" erzeugte SVG-Grafik.

tert, für welche auch Einträge existieren, und in einer Menge von Kategorien (*Category Series*) zusammenfasst. Die eigentliche Aufgabe der Funktion ist jedoch eine andere, nämlich die Konvertierung eines Projektplans in eine Liste von Zeitreihen (*List of Time Series*). Jede Zeitreihe entspricht den Einträgen einer Phase bzw. Aktivität. Hierfür wird jeweils eine eigene Reihe benötigt, da es möglich ist, dass zu einem späteren Zeitpunkt nochmals Zeit für eine Phase oder Aktivität eingeplant wird.

Eine optionale Eingabe der Sicht ist ein Referenzplan, der z.B. den aktuellen Verlauf repräsentiert, wobei auch dieser Plan zuerst durch die Funktion konvertiert werden muss, damit er vom gleichen Typ ist.

Schließlich gehen noch die üblichen Parameter in die Sicht ein, nämlich der Titel des Diagramms (*Title*) und die Beschriftungen der beiden Achsen (*x-Axis Title, y-Axis Title*).

In Abbildung 39 ist die Zusammenfassung aller Eingaben und Parameter der Sicht zu sehen und in Abbildung 40 der Datenfluss einer möglichen Instanz, wobei hier die beiden Pläne identisch sind.

Eine Besonderheit bei der Implementierung besteht darin, dass darauf verzichtet wurde, für die y-Achse eine feste Größe anzunehmen, sondern vielmehr der Abstand

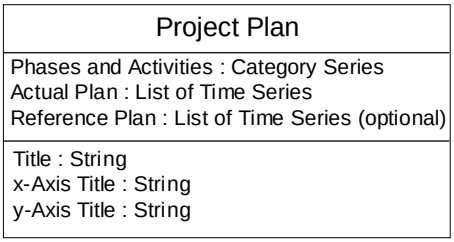


Abbildung 39: Die Sicht „Project Plan” mit den zugehörigen Eingaben und Parametern.

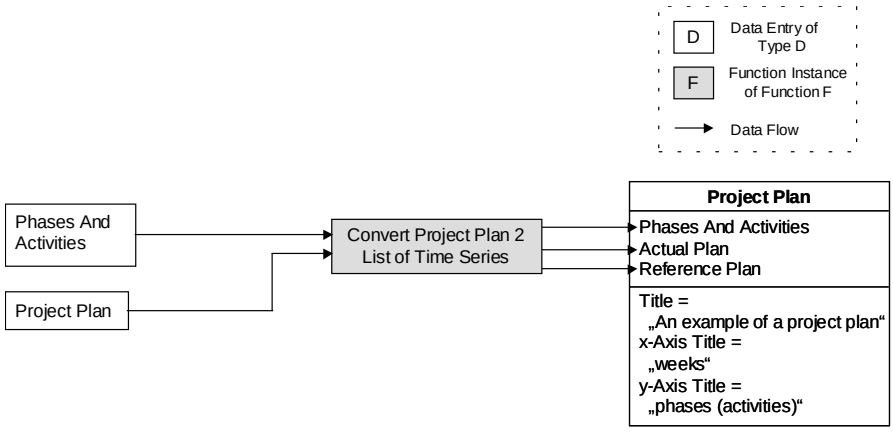


Abbildung 40: Datenfluss einer möglichen Instanz der Sicht „Project Plan”, wobei die beiden Pläne identisch sind.

zwischen den Einträgen konstant ist und die Länge somit abhängig von der Anzahl der Einträge berechnet wird, ebenso wie die notwendige Höhe der Grafik. Der Grund hierfür liegt darin, dass die Beschriftung der y-Achse noch lesbar sein sollte und daher nicht zu klein skaliert werden oder gar übereinander liegen darf.

Die Beschriftung der y-Achse war noch in anderer Hinsicht problematisch. Da die Anzahl der Buchstaben für die Phasenbezeichnung und Aktivitätenbeschreibung stark variiert, ist es keine besonders gute Lösung die y-Achse soweit zu verschieben, dass auch die längste Bezeichnung am Stück ausgegeben werden kann, ohne in das Koordinatensystem hineinzuragen. Stattdessen wurden die einzelnen Worte auf mehrere Zeilen verteilt. Leider unterstützt SVG keinen automatischen Zeilenumbruch, daher musste dieser mit Hilfe von XSL durchgeführt werden. Die einfachste Lösung würde darin bestehen, jedes Wort ein eine eigene Zeile zu schreiben. Dies führt jedoch zu einem großen Abstand zwischen den Einträgen, da manche Phasen- bzw. Aktivitätsbezeichnungen aus vielen - zum Teil auch kurzen - Worten bestehen wobei jedoch Platz für 21 Zeichen ist. Die Lösung sieht vor, dass zuerst überprüft wird, ob der noch nicht ausgegebene Text weniger als 22 Zeichen hat. Falls dies der

Fall ist, wird er direkt ausgegeben. Ansonsten wird überprüft, ob der Text vor dem zweiten auftauchenden Leerzeichen weniger als 22 Zeichen hat und dann ausgegeben. Sollten beide Bedingungen nicht zutreffen, wird der Text bis zum ersten Leerzeichen ausgegeben. Für den restlichen Text wird jeweils das gleiche Verfahren angewendet. Sollte der unwahrscheinliche Fall auftreten, dass kein Leerzeichen mehr vorkommt und der Text auch nicht weniger als 22 Zeichen hat, wird er trotzdem ausgegeben.

Die Skalierung der x-Achse geschieht automatisch, wobei die maximale Anzahl an Wochen verwendet wird, um den Plan zu skalieren und in zehn Abschnitte einzuteilen. Da es relativ schwierig ist, die genauen Zeitpunkte, wann eine Phase beginnt und endet, abzulesen, wird noch unterstützt, dass der Betrachter eine Linie anwählen kann und in einem Fenster die genauen Zeitpunkte für Beginn und Ende angezeigt bekommt. Es wurde ein Fenster statt beispielsweise eines auftauchenden Textes gewählt, damit die die SVG-Grafik auch unter Batik 1.1.1 betrachtet werden kann.

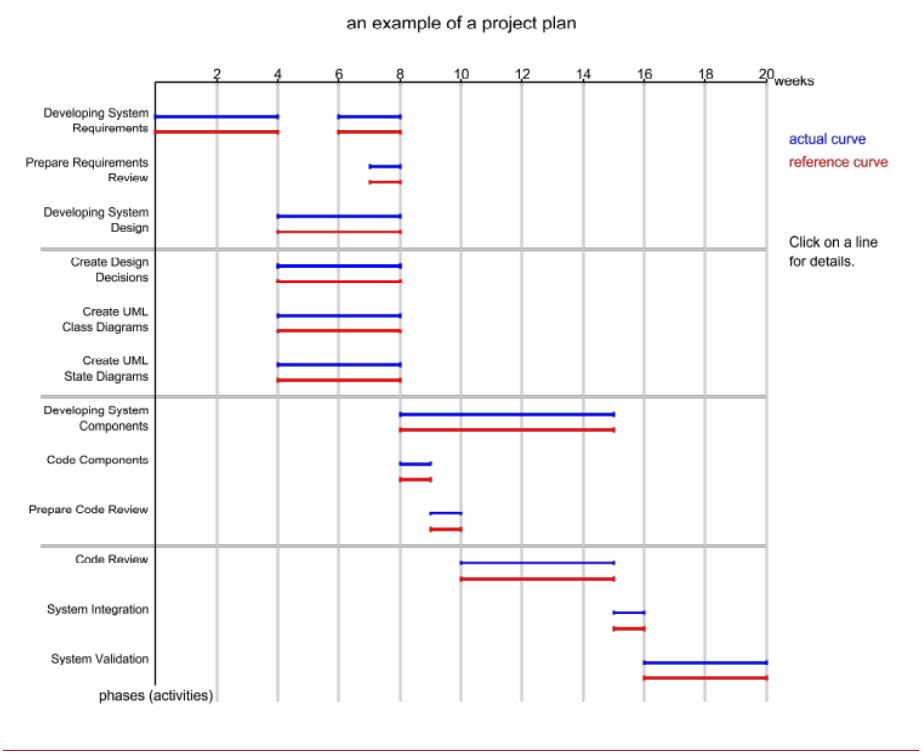


Abbildung 41: Mittels der Sicht „Project Plan" erzeugte SVG-Grafik.

Ein Beispiel einer SVG-Grafik, die mit Hilfe dieser Sicht erzeugt wurde, ist in Abbildung 41 zu sehen.

5.2.5 Kreisdiagramm (*Pie Chart*)

Die letzte Sicht, die implementiert wurde, ist ein Kreisdiagramm und kann vielseitig angewendet werden. So sind auch bereits zwei Sichteninstanzen vorhanden. Die

erste zeigt die Verteilung der Fehler auf die verschiedenen Fehlerklassen, die zweite die Verteilung entsprechend dem jeweiligen Status. Aber auch weitere Verteilungen können mit Hilfe des Kreisdiagramms dargestellt werden.

Die Sicht ist sehr einfach und benötigt nur eine Eingabe und einen Parameter. Der Parameter dient lediglich dazu, den Titel des Diagramms anzupassen (*Title*). Die Eingabe besteht aus realen Daten, wobei es sich um eine Reihe von Kategorien handelt (*Category Series*). Ein Eintrag in einer solchen Reihe besteht aus einer Kategorie, die der Bezeichnung für die Teilmenge entspricht, und einem zugehörigen Wert, der angibt, wie häufig diese Kategorie vertreten ist. Dadurch wird eine Funktion (*Defects Processing*) notwendig, die die realen Daten in den benötigten Typ zu konvertieren. Diese Funktion zählt für jede Teilmenge, wie häufig diese auftritt, und ordnet diesen Wert der entsprechenden Bezeichnung zu.

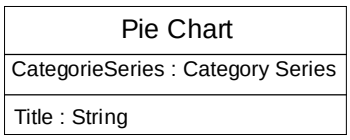


Abbildung 42: Die Sicht „Pie Chart” mit zugehörigen Eingaben und Parametern.

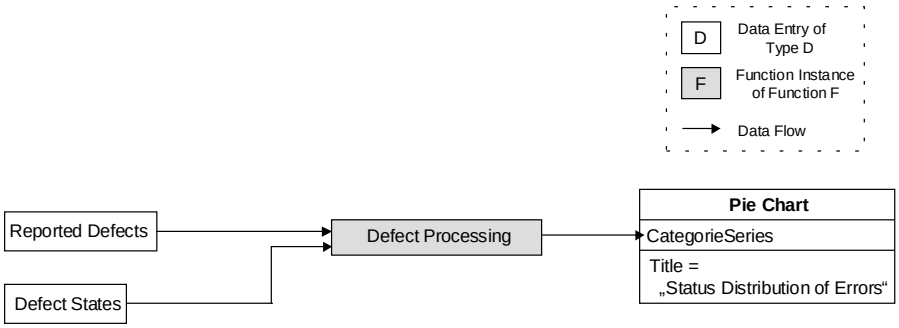


Abbildung 43: Datenfluss einer möglichen Instanz der Sicht „Pie Chart”.

In den Abbildungen 42 und 43 sind noch einmal die Sicht mit allen Eingaben und Parametern zu sehen, sowie der Datenfluss einer möglichen Instanz.

Das Erzeugen des Kreisdiagramms geschieht rekursiv, damit bekannt ist, welche Koordinaten das Ende des vorangegangen Kreisabschnitts hat. Außerdem muss für jeden Anteil, der dargestellt werden soll, überprüft werden, ob er mehr als die Hälfte des Gesamten ausmacht, da dies einen Unterschied für den entsprechenden SVG-Befehl bedeutet.

Eine Besonderheit in der Implementierung besteht darin, dass die Farbe für die Abschnitte automatisch berechnet wird, damit nicht für jede Instanz Änderungen vorgenommen werden müssen, weil die Anzahl sich ändert. Die Farben werden in RGB-Werten angegeben. Ein Ziel bei der Farbberechnung war auch, dass nebenein-

anderliegende Abschnitte einen möglichst guten Kontrast haben. So wird zu Beginn der Rot-Anteil und Grün-Anteil auf 255 und der Blau-Anteil auf Null gesetzt. Bei jedem Aufruf wird der Rot-Anteil um 16 erniedrigt und der Blau-Anteil um 16 erhöht. Die Berechnung des Grün-Anteils ist etwas komplexer, um den Kontrast noch weiter zu erhöhen. Es wird eine alternierende Funktion verwendet, die sich dem Wert 128 von beiden Seiten nähert. Durch diese Art der Berechnung ist die Anzahl der Abschnitte auf maximal 16 beschränkt. Weitere Abschnitte haben immer im Wechsel die gleiche Farbe und können damit nicht mehr unterschieden werden.

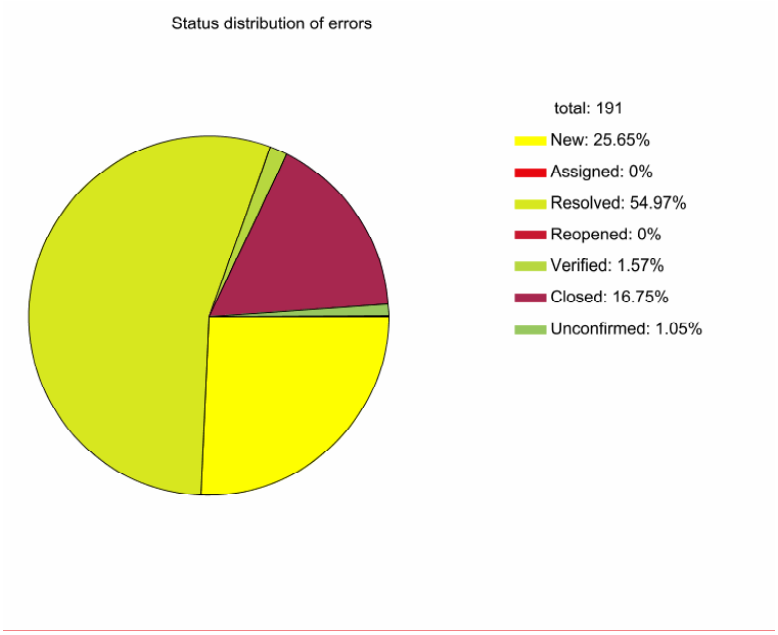


Abbildung 44: SVG-Grafik, die mit Hilfe der Sicht „Pie Chart" erzeugt wurde, zeigt die Fehlerverteilung auf die Fehlerstatus.

In Abbildung 44 ist ein Beispiel einer SVG-Grafik zu sehen, die durch diese Sicht erzeugt wurde und die Verteilung der Fehler pro Status zeigt.

5.3 Bewertung der Sichten

In diesem Abschnitt werden die zuvor beschriebenen Sichten anhand den Anforderungen aus 3.1 analog zu den Sichten aus der Literatur bewertet. Im Gegensatz zu letzteren werden jedoch alle Anforderungen berücksichtigt, da diese Sichten im Einsatz überprüft werden können und außerdem die Implementierung bekannt ist, was z.B. Aussagen über die Robustheit erlaubt.

Die Tabelle 23 zeigt eine Zusammenfassung der Bewertung der einzelnen Sichten, wobei dies ausführlicher im Folgenden ausführlich erläutert wird und auf die einzelnen Punkte näher eingegangen wird.

	Phases Line Plot	Phases Line Plot With Activities	Defect View	Project Plan	Pie Chart
Übersichtlichkeit	+	+	+	+	+
Verteilung	nicht nötig	+	nicht nötig	nicht nötig	nicht nötig
Vergleichsmöglichkeit	Vergleichskurve; obere und untere Schranke	Vergleichskurve; obere und untere Schranke	-	Referenzplan	Teilmengenvergleich
Hervorhebung	+	+	Vergleichsbedingt	-	nicht möglich
Skalierung	keine	Verkleinerung der einzelnen Diagramme	-	+	nicht nötig
Robustheit	+	+	+	+	nicht nötig
Konsistenz	Konsistenztests	Konsistenztests	Konsistenztests	Konsistenztests	Konsistenztests
zielorientierte Darstellung	+	+	+	+	+
Variabilität/ Anpassbarkeit	Titel, Beschriftungen	Titel, Beschriftung, Kodierung des Zeitstempels	Titel, Beschriftung	Titel, Beschriftung	Titel

Tabelle 23: Bewertung der Sichten.

Übersichtlichkeit und Verteilung

Es werden in allen Sichten nur ausgewählte Daten eines Projektes zusammen mit evtl. vorhandenen Vergleichsdaten eingetragen. Damit werden nur wenige Daten gleichzeitig dargestellt und somit sind alle Diagramme übersichtlich. Nur im Fall der Sicht „Phases Line Plot View With Activities" werden mehrere Diagramme verwendet, um die Aufwandsverteilung auf die Aktivitäten einer Phase darzustellen, da ansonsten nur wenige Daten gleichzeitig dargestellt werden und dies damit in einem Diagramm möglich ist.

Vergleichsmöglichkeit

Außer für die Sicht „Defect View" gibt es Möglichkeiten die Daten zu vergleichen. Hierbei ist zu unterscheiden zwischen der Sicht „Pie Chart", bei der ein Vergleich der einzelnen Teilmengen untereinander möglich ist und den restlichen Sichten, bei denen Referenzkurven eingetragen werden. Diese Referenzkurven können sowohl geplante Verläufe als auch Schranken sein. Da bei der Sicht „Defect View" der Schwerpunkt darin liegt, die Phasen zu identifizieren, in denen viele Fehler mit hohem resultierenden Aufwand verursacht wurden, und Aussagen über die Qualität der Produkte zu treffen, ist es weniger wichtig, Vergleichsdaten mit anzubieten (siehe auch 4.3.3).

Hervorhebung

Wie bereits zuvor bei der Bewertung der Sichten aus der Literatur müssen Vergleichsdaten vorhanden sein, um abweichende Werte markieren zu können, damit entfällt dies für die Sicht „Defect View". Auch für ein Kreisdiagramm werden keine Werte markiert, da der Vergleich nur zwischen den Teilmengen möglich ist. Für die Sicht „Project Plan" wird dies überhaupt nicht unterstützt. Der Grund hierfür liegt darin, dass es dadurch dass die beiden Pläne untereinander eingetragen werden, sofort ersichtlich ist, zu welchen Zeitpunkten der Plan nicht eingehalten wird. Für die anderen beiden Sichten werden jeweils die abweichenden Werte markiert.

Skalierung

Eine Skalierung der Sichten findet nur in zwei Fällen statt, nämlich für die Sicht „Phases Line Plot With Activities" und „Project Plan". Bei der ersteren bedeutete dies, dass die einzelnen Teildiagramme so verkleinert werden, dass alle angeordnet werden können. Dies ist möglich, da das anzuzeigenden Teildiagramm jeweils wieder so vergrößert werden, dass es die gesamte so genannte Viewbox einnimmt und somit wieder gut zu erkennen ist. Bei der zweiten Sicht wird das Diagramm so vergrößert, dass alle Einträge im Projektplan untereinander angeordnet werden können. Bei einem Kreisdiagramm ist dies nicht nötig, da die Gesamtanzahl aller Daten immer auf 100% normiert wird. Für die anderen beiden Sichten findet keine Skalierung statt.

Robustheit

Der nächste Bewertungspunkt, die Robustheit, ist wieder für die letzte Sicht „Pie Chart" nicht notwendig, da nur die Daten berücksichtigt werden, die vorhanden sind.

Für die anderen Sichten ist dies jeweils gegeben, da die Daten, die nicht-vorhanden sind durch verschiedene Funktionen auf den Wert Null gesetzt werden und somit alle dargestellt werden.

Konsistenz

Die Konsistenz der verschiedenen Sichten, kann mittels Konsistenzchecks überprüft werden. Momentan sind jedoch noch keine implementiert (siehe auch 6.2).

Zielorientierte Darstellung

Die Sichten sind alle zielorientiert in ihrer Darstellung, wobei dies im Einzelnen noch genauer begründet wird.

Die Sichten „Phases Line Plot" und „Phases Line Plot With Activities" sollen jeweils den Projektmanager und den Projektplaner bei ihrer Arbeit unterstützen. Der Projektplaner kann durch die Markierung der abweichenden Aufwandswerte erkennen, ob eine Anpassung des Planes notwendig ist und mehr Zeit - und damit verbunden - auch mehr Aufwand für die einzelnen Phasen bzw. Aktivitäten eingeplant werden muss. Ähnliches gilt für den Projektmanager, der leicht feststellen kann, ob eine Umverteilung des Personals notwendig ist.

Bei der Sicht „Defect View" ist es möglich, auf einen Blick, diejenigen Phasen zu erfassen, in denen Fehler einen großen Aufwand zur Folge haben (auch ohne Hervorhebung möglich). Dies kann sowohl der Projektmanager dazu benutzen, um die Phasen zu bestimmen, für welche zukünftig mehr Aufwand (mehr Entwickler) für das Review eingeplant werden muss und gegebenenfalls auch andere Werkzeuge und Methoden eingesetzt werden müssen. Der Qualitätssicherer kann erkennen wie gut die verwendeten Projekte und Prozesse sind und damit Aussagen über die Qualität der resultierenden Produkte machen und überprüfen, ob die Ziele bezüglich der Qualität auch eingehalten werden.

Durch den direkten möglichen Vergleich des geplanten Projektverlaufs zusammen mit dem aktuellen, ist es möglich zu überprüfen, inwieweit diese miteinander übereinstimmen und ob eine Anpassung des Plan notwendig ist. Dies ist eine Aufgabe des Projektplaners, welcher den tatsächlichen Verlauf auch als Anhaltspunkt benutzen kann, den Plan anzupassen. Der Projektmanager kann den Plan überwachen und gegebenenfalls durch Umverteilen von Personal dessen Einhaltung sicherstellen.

Die letzte Sicht soll in der Sichteninstanz „Fehlerverteilung nach Status" (siehe Abbildung 44) sowohl den Qualitätssicherer als auch den Entwickler unterstützen. Letzterer kann leicht erkennen, ob er Zeit in die Behebung von Fehlern investieren muss (viele Fehler mit dem Status: New) oder für das Verifizieren (noch viele Fehler mit dem Status: Resolved"). Durch die Möglichkeit die Teilmengen miteinander zu vergleichen kann erkannt werden, wie sich die Fehler auf die unterschiedlichen Status verteilen und damit auch wie viele Fehler nochmals auftreten, da die Lösung nicht fehlerfrei war (Status: Reopened). Dies ermöglicht dem Qualitätssicherer Aussagen über das Projekt zu machen.

Variabilität/Anpassbarkeit

Die Anpassbarkeit der einzelnen Sichten ist hauptsächlich beschränkt auf wenige Parameter, nämlich auf Titel und Achsenbeschriftungen (außer „Pie Chart"). Die

Sicht „Phases Line Plot With Activities" bietet noch die Möglichkeit an, den Faktor, der zur Kodierung des Zeitstempels benutzt wird (siehe auch Abbildung 32) anzupassen. Es gibt jedoch keine Möglichkeit den Diagrammtyp zu ändern oder auch andere Farben zu verwenden, ohne an der Implementierung zu ändern.

5.4 Einsatz der Sichten

Für die Evaluation wurde der SPCC bereits in einem Praktikum eingesetzt, wobei nur zwei Sichten zur Verfügung standen, nämlich die Aufwandsverteilung auf die Phasen und die Fehlerverteilung auf Phasen und Affiliation. Die anderen Sichten wurden erst später fertiggestellt, sollen jedoch in weiteren Praktika eingesetzt werden. So ist bereits der Einsatz im Softwarepraktikum im Sommersemester 2003 geplant. Es ist daher auch noch nicht möglich viele Aussagen über die Sichten zu machen, anhand der Rückmeldungen am Ende des Praktikums können jedoch einige grundsätzliche Dinge festgestellt werden, welche hier zusammengefasst werden.

Da das Praktikum bereits begonnen hatte, bevor die Messdatenerfassung und die Sichten zur Verfügung standen, wurden diese nur in sehr geringem Maße verwendet. Im Allgemeinen wurde jedoch die Aufwandssicht häufiger eingesetzt und als nützlicher empfunden, als die Fehlerverteilungssicht. Es wurde angeregt, dass die Aufwandsverteilung statt auf Phasen auf Aktivitäten dargestellt werden soll. Da diese Sicht erst zu spät verfügbar war, konnte sie nicht mehr eingesetzt werden.

Ähnliches gilt für die Sicht des Projektplans. Es kam der Wunsch auf, dass die Verteilung der Aktivitäten auf die unterschiedlichen Projektwochen dargestellt werden sollte. Als Lösung hierfür kann zukünftig die zuvor entwickelte Sicht (*Project Plan*, siehe 5.2.4) dienen. Beide Sichten stehen inzwischen jedoch zur Verfügung und können für weitere Praktika verwendet werden, in welchen sich zeigen wird, in wieweit diese nützlich sind und den Bedürfnissen entsprechen.

Die Sicht, welche die Fehlerverteilung zeigt, wurde kaum eingesetzt, da den Studenten den Zweck der Sicht nicht ausreichend klar war und diese daher keinen Nutzen in der Sicht sahen, wobei die Sicht auch mehr für den Projektmanager und den Qualitätssicherer von Interesse ist und weniger für die einzelnen Entwickler.

Anhand der Rückmeldungen ist zu ersehen, dass die implementierten Sichten, im Großen und Ganzen diejenigen sind, die hauptsächlich von Interesse sind. Die verwendete Darstellung ist jeweils auch verständlich.

6 Zusammenfassung und Ausblick

Abschließend wird noch einmal einen zusammenfassender Überblick über die Arbeit gegeben und auch mögliche Erweiterungen und Verbesserungen vorgestellt.

6.1 Überblick

Die Arbeit befasst sich zunächst damit, dass die notwendigen Grundlagen, um die Arbeit einordnen zu können, ausführlich erläutert werden. Hierzu gehört vor allem die Vorstellung des SPCC und seine Einordnung in eine Software-Entwicklungsumgebung. Neben Metriken und dem GQM-Ansatz werden auch die verschiedenen Diagrammtypen zusammen mit ihren Vor- und Nachteilen ausführlich beschrieben.

Kapitel 2 umfasst die Literaturrecherche, wobei zuerst die Vorgehensweise hierfür erläutert wurde und im Anschluss die Ansätze der resultierenden Artikel. Diese Ansätze wurden miteinander verglichen, sowohl in Hinblick auf die Motivation, die den Ansätzen zu Grunde liegt, als auch auf die verwendeten Metriken und Darstellungsarten. Schließlich wurden die Sichten aufgelistet und kurz vorgestellt.

Aus den Artikeln, die nicht die Grundlage für Kapitel 2 waren, wurden verschiedene allgemeine Anforderungen an Sichten abgeleitet, diese wurden ausführlich beschrieben und jeweils auch begründet, aus welchem Grund diese notwendig sind. In einem nächsten Schritt wurde zuerst eine Klassifikation für Sichten vorgestellt, um anschließend ein Meta-Modell zu entwickeln, mit welchem die Zusammenhänge zwischen verschiedenen Sichten dargestellt und verdeutlicht werden können. Abschließend wurden die Sichten aus der Literatur anhand einer Teilmenge der abgeleiteten Anforderungen beschrieben. Da für einige Anforderungen der praktischen Test notwendig ist, wurde auf diese hierbei verzichtet.

In Kapitel 4 wurden zuerst die einzelnen Rollen beschrieben, die an einem Projekt beteiligt sind und auch eine Auswahl getroffen, welche der Aufgaben für welche Rollen, unterstützt werden sollen. Zusätzlich wurde eine Menge von Metriken aus verschiedenen Kategorien definiert, die als Grundlage für die späteren Sichten dienen sollen. Hierbei wurden alle Kategorien betrachtet, in welche sich die Metriken aus der Literatur einordnen lassen, und jeweils mindestens eine Metrik aus diesen Kategorien verwendet. Dabei können Metriken auch in mehr als eine Kategorie eingeordnet werden. Schließlich wurde eine Menge von Sichten vorgestellt, hierfür wurde auch die Klassifikation aus dem vorangegangenen Kapitel verwendet, um diese zu beschreiben. Für jede Sicht wurden unter anderem die Rolle, die sie unterstützen soll, als auch die verwendeten Metriken, verwandte Sichten und mögliche Darstellungsformen beschrieben.

Von diesen Sichten wurde ein Teil realisiert. Diese wurden zusammen mit den Eingaben einer Sicht und den Parametern beschrieben und auch auf Besonderheiten in der Implementierung eingegangen. Es wurde eine eigene Visualisierungskette, die für jede Sicht mindestens eine Instanz enthält, erstellt und in den SPCC integriert. Ein Teil der Sichten wurde bereits in einem Praktikum an der Universität Kaiserslautern erprobt und die dort gesammelten Erfahrungen sind bereits zum größten Teil in die bestehenden Sichten integriert worden.

Die resultierenden Sichten können von verschiedenen Rollen benutzt werden, um ihre Aufgaben durchzuführen. So kann der Projektplaner die Sichten benutzen, um einen Projektplan zu erstellen und gegebenenfalls anzupassen. Auch der Pro-

jektmanager wird unterstützt, da er die Einhaltung des erstellten Projektplans mit Hilfe der Sichten kontrollieren kann und auch diejenigen Phasen identifizieren kann, welche mehr Aufwand benötigen, als geplant wurde, um dies in Zukunft zu vermeiden, indem mehr Personal zugeteilt wird. Zusätzlich kann er entscheiden, ob die geeigneten Methoden und Werkzeuge verwendet werden. Außerdem kann der Qualitätssicherer belegbare Aussagen über die Qualität der Projektes und der erstellten Produkte machen und auch die Einhaltung von vorgegebenen Qualitätszielen überprüfen. Auch der Entwickler wird bei seiner Arbeit unterstützt,da er einen Überblick erhalten kann, wie sich die Fehler auf die jeweiligen Status verteilen. Dadurch kann er entscheiden, welche Arbeit er durchführen sollte, z.B. Fehlerkorrektur.

6.2 Ausblick

In diesem Abschnitt werden einige mögliche Erweiterungen bzw. Veränderungen vorgestellt, die für die Zukunft denkbar sind. Hierzu zählen auch Änderungen, die zu einer besseren Erfüllung der Anforderungen dienen.

Da bei einigen Sichten das Problem besteht, wie die benötigten Daten zu erfassen sind, wurden diese auch nicht implementiert. Daher ist zu überdenken, ob die Metriken nicht vereinfacht werden können und somit die Möglichkeit besteht, die Sichten in leicht abgewandelter Form zu implementieren und zu verwenden. Ein Beispiel für eine solche Metrik sind die Änderungen pro Phase (siehe 4.2, Metrik 14). Ein Problem besteht bereits darin, zu definieren, was man unter einer Änderung versteht, werden Änderungen erst berücksichtigt, wenn alle Phasen einmal komplett durchlaufen sind, oder auch bereits Umformulierungen, während des ersten Durchlaufs der Phasen, um die Zusammenhänge zu verdeutlichen. Außerdem ist es schwierig zu entscheiden, wie weit eine Änderung reicht, wird also beispielsweise jede Klasse, in der geändert wird, als eigene Änderung betrachtet oder zu einer zusammengefasst, wenn es sich auf eine Änderung - in den Schnittstellen beispielsweise - zurückführen lässt. Eine Vereinfachung der Metrik würde beispielsweise die Anzahl der Änderungen an der Anzahl der Versionen für die unterschiedlichen Klassen und erstellten Dokumente festmachen.

Außerdem sollte versucht werden, ob es nicht möglich ist, die beiden Sichten *Phases Line Plot* und *Phases Line Plot With Activities* zusammenzufassen, da die Gemeinsamkeiten doch sehr groß sind.

Momentan ist auch in der Sicht *Defect View* die Anzahl der darstellbaren Fehlerquellen durch die Implementierung auf vier beschränkt. In einer weiteren Änderungen sollte dies variabel gestaltet werden, so dass auch mehr Affiliation möglich sind, ohne dass die Implementierung jedes Mal neu angepasst werden muss.

Eine weniger wichtige, aber ähnliche Einschränkung, ist auch in der Sicht *Pie Chart* gegeben, da dort die Anzahl der darstellbaren Teilmengen auf 16 beschränkt ist. Auch dies ist durch die Implementierung festgelegt, wobei allerdings die Anzahl an möglichen Teilmengen vorübergehend als ausreichend akzeptiert wurde, da bei einer größeren Anzahl von Mengen auch die Aussagekraft der Sicht geringer wird, da nichts mehr zu erkennen ist. Eine mögliche Behebung der Einschränkung besteht darin, den Algorithmus, der die Farbe der Teilmenge berechnet, auszutauschen. Hierbei ist zu beachten, dass der Kontrast nicht zu stark abgeschwächt wird. Eine alternative Lösung wäre, dass die Farben, als zusätzliche Eingabe in eine Sicht eingehen. Dies wäre jedoch nicht nur für diese Sicht sinnvoll, sondern auch für die

anderen, da somit die jeweilige Sicht auch besser an die Wünsche des Betrachters angepasst werden kann. Diese Anforderung wurde nämlich noch kaum berücksichtigt bei den bisher realisierten Sichten.

Eigentlich sollte jede Sicht, ebenso wie Datentypen und Funktionen, eine Menge von Konsistenztests überprüfen, jedoch sind nur wenige überhaupt definiert und kurz erläutert worden. Bis zu dem jetzigen Zeitpunkt wird noch kein einziger durchgeführt. So wäre es sinnvoll, weitere notwendige Konsistenztests zu definieren und diese auch durchzuführen.

Abbildungsverzeichnis

1 Einordnung des SPCCs in eine Software-Entwicklungsumgebung (aus [11]). 2
2 Überblick über die einzelnen Komponenten des SPCCs (aus [11]). Die oberen Teile zeigen den Datenfluss und die Instanziierungskette zwischen den Elementen der Visualisierungskette. Der untere Teil zeigt die zugehörigen Komponenten mit ihrer formalen Spezifikation. 3
3 Beispiel einer möglichen Visualisierungskette für die Rollen des Projektmanagers und des Qualitätssicherers (aus [11]). 5
4 Hauptfenster Client nach dem Starten, Verbinden mit dem Server und Einloggen. 6
5 Auswahl der Visualisierungskette, die bearbeitet werden soll. 6
6 Menü zur Auswahl der Komponenten. 7
7 Fenster zum Bearbeiten der Funktion „Reported Effort Processing". 7
8 Beispiel einer möglichen Hierarchie von Zielen, Fragen und Maßen. . 10
9 Beispiel eines zweidimensionalen Liniendiagramms. Es wird hierbei eine mögliche Verteilung der verschiedenen Fehlerklassen (x-Achse) auf die unterschiedlichen Status (pro Status eine Kurve) dargestellt. 11
10 Beispiel des Sachverhalts von Abbildung 9 als dreidimensionales Liniendiagramm. Der zweite Parameter wäre hier der Status der Fehler. . 12
11 Beispiel eines zweidimensionalen Balkendiagramms. Es wird für jede Fehlerklasse die absolute Häufigkeit dargestellt. 13
12 Beispiel für ein Balkendiagramm mit zwei Parametern. Der erste Parameter wird entlang der x-Achse aufgetragen und der zweite Parameter wird über die unterschiedlichen Farben der Balken kodiert. Im Gegensatz zum vorangegangenen Diagramm, wird hier nicht nur die Anzahl der Fehler pro Fehlerklasse, sondern auch der Status der jeweiligen Fehler aufgetragen. 13
13 Beispiel eines möglichen dreidimensionalen Balkendiagramms. Es werden die gleichen Daten wie im vorangegangen Diagramm dargestellt, mit dem Unterschied, dass dies hier dreidimensional geschieht, indem der Status eines Fehlers entlang einer eigenen Achse aufgetragen werden. 14
14 Beispiel eines Kreisdiagramms; es wird die relative Häufigkeit der einzelnen Fehlerklassen dargestellt. 15
15 Beispiel einer Matrix-Darstellung (nach [4]). Hierbei sind entlang der y-Achse die unterschiedlichen Entwickler aufgetragen und entlang der x-Achse die Module. Die Breite des Balkens kodiert den Umfang des Moduls und die Farbe den Entwickler. 16
16 Vergrößerter Ausschnitt der vorangegangenen Abbildung einer Matrix. 17
17 Beispiel eines Netzwerks (nach [4]). Es wird dargestellt, welche Dateien von einer Änderungen betroffen sind, d.h. wenn Datei 1 geändert wurde, wie häufig war davon auch Datei 2 betroffen. Die Häufigkeit wird über die Farbe der Verbindung kodiert. 18
18 Beispiel einer sinnvollen Anwendung einer Tabelle. Es werden die Anforderungen zusammen mit der zugehörigen Priorität dargestellt und diejenigen markiert, die bereits erfüllt sind. 19

19 Beispiel eines Zeitplans. 19
20 Aufbau eines Boxplots. 20
21 Beispiel eines Boxplots, das die Anzahl gelöschter und hinzugefügter Kodezeilen zeigt. 21
22 Sichten-Meta-Modell als UML-Diagramm. 42
23 Klassifikation der Rollen eines Software-Entwicklungsprozesses. . . . 49
24 Darstellung des Transformationsprozesses; ausgehend von externen Daten über deren Verarbeitung zu internen Daten, die vom Generator mit Hilfe eines XSLT-Stylesheets in SVG-Grafiken umgewandelt werden. 63
25 Fenster, welches die zur Verfügung stehenden Funktionen zeigt. . . . 64
26 Fenster, welches die zur Verfügung stehenden Sichten zeigt. 65
27 Fenster, welches die zur Verfügung stehenden SVG-Grafiken zeigt. . 65
28 Die Sicht „Phases Line Plot” mit den zugehörigen Eingaben und Parametern. 68
29 Datenfluss der Daten einer möglichen Instanz der Sicht „Phases Line Plot”. 68
30 Mittels der Sicht „Phases Line Plot” erzeugte SVG-Grafik. 69
31 Die Sicht „Phases Line Plot With Activities” mit den zugehörigen Eingaben und Parametern. 70
32 Datenfluss einer möglichen Instanz der Sicht „Phases Line Plot With Activities”. 71
33 Anordnung der Phasen-Diagramme, welche durch die Sicht „Phases Line Plot With Activities” erzeugt werden. 72
34 Ausschnitt der durch die Sicht „Phases Line Plot With Activities” erzeugten SVG-Grafik, Phasenübersicht. 73
35 Ausschnitt der durch die Sicht „Phases Line Plot With Activities” erzeugten SVG-Grafik, der ersten Phase. 74
36 Die Sicht „Defect View” mit den zugehörigen Eingaben und Parametern. 75
37 Datenfluss einer möglichen Instanz der Sicht „Defect View”. 75
38 Mittels der Sicht „Defect View” erzeugte SVG-Grafik. 76
39 Die Sicht „Project Plan” mit den zugehörigen Eingaben und Parametern. 77
40 Datenfluss einer möglichen Instanz der Sicht „Project Plan”, wobei die beiden Pläne identisch sind. 77
41 Mittels der Sicht „Project Plan” erzeugte SVG-Grafik. 78
42 Die Sicht „Pie Chart” mit zugehörigen Eingaben und Parametern. . 79
43 Datenfluss einer möglichen Instanz der Sicht „Pie Chart”. 79
44 SVG-Grafik, die mit Hilfe der Sicht „Pie Chart” erzeugt wurde, zeigt die Fehlerverteilung auf die Fehlerstatus. 80

Tabellenverzeichnis

1 Übersicht über die unterschiedlichen Quellen. 36
2 Bewertung der Sichten „Software Measurement Guidebook”. 43
3 Bewertung der Sichten „Collecting and Analyzing Web-Based Project Metrics”. 44
4 Bewertung der Sichten „A Practical View of Software Measurement and Implementation Experiences Within Motorola”. 45
5 Bewertung der Sichten „Visualizing Software Changes”. 46
6 Bewertung der Sichten „Software Management Environment (SME) - Components and Algorithms”. 47
7 Klassifikation der Sicht: Aufwandsverteilung auf Aktivitäten. 55
8 Klassifikation der Sicht: zeitlicher Projektverlauf. 56
9 Klassifikation der Sicht: Fehlerverteilung auf Phasen und Affiliation und zugehörigem Aufwand. 56
10 Klassifikation der Sicht: Fehlerverteilung auf Fehlerklassen. 57
11 Klassifikation der Sicht: Fehlerverteilung auf Status. 57
12 Klassifikation der Sicht: Fehlerverteilung nach Status und Fehlerklasse. 57
13 Klassifikation der Sicht: Auflistung der Anforderungen. 58
14 Klassifikation der Sicht: Ort der Änderungen. 58
15 Klassifikation der Sicht: Kategorie der Änderungen. 59
16 Klassifikation der Sicht: Anzahl der Änderungen pro Phase. 59
17 Klassifikation der Sicht: Aufwand für die Änderungen pro Phase. . . 60
18 Klassifikation der Sicht: Änderungen und damit verbundener Aufwand pro Phase. 60
19 Klassifikation der Sicht: Prozentuales Verhältnis zwischen Kodezeilen und Dokumentationszeilen. 61
20 Klassifikation der Sicht: Aufwand für Tests pro Klasse. 61
21 Klassifikation der Sicht: Kostenvergleich pro Phase. 62
22 Klassifikation der Sicht: Verteilung der Gesamtkosten. 62
23 Bewertung der Sichten. 81

Literatur

[1] „A Practical View of Software Measurement and Implementation Experiences Within Motorola"; Michael K. Daskalantonakis; IEEE Transaction On Software Engineering, VOL. 18, No. 11; November 1992.

[2] „Software measurement guidebook"; Software Engineering Laboratory Series; Mitchell J. Bassman, Frank McGarry, Rose Pajerski; Juni 1995.

[3] „Software Management Environment (SME) - Components and Algorithms"; Software Engineering Laboratory Series; Robert Hendrick, David Kistler, Jon Valett; Februar 1994.

[4] „Visualizing Software Changes"; Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, Paul Schuster; IEEE Transaction On Software Engineering, VOL. 28, No. 4; April 2002.

[5] „Collecting and Analyzing Web-Based Project Metrics"; Rob Pooley, Dave Senior, Duncan Christie; IEEE Software; Januar/ Februar 2002.

[6] „Business Information Visualization"; David P. Tegarden; Communication of AIS Volume 1, Article 4; Januar 1999.

[7] „13 Ways to Say Nothing with Scientific Visualization"; A. Globus, E. Raible; Report RNR-92-006; 25. Februar 1992.

[8] „The Goal Question Metric Approach"; Victor R. Basili, Gianluigi Caldiera, H. Dieter Rombach; Institute for Advanced Computer Studies, Department of Computer Science; 1994.

[9] Skripte zu den Software Engineering 1 und 2 Vorlesungen; AG Software Engineering; Universität Kaiserslautern, 2002.

[10] „SPCC Anforderungsdokument - Anforderungen an einen Software Project Control Center"; Jens Heidrich; AG Software Engineering; Universität Kaiserslautern 2002.

[11] „A Logical Architecture for Process-accompanying Data-Visualisation and Interpretation"; Jens Heidrich; Technischer Bericht; Sonderforschungsbereich 501; AG Software Engineering; Universität Kaiserslautern; 2003 (noch nicht veröffentlicht).

[12] „Software Project Control Centers: Concepts and Approaches"; Jürgen Münch, Jens Heidrich; Sonderforschungsbereich 501, Bericht Nr. 02/2002, AG Software Engineering, Universität Kaiserslautern; Februar 2002.

[13] „Entwicklung eines Werkzeuges zur prozessbegleitenden Visualisierung von Messdaten"; Pascal Hansen; Diplomarbeit; Fachbereich Informatik; Universität Kaiserslautern; September 2002.

[14] „Software Metrics: Establishing a company-wide program"; M. Robert B. Grady, Deborah L. Caswell; Prentice-Hall; ISBN 0-13-821844-7; 1987.

[15] „Direction in Software Process Research"; H. Dieter Rombach, Martin Verlage; Fachbereich Informatik, Universität Kaiserslautern; 1995.

[16] „A Formal Specification of a Visualization Catena for Software Project Control"; Jens Heidrich; Technischer Bericht; Sonderforschungsbereich 501; AG Software Engineering; Universität Kaiserslautern; 2003 (noch nicht veröffentlicht).

[17] SVG, Scalable Vector Grafics Specification; `http://www.w3.org/TR/SVG/`; besucht im Mai 2003.

[18] XSL, extensible Stylesheet Language Specification; `http://www.w3.org/Style/XSL/` besucht im Mai 2003.

[19] Adobe SVG Viewer; `http://www.adobe.com/svg/main.html` besucht im Mai 2003.

[20] „XML - kurz und gut"; Robert Eckstein, Michel Casabianca, Übersetzung Nik Klever; O'Reilly & Associates; ISBN 3-89721-235-8; 2002.

[21] Apache Batik; `http://xml.apache.org/batik/` besucht im Mai 2003.

[22] „An Integrated Approach to Software Engineering"; Pankaj Jalote; Springer Verlag, zweite Auflage; ISBN 0-387-94899-6; 1997.

[23] „Software Engineering - Methods and Management"; Anneliese von Mayrhauser; Department of Computer Science, Illinois Institute of Technologie, Chicago Illinois; ISBN 0-12-727320-4; 1990.

[24] „Goal/Question/Metric"; `http://www.iese.fraunhofer.de/PERFECT/GQM/GQM.html` besucht im Mai 2003.

[25] „Goal Question Metric - A GQM Overview"; `http://dec.bournemouth.ac.uk/ESERG/mshepperd/SEMGQM.html` besucht im Mai 2003.

[26] „Developing a Successful Metrics Programm"; L. Rosenberg, L. Hyatt; `http://satc.qsfc.nasa.gov/support/STC_APR96/metrics/stc_met.html` besucht im Mai 2003.

[27] „SVG - Learning By Coding"; `http://www.drahtenverdrahten.de/svglbc/` besucht im Mai 2003.

[28] „Controlling Software Project Risks - an Empirical Study of Methods used by Experienced Project Managers"; Tom Addison, Seema Vallabh; Proceeding of SAICSIT Pages 128-140; 2002.

[29] „Power: A Tool for Quantitative Evaluation of Software Project Effectiveness"; Michael W. Evans & Louis M. Picinich; Integrated Computer Engineering; 1984.

[30] „Software Project Management: The Manager's View"; Jaak Jurison; Graduate School of Business, Fordham University; Communication of the Association for Information Systems; September 1999.

[31] „Software Metrics: Ten Traps to avoid"; Karl E. Wiegers; Software Developement, 4, No. 5; 1996.

[32] „Using Metrics in Managing and Improving Software Development Processes"; Helle Damborg Frederiksen; Department of Computer Science; Aalborg, University 2001.

[33] „Metrics News"; Journal of the GI Interest Group on Software Metrics; Volume 2, Number 2; Dezember 1997.

[34] „Entwicklung von Softwaresystemen 2: Design Patterns"; Colin Atkinson; Auszug eines Vorlesungskapitels; Sommersemester 2000.

A Sichten

Die folgenden Abbildungen wurden den jeweiligen Artikeln entnommen und sind hier lediglich zu Vergleichszwecken abgebildet. Die Grafiken enthalten jeweils die Orginal-Abbildungsnummer, sowie die in Abschnitt 2.2 der Diplomarbeit vergebene Sichtennummer.

A.1 „Software measurement guidebook" (aus [2])

Sicht 1

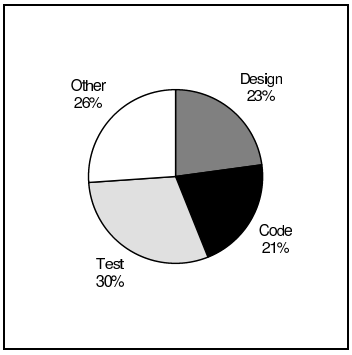


Figure 2-2. Effort Distribution by Activity

Sicht 2

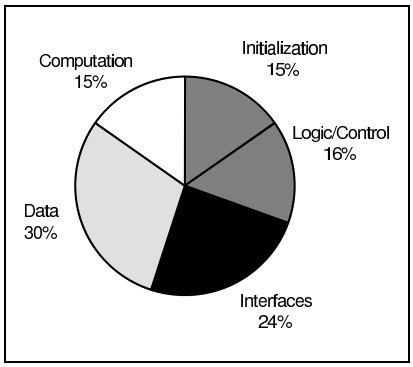


Figure 2-3. Error Class Distribution

Sicht 3

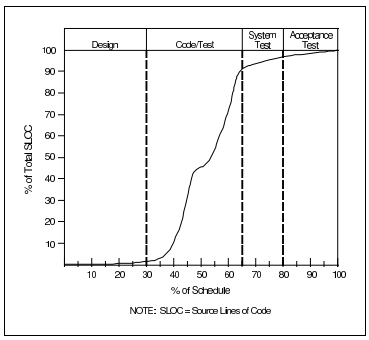


Figure 2-4. Growth Rate of Source Code

Sicht 4

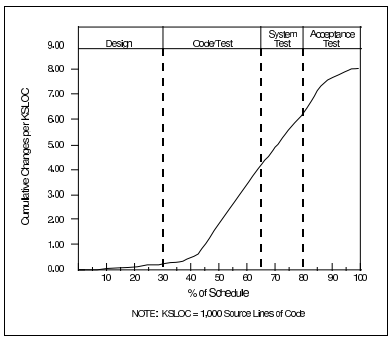


Figure 2-5. Change Rate of Source Code

Sicht 5

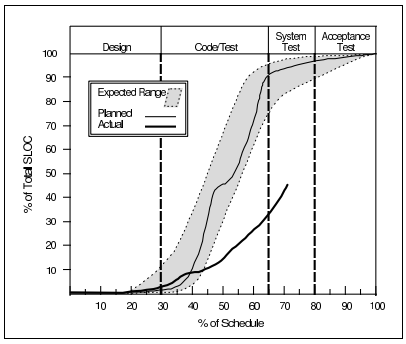


Figure 2-7. Tracking Growth Rate

Sicht 6

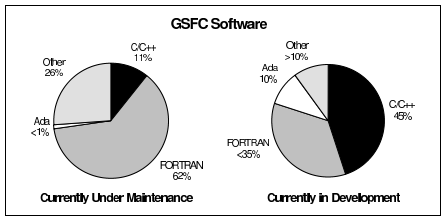


Figure 6-1. Language Usage Trend

Sicht 7

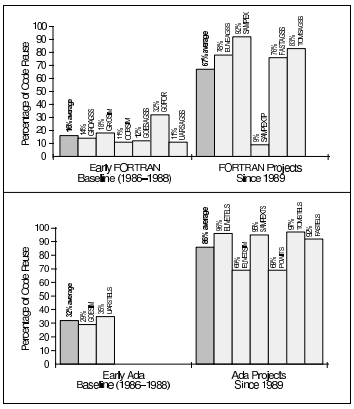


Figure 6-2. Code Reuse Trend

Sicht 8 (Teil 1)

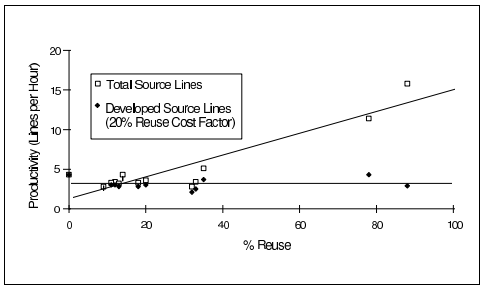


Figure 6-3. Derivation of 20 Percent Reuse Cost Factor for FORTRAN

Sicht 8 (Teil 2)

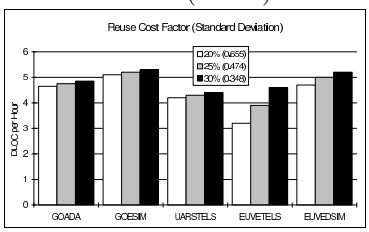


Figure 6-4. Derivation of 30 Percent Reuse Cost Factor for Ada

Sicht 9

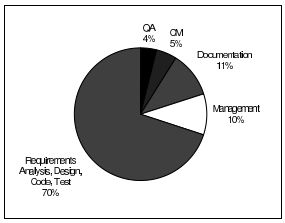


Figure 6-7. Typical Allocation of Software Project Resources

Sicht 10

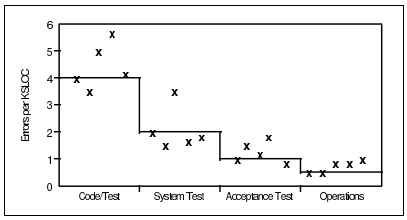


Figure 6-8. Error Detection Rate by Phase

Sicht 11

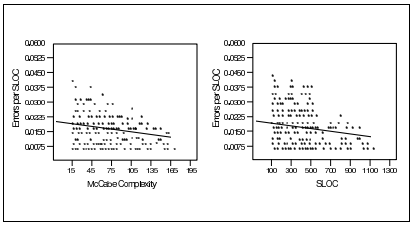


Figure 6-10. Cyclomatic Complexity and SLOC as Indicators of Errors (Preliminary Analysis)

Sicht 12

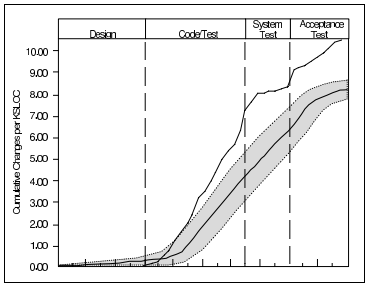


Figure 6-14. Change Rate Deviation

Sicht 13

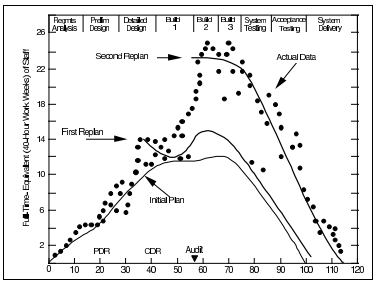


Figure 6-15. Staff Effort Deviation

Sicht 14

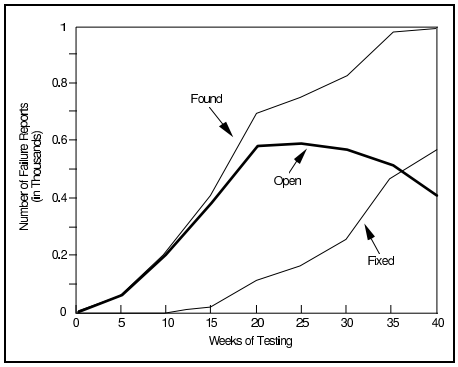


Figure 6-16. Tracking Discrepancies

Sicht 15

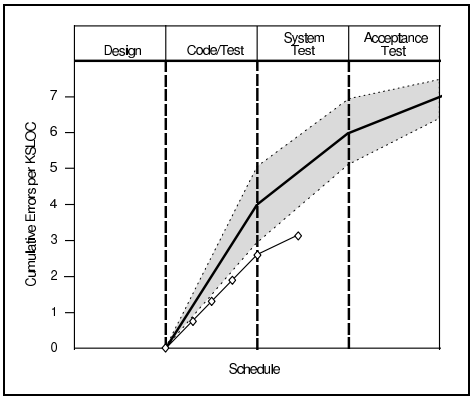


Figure 6-17. Projecting Software Quality

A.2 „Collecting and Analyzing Web-Based Project Metrics" (aus [5])

Sicht 16

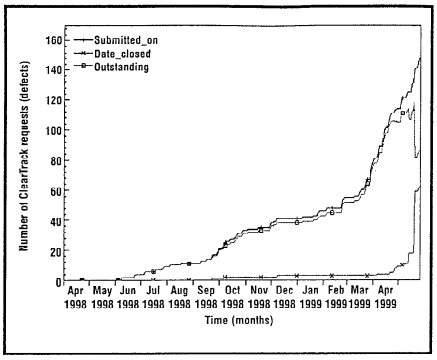


Figure 4. Plot of CTRs over time. Note the sharp climb in closures after the project manager corrected an implementation problem.

Sicht 17

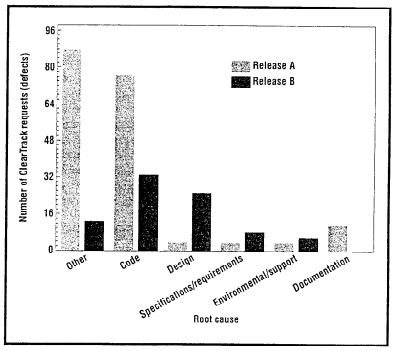


Figure 5. A root cause example for two releases. The amorphous category of "other" for the older release is not helpful because it is so heavily populated; the distribution of root causes is more balanced in the newer release.

Sicht 18

Table 1

CTR time to fix according to date_closed and tested_on metrics

	Critical		High		Medium		Low		Project	
	date_closed	tested_on	date_closed	tested_on	date_closed	tested_on	date_closed	tested_on	date_closed	tested_on
Max	64	8	196	59	398	136	472	472	472	472
Min	39	3	11	1	0	0	0	0	0	0
Average	54.33	6.25	88.60	23.18	116.62	27.06	69.37	63.04	85.50	40.63
Count	3	4	10	11	16	17	27	25	56	57

Table 2

CTR age according to date_closed and tested_on metrics

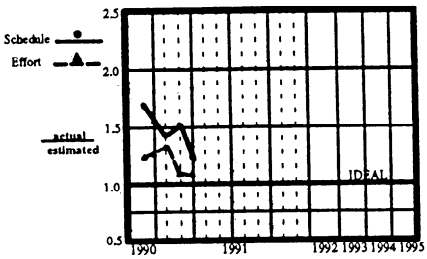
	Critical		High		Medium		Low		Project	
	date_closed	tested_on	date_closed	tested_on	date_closed	tested_on	date_closed	tested_on	date_closed	tested_on
Max	176	—	258	—	342	563	484	560	484	563
Min	176	—	258	—	85	85	126	127	85	85
Average	176	—	258	—	224.50	330	257.38	510.70	242.21	469
Count	1	0	1	0	4	3	8	10	56	13

A.3 „A Practical View of Software Measurement and Implementation Experiences Within Motorola" (aus [1])

Sicht 19

CHART 9

ESTIMATION ACCURACY

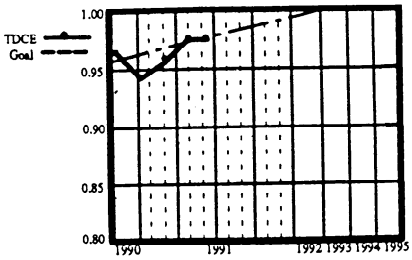


Schedule and Effort Estimation Accuracy (SEA and EEA) for the initial estimates of a Motorola Division's software projects.

Sicht 20

CHART 7

TOTAL DEFECT CONTAINMENT EFFECTIVENESS

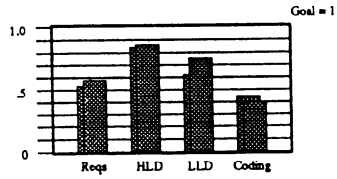


Total Defect Containment Effectiveness (TDCE) for the projects of a Motorola division.

Sicht 21

CHART 8

PHASE CONTAINMENT EFFECTIVENESS



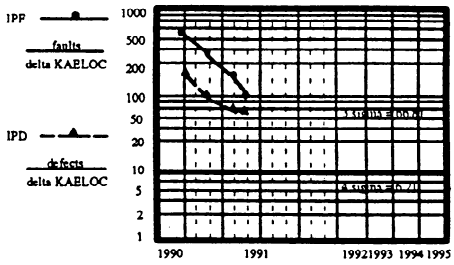
Phase Containment Effectiveness (PCE) for the constructive phases of projects released in the last twelve months within a Motorola Division.

Right: projects released in the last six months.
Left: projects released the prior six months.

Sicht 22

CHART 1

SOFTWARE DEVELOPMENT PROCESS QUALITY (IN SIGMA)

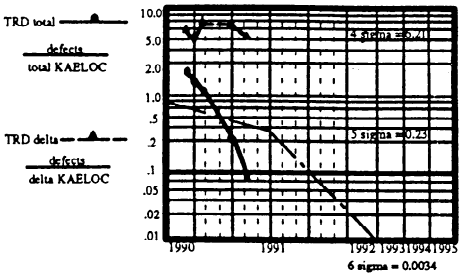


In–Process Faults (IPF) metric and In–Process Defects (IPD) metric for the software released by a Motorola Division.

Sicht 23

CHART 2

RELEASED SOFTWARE QUALITY (IN SIGMA)

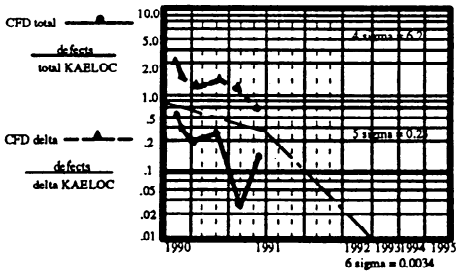


Total Released Defects (TRD) in the software of a Motorola Division.

Sicht 24

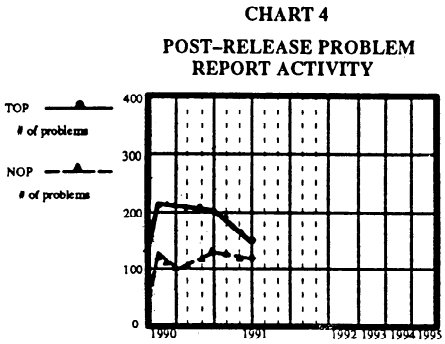
CHART 3

CUSTOMER–FOUND DEFECTS (IN SIGMA)



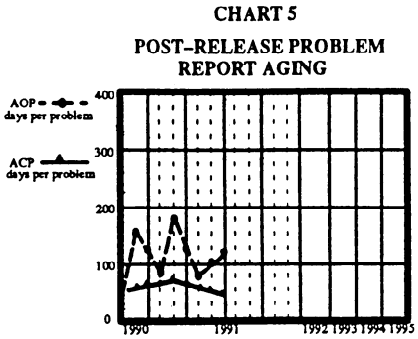
Customer–Found Defects (CFD) in the software of a Motorola division.

Sicht 25



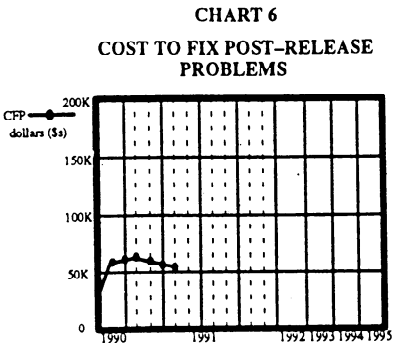
Mean New Open Problems (NOP) and Total Open Problems (TOP) for the software of a Motorola Division.

Sicht 26



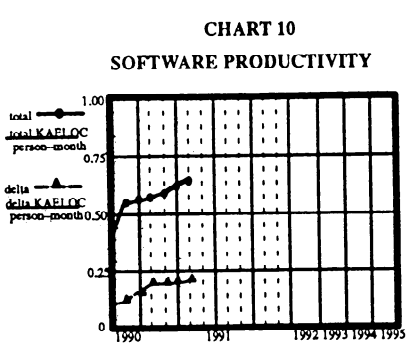
Mean Age of Open Problems (AOP) and mean Age of Closed Problems (ACP) for the software of a Motorola Division.

Sicht 27



Cost to Fix post-release Problems (CFP) within.a Motorola Division.

Sicht 28



Software Productivity (SP) total and delta for the projects of a Motorola Division.

Sicht 29

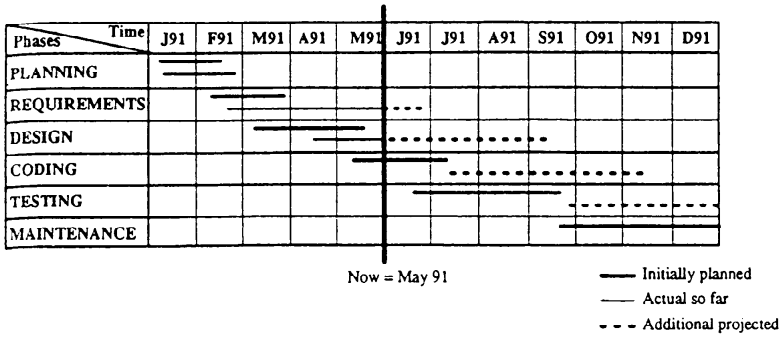


Fig. 7. A chart indicating the progress of planned activities for a project.

Sicht 30

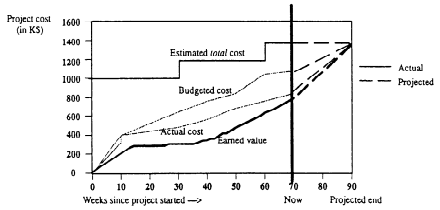


Fig. 8. A chart for tracking the earned value of the project (relative to the budgeted and actual cost).

Sicht 31

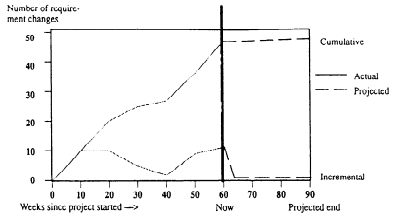


Fig. 9. Number of changes to the requirements document over time for a project.

Sicht 32

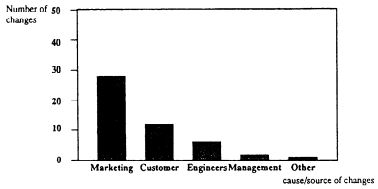


Fig. 10. Pareto chart for identifying major causes/sources of requirement changes in order to take corrective action (in-process).

Sicht 33

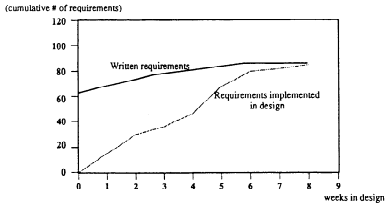


Fig. 11. Design creation progress metric for a software project.

Sicht 34

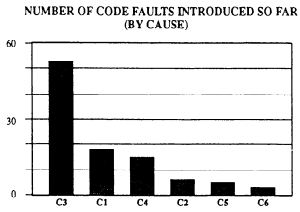


Fig. 12. Pareto chart for identifying what type of code faults is occurring most frequently, so that coders of additional modules can avoid their introduction in such modules.

Sicht 35

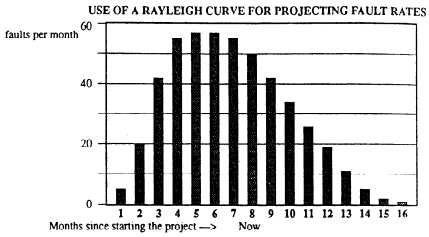


Fig. 13. Fault finding rate over the life-cycle of a software project.

Sicht 36

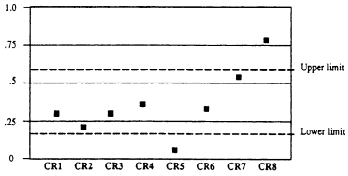


Fig. 14. Error density of source code reviewed within the project (or similar projects in the Division).

Sicht 37

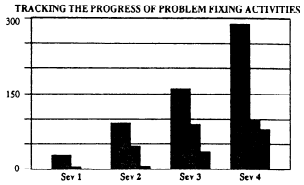


Fig. 15. Total open problems for the past three weeks of system testing for a software project.

Sicht 38

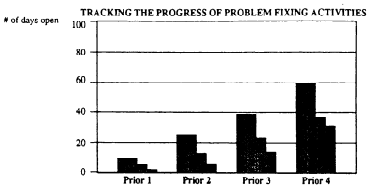


Fig. 16. Age of Open Problems for the past three months of system testing for a software project.

A.4 „Visualizing Software Changes” (aus [4])

Sicht 39

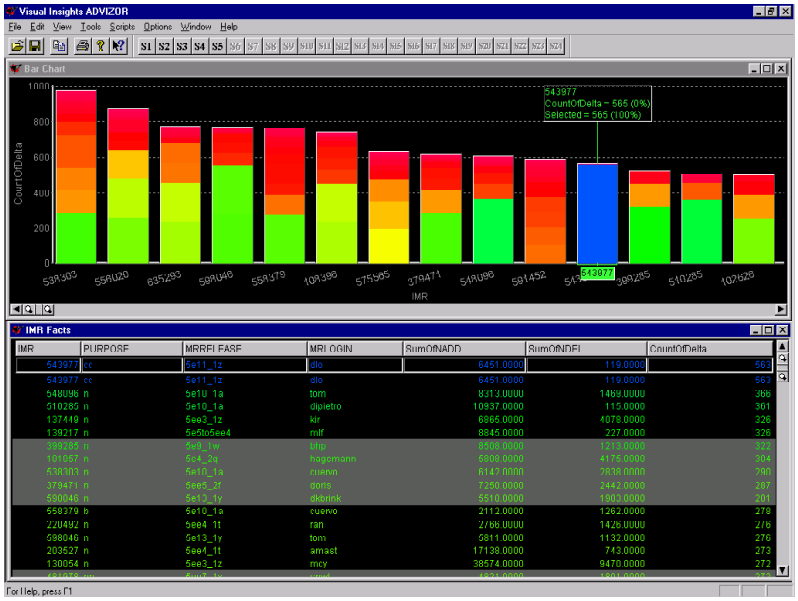


Figure 1

Sicht 40

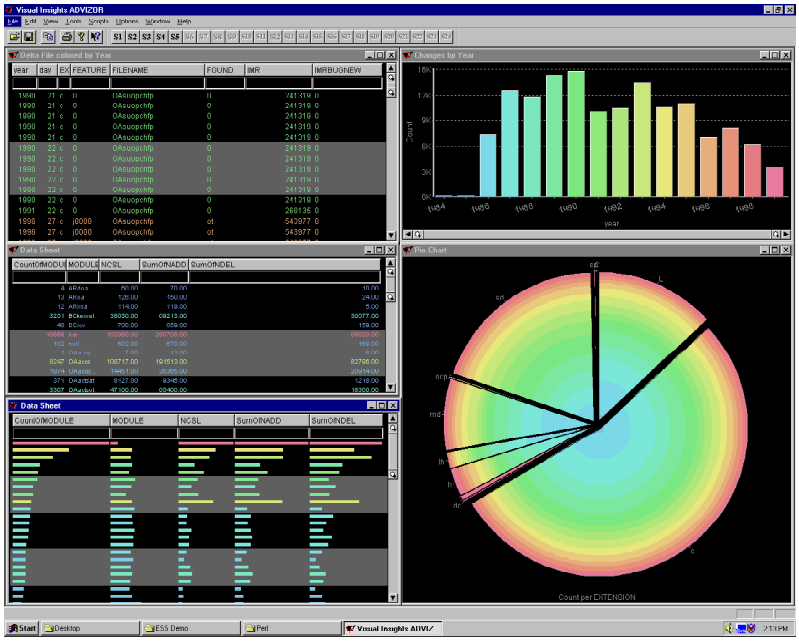


Figure 2

Sicht 41

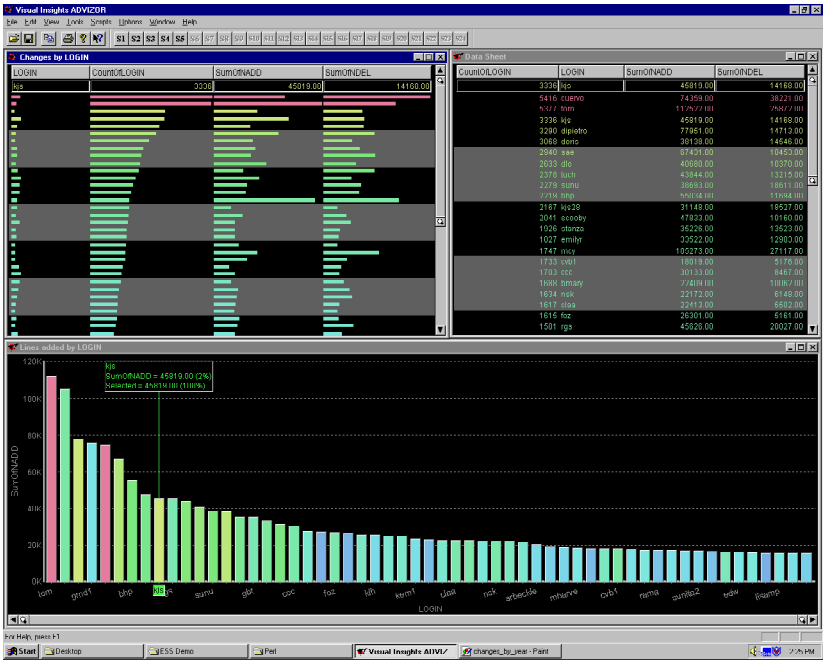


Figure 3

Sicht 42

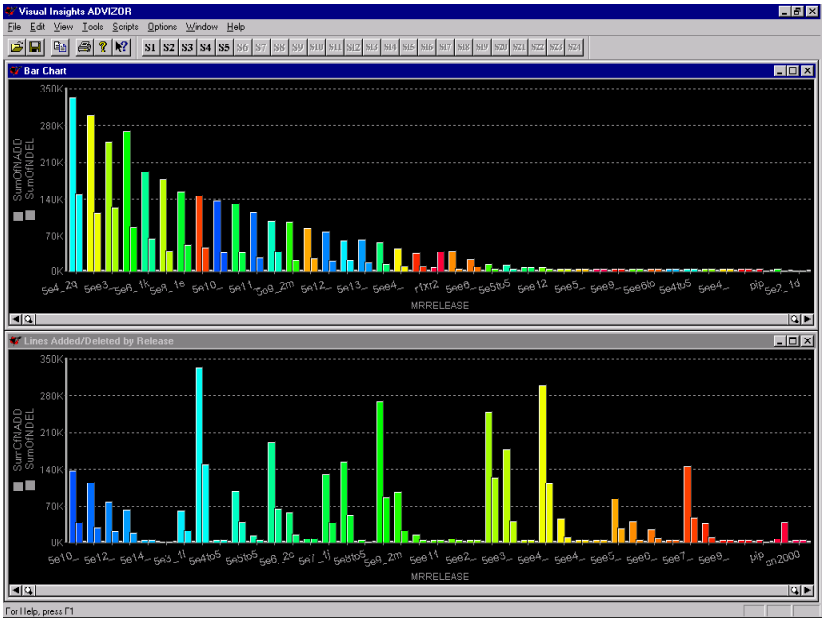


Figure 4

Sicht 43 (Teil 1)

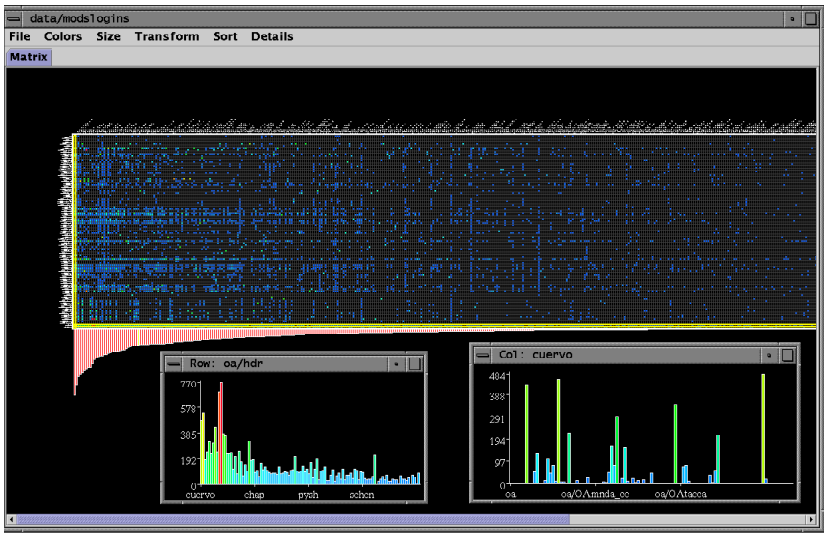


Figure 5

Sicht 43 (Teil 2)

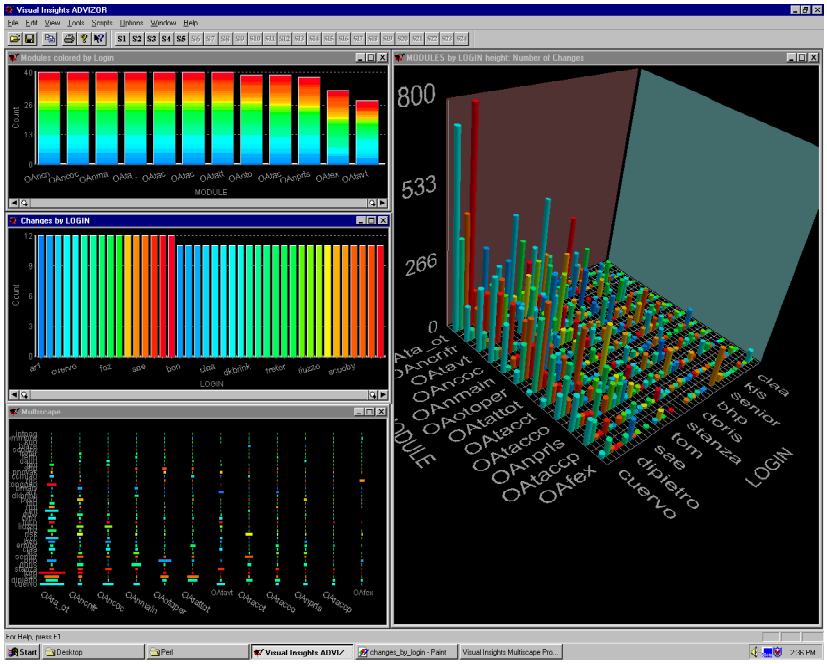


Figure 6

Sicht 44

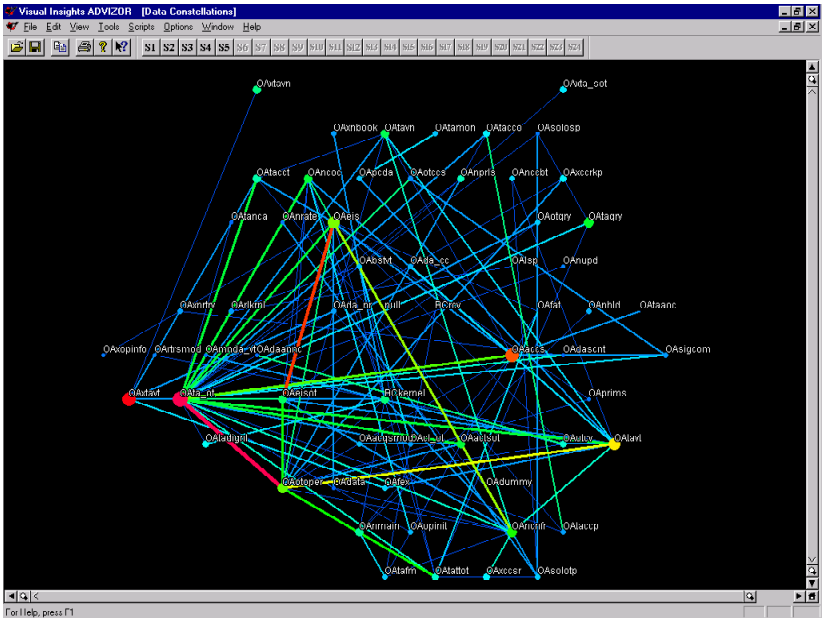


Figure 7

A.5 „Software Management Environment (SME) - Components and Algorithms" (aus [3])

Sicht 45

2.1.7.1.1 Effort Data

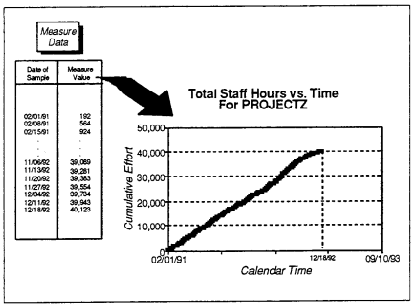


Figure 2-8. Effort Data for an Ongoing Project

Sicht 46

2.1.7.1.2 Lines of Code Data

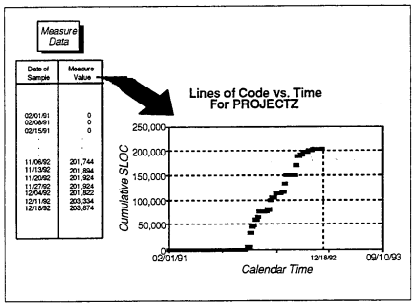


Figure 2-9. Lines of Code Data for an Ongoing Project

Sicht 47

2.1.7.1.3 Module Count Data

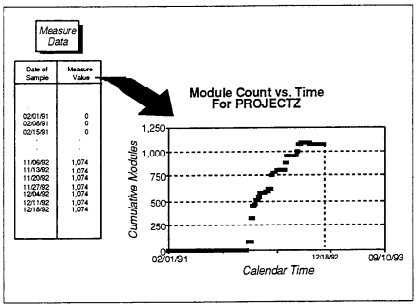


Figure 2-10. Module Count Data for an Ongoing Project

Sicht 48

2.1.7.1.4 Computer Hours Data

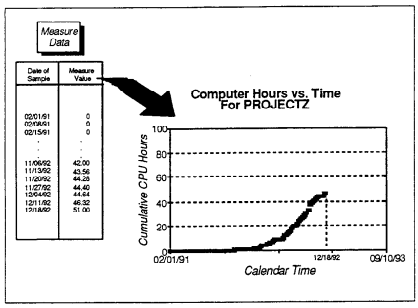


Figure 2-11. Computer Hours Data for an Ongoing Project

Sicht 49

2.1.7.1.5 Computer Runs Data

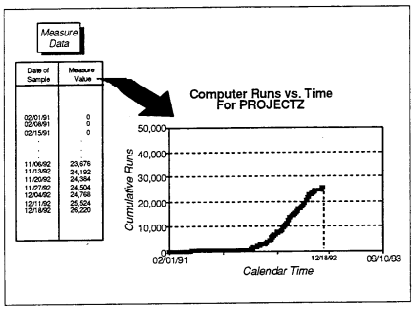


Figure 2-12. Computer Runs Data for an Ongoing Project

Sicht 50

2.1.7.1.6 Changed Modules Data

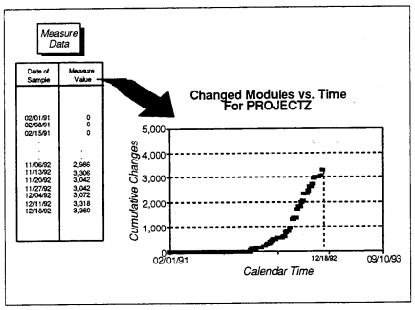


Figure 2-13. Changed Modules Data for an Ongoing Project

Sicht 51

2.1.7.1.7 Reported Changes Data

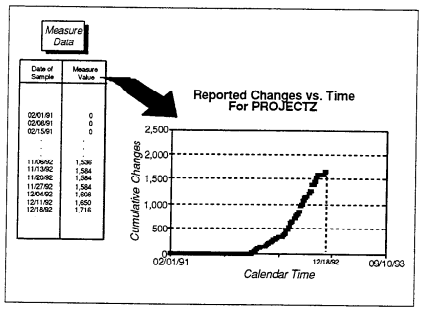


Figure 2-14. Reported Changes Data for an Ongoing Project

Sicht 52

2.1.7.1.8 Reported Errors Data

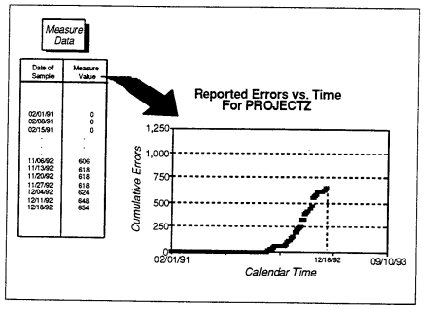


Figure 2-15. Reported Errors Data for an Ongoing Project

Sicht 53

2.1.8.1.1 Effort to Isolate Change Data

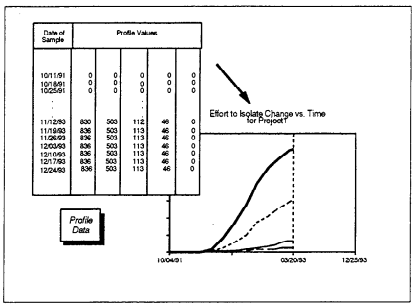


Figure 2-17. Effort to Isolate Change Data for an Ongoing Project

Sicht 54

2.1.8.1.2 Effort to Implement Change Data

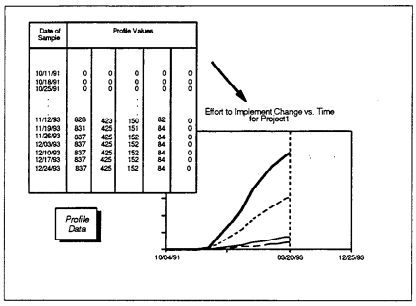


Figure 2-18. Effort to Implement Change Data for an Ongoing Project

Sicht 55

2.1.8.1.3 Effort to Isolate Error Data

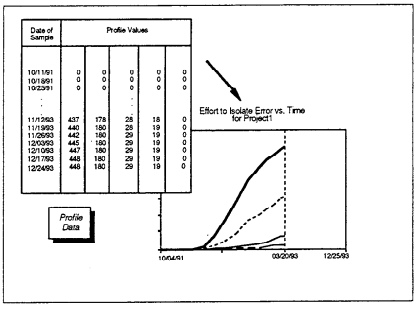


Figure 2-19. Effort to Isolate Error Data for an Ongoing Project

Sicht 56

2.1.8.1.4 Effort to Correct Error Data

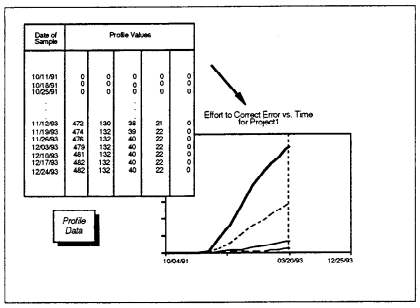


Figure 2-20. Effort to Correct Error Data for an Ongoing Project

B Implementierung der Sichten

Hier finden sich die XSLT Dateien, welche zur Erstellung einer Sicht benötigt werden.

B.1 Aufwandsverteilung auf Phasen (*Phases Line Plot*)

```
<!-- ================================================================   -->
<!-- The stylesheet creates a diagramm that shows the effort of all      -->
<!-- phases. The values that are too low or too high are marked.         -->
<!-- ================================================================   -->
<xsl:stylesheet version="1.0"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="svg"
       doctype-public="-//W3c//DTD SVG 1.0//EN"
       doctype-system="http://w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"/>

  <!-- ============================================================   -->
  <!-- The "master" template, sets the viewbox and calls the other     -->
  <!-- templates that do the work and defines a set of symbols.        -->
  <!-- ============================================================   -->
  <xsl:template match="globaldata">

    <svg xml:space="preserve"
         xmlns="http://www.w3.org/2000/svg"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         x="0cm" y="0cm" width="15cm" height="15cm"
         viewBox="0 0 800 800"
         id="master">
      <desc> In the 'main' template only the symbols and the script are
          defined . The 'real' work is done by other templates.
      </desc>

      <!-- ========================================================   -->
      <!--  Define a symbol for the coordinate system.                 -->
      <!-- ========================================================   -->
      <defs>
        <desc> The coordinate system is defined here as a symbol, in
            order to use it later and to simplify the stylesheet.
        </desc>
  <symbol id="coordinateSystem">
        <!-- X-axis -->
    <line x1="60" y1="550" x2="560" y2="550"
          style="stroke-width:2 ; stroke: black;"/>
        <!-- Y-axis (greater zero)-->
    <line x1="60" y1="550" x2="60" y2="50"
          style="stroke-width:2 ; stroke: black;"/>
        <!-- the scaling of the y-axis without values -->
    <line x1="55" y1="50" x2="65" y2="50"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="100" x2="65" y2="100"
          style="stroke-width:2 ; stroke: black;"/>
    <line x1="55" y1="150" x2="65" y2="150"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="200" x2="65" y2="200"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="250" x2="65" y2="250"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="300" x2="65" y2="300"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="350" x2="65" y2="350"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="400" x2="65" y2="400"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="450" x2="65" y2="450"
          style="stroke-width:2; stroke:black;"/>
    <line x1="55" y1="500" x2="65" y2="500"
          style="stroke-width:2; stroke:black;"/>
  </symbol>
</defs>

<!-- ========================================================   -->
<!--  Define a symbol for the title.                              -->
<!-- ========================================================   -->
<defs>
  <desc> The title of the diagramm is defined as a symbol.
  </desc>
  <symbol id="title">
    <text x="320" y="50"
          style="text-anchor: middle; font-size:14pt;">
      <xsl:value-of select="//parameter[@id='Title']/@value"/>
```

```
    </text>
  </symbol>
</defs>

<!-- ========================================================   -->
<!--  Define a symbol for labeling the y-axis.                  -->
<!-- ========================================================   -->
<defs>
  <desc> The unit of the y-axis is defined as a symbol.
  </desc>
  <symbol id="unit y-axis">
    <text x="60" y="80"
          style="text-anchor: middle; font-size:12pt;">
      <xsl:value-of select="//parameter[@id='y-Axis Title']/@value"/>
    </text>
  </symbol>
</defs>

<!-- ========================================================   -->
<!--  Define a symbol for the legend.                           -->
<!-- ========================================================   -->
<defs>
  <desc> The legend is defined as symbol, because the legend
      is the same for all diagramms. The legend describes
      what the different curves mean.
  </desc>
  <symbol id="legend">
    <line x1="562" y1="115" x2="600" y2="115"
          style="stroke-width:3; stroke:green;"/>
    <text x="607" y="120"
          style="font-size:12pt;">Lower bound</text>

    <line x1="562" y1="140" x2="600" y2="140"
          style="stroke-width:2; stroke:red;"/>
    <circle cx="581" cy="140" r="4"
            style="stroke-width:2; stroke:red; fill:red;"/>
    <text x="607" y="145"
          style="font-size:12pt;">Current curve values</text>

    <line x1="562" y1="165" x2="600" y2="165"
          style="stroke-width:3; stroke:blue;"/>
    <text x="607" y="170"
          style="font-size:12pt;">Upper bound</text>

    <line x1="562" y1="190" x2="600" y2="190"
          style="stroke-width:2 ; stroke: red;"/>
    <circle cx="581" cy="190" r="4"
            style="stroke-width:2; stroke:red; fill:red;"/>
    <circle cx="581" cy="190" r="8"
            style="stroke-width:2; stroke:maroon; fill:none;"/>
    <text x="607" y="195"
          style="font-size:12pt;">Out of bound values</text>
  </symbol>
</defs>

  <xsl:call-template name="Phases overview"/>
</svg>
</xsl:template>

<!-- ============================================================   -->
<!-- The template creates the diagramm that shows the overview      -->
<!-- over all phases.                                               -->
<!-- ============================================================   -->
<xsl:template name="Phases overview">
  <desc> Prints a diagramm that contains the view of all phases.
  </desc>
  <use xlink:href="#coordinateSystem"
       x="0" y="50" width="20cm" height="15cm"/>
  <use xlink:href="#legend"
       x="0" y="0" width="20cm" height="15cm"/>
  <use xlink:href="#unit y-axis"
       x="0" y="0" width="20cm" height="15cm"/>
  <use xlink:href="#title"
       x="0" y="0" width="20cm" height="15cm"/>

  <!-- Print the unit of the x-axis.                          -->
  <text x="612" y="607"
        style="text-anchor: middle; font-size:12pt;">
    <xsl:value-of select="//parameter[@id='x-Axis Title']/@value"/>
  </text>

  <!-- Label the x-axis.                                      -->
  <xsl:call-template name="label x axis with phases"/>

  <!-- Search the maximum value and therefore the values are ordered in
       ascending order. The matching template although call all templates
       that need the scaling factor, e.g. scaling the y-axis -->
    <xsl:apply-templates select="//data/entry/vl">
      <xsl:sort data-type="number" order="descending"/>
    </xsl:apply-templates>
```

```
</xsl:template>

<!-- ============================================================   -->
<!-- The template searches the highest value that occures therefor  -->
<!-- it is called with the sorted values. Then the template calcu-  -->
<!-- lates the maximum to be 10 percent greater. With that value    -->
<!-- the other templates are called.                                -->
<!-- ============================================================   -->
<xsl:template match="//data/entry/vl">
  <xsl:if test="position() = 1">
    <xsl:variable name="max"
                  select="."/>
    <xsl:variable name="max"
                  select="($max) * 1.1"/>
    <xsl:variable name="max"
                  select="ceiling( ($max) ) +1"/>

    <xsl:variable name="dist"
                  select=" ($max) div 10"/>

    <xsl:call-template name="label y-axis">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
      <xsl:with-param name="dist">
        <xsl:value-of select="($dist)"/>
      </xsl:with-param>
      <xsl:with-param name="deltadist">
        <xsl:value-of select="($dist)"/>
      </xsl:with-param>
    </xsl:call-template>

    <!-- Then call the template for printing the upper border of  -->
    <!-- the current series (phase level).                        -->
    <xsl:call-template name="UpperBorder phases">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
      <xsl:with-param name="count">
        <xsl:value-of select="1"/>
      </xsl:with-param>
      <xsl:with-param name="oldX">
        <xsl:value-of select="0"/>
      </xsl:with-param>
      <xsl:with-param name="oldY">
        <xsl:value-of select="0"/>
      </xsl:with-param>
    </xsl:call-template>

    <!-- Then call the template for marking the values of the   -->
    <!-- current series that are too high (phase level).        -->
    <xsl:call-template name="TooHighValues phases">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
    </xsl:call-template>

    <!-- Then call the template for printing the current series -->
    <!-- (phase level).                                          -->
    <xsl:call-template name="Controlled series phases">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
      <xsl:with-param name="count">
        <xsl:value-of select="1"/>
      </xsl:with-param>
      <xsl:with-param name="oldX">
        <xsl:value-of select="0"/>
      </xsl:with-param>
      <xsl:with-param name="oldY">
        <xsl:value-of select="0"/>
      </xsl:with-param>
    </xsl:call-template>

    <!-- Then call the template for marking the values of the   -->
    <!-- current series that are too low (phase level).         -->
    <xsl:call-template name="TooLowValues phases">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
    </xsl:call-template>

    <!-- Now call the template for printing the lower border of   -->
    <!-- the current series (phase level).                        -->
    <xsl:call-template name="LowerBorder phases">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
      <xsl:with-param name="count">
        <xsl:value-of select="1"/>
```

```
      </xsl:with-param>
      <xsl:with-param name="oldX">
        <xsl:value-of select="0"/>
      </xsl:with-param>
      <xsl:with-param name="oldY">
        <xsl:value-of select="0"/>
      </xsl:with-param>
    </xsl:call-template>

  </xsl:if>
</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the y-axis with the corresponding values.   -->
<!-- It needs the scale value (the maximum value), the distance      -->
<!-- between x-axis and the last value and the distance between two  -->
<!-- values of the scale bar.                                        -->
<!-- The template calls itself recursive until the maximum value is  -->
<!-- reached (scale = dist).                                         -->
<!-- ============================================================   -->
<xsl:template name="label y-axis">
  <xsl:param name="scale"/>
  <xsl:param name="dist"/>
  <xsl:param name="deltadist"/>

  <xsl:variable name="dy"
                select="550 - ($dist) * 500 div ($scale)"/>
  <xsl:variable name="dyt"
                select="($dy) + 56"/>
  <text x="50" y="{$dyt}"
        style="font-size:12pt;text-anchor:end">
    <xsl:value-of select="($dist)"/>
  </text>

  <!-- Preparation for recursive call -->
  <xsl:variable name ="newdist"
                select="($dist) + ($deltadist)"/>
  <xsl:variable name ="newdist"
                select="floor( ($newdist) * 100)"/>
  <xsl:variable name ="newdist"
                select=" ($newdist) div 100"/>

  <!-- Test if recursiv call is necessary -->
  <xsl:if test="not( ($scale) &lt; ($newdist) )">
    <xsl:call-template name="label y-axis">
      <xsl:with-param name="scale">
        <xsl:value-of select="($scale)"/>
      </xsl:with-param>
      <xsl:with-param name="dist">
        <xsl:value-of select="($newdist)"/>
      </xsl:with-param>
      <xsl:with-param name="deltadist">
        <xsl:value-of select="($deltadist)"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:if>

</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the x-axis with the names of the different  -->
<!-- phases.                                                         -->
<!-- ============================================================   -->
<xsl:template name="label x axis with phases">
  <xsl:variable name="size"
            select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <xsl:for-each select="//*[@id='Phases and Activities']/data/entry">
    <xsl:variable name="x"
                  select="phaseNumber * 500 div ($size) + 60"/>
    <xsl:variable name="tx"
                  select="($x) - 500 div ($size) div 2 - 5"/>
   <line x1="{$x}" y1="605" x2="{$x}" y2="595"
         style="stroke-width:2; stroke:black;"/>

    <text x="{$tx}" y="609"
          style="font-size:10pt;"
          transform="rotate(45,{$tx},609)">
      <xsl:value-of select="phaseName"/>
    </text>
  </xsl:for-each>

</xsl:template>

<!-- ============================================================   -->
<!-- The template prints the upper border of the current series.     -->
<!-- This template needs the scaling factor in order to calculate    -->
<!-- the right position of the points. Additionally it needs the     -->
<!-- number of phase whose value is to be printed. Because a line    -->
<!-- is printed from value to value the proceeding coordinates are   -->
```

```
<!-- needed, too.                                                    -->
<!-- The templates calls itself recursivly, this is necessary for    -->
<!-- the proceeding values.                                          -->
<!-- ============================================================   -->
<xsl:template name="UpperBorder phases">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <xsl:variable name="ts"
               select="//*[@id='Upper Border']/data/
                        entry[position()=($count)]/realts"/>
  <xsl:variable name="vl"
      select="//*[@id='Upper Border']/data/entry[position()=($count)]/vl"/>

 <xsl:variable name="x"
            select="($ts) * 500 div ($size) - 500 div ($size) div 2 + 60"/>
 <xsl:variable name="y"
               select="600 - ($vl) * 500 div ($scale)"/>

  <!-- Test if the entry is the first one -->
 <xsl:if test="not($count=1)">
   <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
         style="stroke-width:3; stroke:blue;"/>
 </xsl:if>

  <!-- Test if another recursive call is necessary -->
  <xsl:if test="not(($size) = ($count))">
    <xsl:call-template name="UpperBorder phases">
      <xsl:with-param name="scale">
        <xsl:value-of select="($scale)"/>
      </xsl:with-param>
      <xsl:with-param name="count">
        <xsl:value-of select="($count) + 1"/>
      </xsl:with-param>
      <xsl:with-param name="oldX">
        <xsl:value-of select="($x)"/>
      </xsl:with-param>
      <xsl:with-param name="oldY">
        <xsl:value-of select="($y)"/>
      </xsl:with-param>
      </xsl:call-template>
    </xsl:if>

  </xsl:template>

  <!-- ============================================================   -->
  <!-- The template marks the points of the current series that are   -->
  <!-- too high.                                                       -->
  <!-- This template needs the scaling factor in order to calculate   -->
  <!-- the right position of the points.                               -->
  <!-- ============================================================   -->
  <xsl:template name="TooHighValues phases">
    <xsl:param name="scale"/>

    <xsl:variable name="size"
            select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

    <xsl:for-each select="//*[@id='Values (above)']/data/timeseries/entry">

      <xsl:variable name="x"
              select="ts * 500 div ($size) - 500 div ($size) div 2 + 60"/>
      <xsl:variable name="y"
                    select="600 - value * 500 div ($scale)"/>
      <circle cx="{$x}" cy="{$y}" r="10"
              style="stroke-width:2; stroke:maroon; fill:none;"/>
    </xsl:for-each>
  </xsl:template>

  <!-- ============================================================   -->
  <!-- The template prints the current series.                         -->
  <!-- This template needs the scaling factor in order to calculate    -->
  <!-- the right position of the points. Additionally it needs the     -->
  <!-- number of phase whose value is to be printed. Because a line    -->
  <!-- is printed from value to value the proceeding coordinates are   -->
  <!-- needed, too.                                                    -->
  <!-- The templates calls itself recursivly, this is necessary for    -->
  <!-- the proceeding values.                                          -->
  <!-- ============================================================   -->
  <xsl:template name="Controlled series phases">
    <xsl:param name="scale"/>
    <xsl:param name="count"/>
    <xsl:param name="oldX"/>
    <xsl:param name="oldY"/>

    <xsl:variable name="size"
```

```
         select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

 <xsl:variable name="ts"
              select="//*[@id='Controlled Series']/data/
                      entry[position()=($count)]/realts"/>
 <xsl:variable name="vl"
              select="//*[@id='Controlled Series']/data/
                      entry[position()=($count)]/vl"/>
<xsl:variable name="x"
       select="($ts) * 500 div ($size) - 500 div ($size) div 2 + 60"/>
<xsl:variable name="y"
              select="600 - ($vl) * 500 div ($scale)"/>
<circle cx="{$x}" cy="{$y}" r="5"
        style="stroke-width:2 ; stroke: red; fill:red;"/>

 <!-- Test if the entry is the first one -->
<xsl:if test="not($count=1)">
  <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
        style="stroke-width:3; stroke:red;"/>
</xsl:if>

 <!-- Test if another recursive call is necessary -->
 <xsl:if test="not(($size) = ($count))">
   <xsl:call-template name="Controlled series phases">
     <xsl:with-param name="scale">
       <xsl:value-of select="($scale)"/>
     </xsl:with-param>
     <xsl:with-param name="count">
       <xsl:value-of select="($count) + 1"/>
     </xsl:with-param>
     <xsl:with-param name="oldX">
       <xsl:value-of select="($x)"/>
     </xsl:with-param>
     <xsl:with-param name="oldY">
       <xsl:value-of select="($y)"/>
     </xsl:with-param>
   </xsl:call-template>
 </xsl:if>

</xsl:template>

<!-- ============================================================  -->
<!-- The template marks the points of the current series that are  -->
<!-- too low.                                                      -->
<!-- This template needs the scaling factor in order to calculate  -->
<!-- the right position of the points.                             -->
<!-- ============================================================  -->
<xsl:template name="TooLowValues phases">
  <xsl:param name="scale"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <xsl:for-each select="//*[@id='Values (below)']/data/timeseries/entry">

    <xsl:variable name="x"
             select="ts * 500 div ($size) - 500 div ($size) div 2 + 60"/>
    <xsl:variable name="y"
                  select="600 - value * 500 div ($scale)"/>
    <circle cx="{$x}" cy="{$y}" r="10"
            style="stroke-width:2; stroke:maroon; fill:none;"/>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================  -->
<!-- The template prints the lower border of the current series.   -->
<!-- This template needs the scaling factor in order to calculate  -->
<!-- the right position of the points. Additionally it needs the   -->
<!-- number of phase whose value is to be printed. Because a line  -->
<!-- is printed from value to value the proceeding coordinates are -->
<!-- needed, too.                                                  -->
<!-- The templates calls itself recursivly, this is necessary for  -->
<!-- the proceeding values.                                        -->
<!-- ============================================================  -->
<xsl:template name="LowerBorder phases">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <xsl:variable name="ts"
               select="//*[@id='Lower Border']/data/
                       entry[position()=($count)]/realts"/>
  <xsl:variable name="vl"
               select="//*[@id='Lower Border']/data/
                       entry[position()=($count)]/vl"/>
 <xsl:variable name="x"
```

```
        select="($ts) * 500 div ($size) - 500 div ($size) div 2 + 60"/>
  <xsl:variable name="y"
                select="600 - ($vl) * 500 div ($scale)"/>

   <!-- Test if the entry is the first one -->
  <xsl:if test="not($count=1)">
    <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
          style="stroke-width:3; stroke:green;"/>
  </xsl:if>

   <!-- Test if another recursive call is necessary -->
   <xsl:if test="not(($size) = ($count))">
     <xsl:call-template name="LowerBorder phases">
       <xsl:with-param name="scale">
         <xsl:value-of select="($scale)"/>
       </xsl:with-param>
       <xsl:with-param name="count">
         <xsl:value-of select="($count) + 1"/>
       </xsl:with-param>
       <xsl:with-param name="oldX">
         <xsl:value-of select="($x)"/>
       </xsl:with-param>
       <xsl:with-param name="oldY">
         <xsl:value-of select="($y)"/>
       </xsl:with-param>
     </xsl:call-template>
   </xsl:if>

 </xsl:template>
</xsl:stylesheet>
```

B.2 Aufwandsverteilung auf Phasen und Aktivitäten (*Phases Line Plot With Activities*)

```
<!-- ================================================================  -->
<!-- The stylesheet creates a set of diagramms. The first diagramm shows -->
<!-- the effort of all phases and the other diagramms show the effort   -->
<!-- of the activities of one phase for each phase. The stylesheet does  -->
<!-- not work correct if there are no activities for none of the phases. -->
<!-- This stylesheet creates a svg file that can be viewed with the      -->
<!-- batik svgviewers (version 1.1.1 and 1.5) and with the microsoft     -->
<!-- internet explorer. Unfortunately batik 1.1.1 does not support all   -->
<!-- features. Therefore it was necessary to scale the different         -->
<!-- diagramms down and to add a link to change the viewbox manually.    -->
<!--                                                                     -->
<!-- First a set of symbols that are needed by more than one diagramm    -->
<!-- are defined and the two scripts that change the viewbox (needed by  -->
<!-- the internet explorer). Then two templates follow, the first one    -->
<!-- creates the phases overview and the second create for each phase    -->
<!-- the activity overview. Then follows a template which labels the     -->
<!-- y-axis with the correct effort values. After that a set of templates-->
<!-- follow that fill the phases overview diagramm with data (lower      -->
<!-- border, current series and so on). Last the templates follow that   -->
<!-- do the same for the overview of the activities.                     -->
<!-- ================================================================  -->
<xsl:stylesheet version="1.0"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="svg"
       doctype-public="-//W3c//DTD SVG 1.0//EN"
       doctype-system="http://w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"/>

  <!-- ==============================================================  -->
  <!-- The "master" template, sets the viewbox and calls the other     -->
  <!-- templates that do the work and defines a set of symbols.        -->
  <!-- ==============================================================  -->
  <xsl:template match="globaldata">

    <svg xml:space="preserve"
         xmlns="http://www.w3.org/2000/svg"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         x="0cm" y="0cm" width="15cm" height="15cm"
         viewBox="0 0 800 800"
         id="master">
      <desc> In the 'main' template only the symbols and the script are
          defined . The 'real' work is done by other templates.
      </desc>

      <!-- ============================================================  -->
      <!-- ECMAScript, that changes the viewbox if an phase has been    -->
      <!-- clicked.                                                      -->
      <!-- ============================================================  -->
      <script type="text/ecmascript"> <![CDATA[
```

```
  function phase_click(phase, row) {
    var master = document.getElementById("master");
    if ((phase % 2) == 0){
      var viewBoxY1 = phase * 400 / row;
      var viewBoxY2 = 800 / row;
      var newViewBox = "490 " + viewBoxY1.toString() +
                 " "+ viewBoxY2.toString() + " "+ viewBoxY2.toString();
      master.setAttribute("viewBox", newViewBox);
    }
    else{
      var viewBoxY1 = (phase + 1) * 400 / row;
      var viewBoxY2 = 800 / row;
      var newViewBox = "90 " + viewBoxY1.toString() +
                 " "+ viewBoxY2.toString() + " "+ viewBoxY2.toString();
      master.setAttribute("viewBox", newViewBox);
    }
  }
]]> </script>

<!-- ========================================================   -->
<!-- ECMAScript, that changes the viewbox if the user wishes to   -->
<!-- change to phases overview.                                   -->
<!-- ========================================================   -->
<script type="text/ecmascript"> <![CDATA[
  function activity_click(row) {
    var master = document.getElementById("master");
    var viewBoxY2 = 800 / row;
    var newViewBox = "490 0" +
               " "+ viewBoxY2.toString() + " "+ viewBoxY2.toString();
    master.setAttribute("viewBox", newViewBox);
  }
]]> </script>

<!-- ========================================================   -->
<!--  Define a symbol for the coordinate system.                 -->
<!-- ========================================================   -->
<defs>
  <desc> The coordinate system is defined here as a symbol, in
      order to use it later and to simplify the stylesheet.
  </desc>
  <symbol id="coordinateSystem">
        <!-- X-axis -->
    <line x1="50" y1="550" x2="550" y2="550"
          style="stroke-width:2 ; stroke: black;"/>
        <!-- Y-axis (greater zero)-->
    <line x1="50" y1="550" x2="50" y2="50"
          style="stroke-width:2 ; stroke: black;"/>
        <!-- the scaling of the y-axis without values -->
    <line x1="45" y1="50" x2="55" y2="50"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="100" x2="55" y2="100"
          style="stroke-width:2 ; stroke: black;"/>
    <line x1="45" y1="150" x2="55" y2="150"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="200" x2="55" y2="200"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="250" x2="55" y2="250"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="300" x2="55" y2="300"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="350" x2="55" y2="350"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="400" x2="55" y2="400"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="450" x2="55" y2="450"
          style="stroke-width:2; stroke:black;"/>
    <line x1="45" y1="500" x2="55" y2="500"
          style="stroke-width:2; stroke:black;"/>
  </symbol>
</defs>

<!-- ========================================================   -->
<!--  Define a symbol for the title.                             -->
<!-- ========================================================   -->
<defs>
  <desc> The title of the diagramm is defined as a symbol.
  </desc>
  <symbol id="title">
    <text x="320" y="50"
          style="text-anchor: middle; font-size:14pt;">
      <xsl:value-of select="//parameter[@id='Title']/@value"/>
    </text>
  </symbol>
</defs>

<!-- ========================================================   -->
<!--  Define a symbol for labeling the y-axis.                   -->
<!-- ========================================================   -->
<defs>
  <desc> The unit of the y-axis is defined as a symbol.
```

```
  </desc>
  <symbol id="unit y-axis">
    <text x="50" y="80"
          style="text-anchor: middle; font-size:12pt;">
      <xsl:value-of select="//parameter[@id='y-Axis Title']/@value"/>
    </text>
  </symbol>
</defs>

<!-- ========================================================   -->
<!--  Define a symbol for the legend.                           -->
<!-- ========================================================   -->
<defs>
  <desc> The legend is defined as symbol, because the legend
      is the same for all diagramms. The legend describes
      what the different curves mean.
  </desc>
  <symbol id="legend">
    <line x1="562" y1="115" x2="600" y2="115"
          style="stroke-width:3; stroke:green;"/>
    <text x="607" y="120"
          style="font-size:12pt;">Lower bound</text>

    <line x1="562" y1="140" x2="600" y2="140"
          style="stroke-width:2; stroke:red;"/>
    <circle cx="581" cy="140" r="4"
            style="stroke-width:2; stroke:red; fill:red;"/>
    <text x="607" y="145"
          style="font-size:12pt;">Current curve values</text>

    <line x1="562" y1="165" x2="600" y2="165"
          style="stroke-width:3; stroke:blue;"/>
    <text x="607" y="170"
          style="font-size:12pt;">Upper bound</text>

    <line x1="562" y1="190" x2="600" y2="190"
          style="stroke-width:2 ; stroke: red;"/>
    <circle cx="581" cy="190" r="4"
            style="stroke-width:2; stroke:red; fill:red;"/>
    <circle cx="581" cy="190" r="8"
            style="stroke-width:2; stroke:maroon; fill:none;"/>
    <text x="607" y="195"
          style="font-size:12pt;">Out of bound values</text>
  </symbol>
</defs>
```

```
    <!-- ========================================================   -->
    <!-- Define a symbol for additional information about activities. -->
    <!-- ========================================================   -->
    <defs>
      <symbol id="activity link" >
        <line x1="603" y1="222" x2="737" y2="222"
              style="stroke-width:30 ; stroke: yellow;"/>
        <text x="607" y="230"
              style="font-size:14pt;">Back to phases</text>
      </symbol>
    </defs>

    <xsl:call-template name="activity overview"/>
    <xsl:call-template name="Phases overview"/>
    <xsl:call-template name="ViewBox"/>

  </svg>
</xsl:template>

<!-- ============================================================   -->
<!-- The template creates a link to change the viewBox manually if   -->
<!-- automatical linking does not work (e.g. batik 1.1.1).           -->
<!-- ============================================================   -->
<xsl:template name="ViewBox">
  <xsl:variable name="numberOfPhases"
             select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
  <xsl:variable name="rows"
                select="ceiling(((($numberOfPhases) + 1) div 2)"/>
  <xsl:variable name="onload"
                select="concat('activity_click(',
                               ($rows), ')')"/>

  <!-- If automatical linking does not work, the text and a link  -->
  <!-- is shown to change the viewboy manually.                   -->
  <g onload="{$onload}">
    <xsl:variable name="y"
                  select="800 div ($rows) div 2 - 30"/>
    <text x="0" y="{$y}"
          style="font-size:16pt;">
      <tspan x="70" dy="0">Sorry, but your svgviewer does </tspan>
      <tspan x="70" dy="30">not support automatical linking.</tspan>
    </text>
    <xsl:variable name="y"
                  select="($y) + 65"/>
```

```
        <xsl:variable name="width"
                      select="800 div ($rows)"/>
        <xsl:variable name="hrefString"
                      select="concat('#svgView(viewBox(490,0,',
                                     ($width),',',
                                     ($width),'))')"/>
      <a xlink:href="{$hrefString}">
        <text x="100" y="{$y}"
              style="font-size:16pt; text-decoration:underline; fill:blue">
          <tspan x="80">Please click here to start.</tspan>
        </text>
      </a>
    </g>
  </xsl:template>

  <!-- ============================================================   -->
  <!-- The template creates the diagramm that shows the overview       -->
  <!-- over all phases.                                                -->
  <!-- ============================================================   -->
<xsl:template name="Phases overview">
    <xsl:variable name="numberOfPhases"
               select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
    <xsl:variable name="rows"
                select="ceiling(((($numberOfPhases) + 1) div 2)"/>
    <xsl:variable name="scale"
                select="1 div ($rows)"/>
    <xsl:variable name="transform"
                select="concat('translate(500,0) scale(',($scale), ')')"/>
    <g transform="{$transform}">
      <desc> Prints a diagramm that contains the view of all phases.
      </desc>
      <use xlink:href="#coordinateSystem"
           x="0" y="50" width="20cm" height="15cm"/>
      <use xlink:href="#legend"
           x="0" y="0" width="20cm" height="15cm"/>
      <use xlink:href="#unit y-axis"
           x="0" y="0" width="20cm" height="15cm"/>
      <use xlink:href="#title"
           x="0" y="0" width="20cm" height="15cm"/>

      <!-- Sets the title of the diagramm                        -->
      <xsl:variable name="titleTotal"
                    select="'(Phases overview)'"/>
      <text x="320" y="80"
            style="text-anchor: middle; font-size:14pt;">
        <xsl:value-of select="($titleTotal)"/>
      </text>

      <!-- Print the unit of the x-axis.                          -->
      <text x="600" y="607"
            style="text-anchor: middle; font-size:12pt;">
        <xsl:value-of select="//parameter[@id='x-Axis Title 1']/@value"/>
      </text>

      <!-- Label the x-axis.                                      -->
      <xsl:call-template name="label x axis with phases"/>

      <text x="607" y="265"
            style="font-size:14pt;">Click on a phase</text>
      <text x="609" y="290"
            style="font-size:14pt;">for details.</text>

      <!-- First read the maximum value of the current series       -->
      <!-- (has been calculatet before). The maximum value, that    -->
      <!-- could be diagrammed is about 10 percent greater, and     -->
      <!-- additionally the value is not 0 (therefor adding 1).     -->
      <xsl:variable name="max"
                    select="//*[@id='Maxima']/data/entry[1]/vl"/>

      <xsl:variable name="upperBound"
                    select="//*[@id='Upper Border']/data/entry/vl"/>
      <xsl:variable name="upperBoundMax"
                    select="$upperBound[not($upperBound &gt; .)]"/>

      <xsl:variable name="max"
                    select="($max &gt;= $upperBoundMax) * $max
                            + ($max &lt; $upperBoundMax) * $upperBoundMax"/>

      <xsl:variable name="max"
                    select="($max) * 1.1 "/>
      <xsl:variable name="max"
                    select="ceiling( ($max) ) +1"/>
      <xsl:variable name="dist"
                    select=" ($max) div 10"/>

      <!-- Now call the template for labeling the y-axis with the   -->
      <!-- calculated values. There is no need for a translational  -->
      <!-- displacement.                                            -->
      <xsl:call-template name="label y-axis">
        <xsl:with-param name="scale">
```

```
    <xsl:value-of select="($max)"/>
  </xsl:with-param>
  <xsl:with-param name="dist">
    <xsl:value-of select="($dist)"/>
  </xsl:with-param>
  <xsl:with-param name="deltadist">
    <xsl:value-of select="($dist)"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for printing the upper border of  -->
<!-- the current series (phase level).                        -->
<xsl:call-template name="UpperBorder phases">
  <xsl:with-param name="scale">
    <xsl:value-of select="($max)"/>
  </xsl:with-param>
  <xsl:with-param name="count">
    <xsl:value-of select="1"/>
  </xsl:with-param>
  <xsl:with-param name="oldX">
    <xsl:value-of select="0"/>
  </xsl:with-param>
  <xsl:with-param name="oldY">
    <xsl:value-of select="0"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for marking the values of the   -->
<!-- current series that are too high (phase level).        -->
<xsl:call-template name="TooHighValues phases">
  <xsl:with-param name="scale">
    <xsl:value-of select="($max)"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for printing the current series -->
<!-- (phase level).                                         -->
<xsl:call-template name="Controlled series phases">
  <xsl:with-param name="scale">
    <xsl:value-of select="($max)"/>
  </xsl:with-param>
  <xsl:with-param name="count">
    <xsl:value-of select="1"/>
  </xsl:with-param>
  <xsl:with-param name="oldX">
    <xsl:value-of select="0"/>
  </xsl:with-param>
  <xsl:with-param name="oldY">
    <xsl:value-of select="0"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for marking the values of the   -->
<!-- current series that are too low (phase level).         -->
<xsl:call-template name="TooLowValues phases">
  <xsl:with-param name="scale">
    <xsl:value-of select="($max)"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Now call the template for printing the lower border of   -->
<!-- the current series (phase level).                        -->
<xsl:call-template name="LowerBorder phases">
  <xsl:with-param name="scale">
    <xsl:value-of select="($max)"/>
  </xsl:with-param>
  <xsl:with-param name="count">
    <xsl:value-of select="1"/>
  </xsl:with-param>
  <xsl:with-param name="oldX">
    <xsl:value-of select="0"/>
  </xsl:with-param>
  <xsl:with-param name="oldY">
    <xsl:value-of select="0"/>
  </xsl:with-param>
</xsl:call-template>

  </g>
</xsl:template>

<!-- ============================================================   -->
<!-- The template creates for each phase the overview over the       -->
<!-- activities of this phase.                                       -->
<!-- ============================================================   -->
<xsl:template name="activity overview">
  <xsl:for-each select="//*[@id='Phases and Activities']/data/entry">
    <xsl:variable name="numberOfPhases"
                  select="../numberOfPhases"/>
    <xsl:variable name="rows"
                  select="ceiling((($numberOfPhases) + 1) div 2)"/>
```

```
<xsl:variable name="scale"
            select="1 div ($rows)"/>
<xsl:variable name="positionY"
            select="ceiling((position()) div 2) * ($scale) * 800"/>
<xsl:variable name="positionX"
              select="((position() + 1) mod 2) * 400 + 100"/>
<xsl:variable name="transform"
              select="concat('translate(',
                             ($positionX),
                             ',',
                             ($positionY),
                             ') ')"/>
<xsl:variable name="transform"
              select="concat(($transform),
                              'scale(',
                              ($scale),
                              ')')"/>
<g transform="{$transform}">
  <desc> Prints a diagramm that contains the view of all activities
      of one phase.
  </desc>
  <use xlink:href="#legend"
       x="0" y="0" width="20cm" height="15cm"/>
  <xsl:variable name="width"
                select="800 div ($rows)"/>
  <xsl:variable name="hrefString"
                select="concat('#svgView(viewBox(490,0,',
                               ($width),',',
                               ($width),'))')"/>
  <xsl:variable name="onclickString"
                select="concat('activity_click(',
                               ($rows),')')"/>
  <a xlink:href="{$hrefString}">
    <g onclick="{$onclickString}">
    <use xlink:href="#activity link"
         x="0" y="0" width="20cm" height="15cm"/>
    </g>
  </a>
  <use xlink:href="#unit y-axis"
       x="0" y="0" width="20cm" height="15cm"/>
  <use xlink:href="#title"
       x="0" y="0" width="20cm" height="15cm"/>

  <xsl:variable name="positionY" select="($positionY) + 50 "/>
  <use xlink:href="#coordinateSystem"
       x="0" y="50" width="20cm" height="15cm"/>

<!-- Sets the title of the diagramm                          -->
<xsl:variable name="titleTotal"
              select="concat('(Phase: ', phaseName, ')')"/>
<xsl:variable name="positionY"
              select="($positionY) + 30 "/>
<text x="320" y="80"
      style="text-anchor: middle; font-size:14pt;">
  <xsl:value-of select="($titleTotal)"/>
</text>

<!-- Label the x-axis with the corresponding unit            -->
<xsl:variable name="positionY" select="position() * 1000 + 607 "/>
<text x="600" y="607"
      style="text-anchor: middle; font-size:12pt;">
  <xsl:value-of select="//parameter[@id='x-Axis Title 2']/@value"/>
</text>

<!-- Label the x-axis                                        -->
<xsl:call-template name="label x axis with activities">
  <xsl:with-param name="phase">
    <xsl:value-of select="position()"/>
  </xsl:with-param>
</xsl:call-template>

<xsl:variable name="factor"
              select="//parameter[@id='Phases Factor']/@value"/>

<!-- Preparation for labeling the y-axis.                     -->
<!-- First read the maximum value of the current series       -->
<!-- (has been calculatet before). The maximum value, that    -->
<!-- could be diagrammed is about 10 percent greater, and     -->
<!-- additionally the value is not 0 (therfor adding 1).      -->
<xsl:variable name="count"
              select="phaseNumber + 1"/>
<xsl:variable name="scale"
              select="//*[@id='Maxima']/data/entry[($count)]/vl"/>

<xsl:variable name="upperBound"
              select="//*[@id='Upper Border']/data/
                    entry[(ceiling(realts div $factor) = ($count))
                          and (not (realts mod $factor = 0))]/vl"/>
<xsl:variable name="upperBoundMax"
              select="$upperBound[not($upperBound &gt; .) ]"/>
```

```
<xsl:variable name="scale"
              select="($scale &gt;= $upperBoundMax) * $scale
                      + ($scale &lt; $upperBoundMax) * $upperBoundMax"/>

<xsl:variable name="scale"
              select="($scale) * 1.1 "/>
<xsl:variable name="scale"
              select="ceiling( ($scale) ) + 1"/>
<xsl:variable name="dist"
              select=" ($scale) div 10"/>

<!-- Then call the template for labeling the y-axis with the  -->
<!-- calculated values.                                        -->
<xsl:call-template name="label y-axis">
  <xsl:with-param name="scale">
    <xsl:value-of select="($scale)"/>
  </xsl:with-param>
  <xsl:with-param name="dist">
    <xsl:value-of select="($dist)"/>
  </xsl:with-param>
  <xsl:with-param name="deltadist">
    <xsl:value-of select="($dist)"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for printing the upper border of  -->
<!-- the current series (activity level).                     -->
<xsl:call-template name="UpperBorder activities">
  <xsl:with-param name="scale">
    <xsl:value-of select="($scale)"/>
  </xsl:with-param>
  <xsl:with-param name="count">
    <xsl:value-of select="1"/>
  </xsl:with-param>
  <xsl:with-param name="phase">
    <xsl:value-of select="phaseNumber"/>
  </xsl:with-param>
  <xsl:with-param name="size">
    <xsl:value-of select="./activities/numberOfActivities"/>
  </xsl:with-param>
  <xsl:with-param name="oldX">
    <xsl:value-of select="0"/>
  </xsl:with-param>
  <xsl:with-param name="oldY">
    <xsl:value-of select="0"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for printing the upper border of  -->
<!-- the current series (activity level).                     -->
<xsl:call-template name="TooHighValues activities">
  <xsl:with-param name="scale">
   <xsl:value-of select="($scale)"/>
  </xsl:with-param>
  <xsl:with-param name="phase">
   <xsl:value-of select="phaseNumber"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for printing the current series -->
<!-- (activity level).                                       -->
<xsl:call-template name="Controlled series activities">
  <xsl:with-param name="scale">
    <xsl:value-of select="($scale)"/>
  </xsl:with-param>
  <xsl:with-param name="count">
    <xsl:value-of select="1"/>
  </xsl:with-param>
 <xsl:with-param name="phase">
    <xsl:value-of select="phaseNumber"/>
  </xsl:with-param>
  <xsl:with-param name="size">
    <xsl:value-of select="./activities/numberOfActivities"/>
  </xsl:with-param>
  <xsl:with-param name="oldX">
    <xsl:value-of select="0"/>
  </xsl:with-param>
  <xsl:with-param name="oldY">
    <xsl:value-of select="0"/>
  </xsl:with-param>
</xsl:call-template>

<!-- Then call the template for marking the values of the   -->
<!-- current series that are too low (activity level).      -->
<xsl:call-template name="TooLowValues activities">
  <xsl:with-param name="scale">
    <xsl:value-of select="($scale)"/>
  </xsl:with-param>
  <xsl:with-param name="phase">
```

```
          <xsl:value-of select="phaseNumber"/>
        </xsl:with-param>
      </xsl:call-template>

      <!-- Now call the template for printing the lower border of   -->
      <!-- the current series (activity level).                     -->
      <xsl:call-template name="LowerBorder activities">
        <xsl:with-param name="scale">
          <xsl:value-of select="($scale)"/>
        </xsl:with-param>
        <xsl:with-param name="count">
          <xsl:value-of select="1"/>
        </xsl:with-param>
        <xsl:with-param name="phase">
          <xsl:value-of select="phaseNumber"/>
        </xsl:with-param>
        <xsl:with-param name="size">
          <xsl:value-of select="./activities/numberOfActivities"/>
        </xsl:with-param>
        <xsl:with-param name="oldX">
          <xsl:value-of select="0"/>
        </xsl:with-param>
        <xsl:with-param name="oldY">
          <xsl:value-of select="0"/>
        </xsl:with-param>
      </xsl:call-template>

    </g>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the y-axis with the corresponding values.   -->
<!-- It needs the scale value (the maximum value), the distance      -->
<!-- between x-axis and the last value and the distance between two  -->
<!-- values of the scale bar.                                        -->
<!-- The template calls itself recursive until the maximum value is  -->
<!-- reached (scale = dist).                                         -->
<!-- ============================================================   -->
<xsl:template name="label y-axis">
  <xsl:param name="scale"/>
  <xsl:param name="dist"/>
  <xsl:param name="deltadist"/>

  <xsl:variable name="dy"
                select="550 - ($dist) * 500 div ($scale)"/>
  <xsl:variable name="dyt"
                select="($dy) + 56"/>
  <text x="20" y="{$dyt}"
        style="font-size:12pt;text-anchor:middle">
    <xsl:value-of select="($dist)"/>
  </text>

  <!-- Preparation for recursive call -->
  <xsl:variable name ="newdist"
                select="($dist) + ($deltadist)"/>
  <xsl:variable name ="newdist"
                select="floor( ($newdist) * 100)"/>
  <xsl:variable name ="newdist"
                select=" ($newdist) div 100"/>

  <!-- Test if recursiv call is necessary -->
  <xsl:if test="not( ($scale) &lt; ($newdist) )">
    <xsl:call-template name="label y-axis">
      <xsl:with-param name="scale">
        <xsl:value-of select="($scale)"/>
      </xsl:with-param>
      <xsl:with-param name="dist">
        <xsl:value-of select="($newdist)"/>
      </xsl:with-param>
      <xsl:with-param name="deltadist">
        <xsl:value-of select="($deltadist)"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:if>

</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the x-axis with the names of the different  -->
<!-- phases. For each phase a interaction is possible. If the user   -->
<!-- clicks on a phasename the corresponding diagramm is to be       -->
<!-- shown, this means that the viewbox of the document has to be    -->
<!-- adapted.                                                        -->
<!-- ============================================================   -->
<xsl:template name="label x axis with phases">
  <xsl:variable name="size"
            select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <!-- For each name first calculate the corresponding position then -->
```

```
  <!-- the corresponding viewbox parameters and then print the text. -->
  <xsl:for-each select="//*[@id='Phases and Activities']/data/entry">
    <xsl:variable name="x"
                  select="phaseNumber * 500 div ($size) + 50"/>
    <xsl:variable name="tx"
                  select="($x) - 500 div ($size) div 2 - 5"/>
   <line x1="{$x}" y1="605" x2="{$x}" y2="595"
         style="stroke-width:2; stroke:black;"/>

    <xsl:variable name="row"
                  select="ceiling((($size) + 1) div 2)"/>

    <xsl:variable name="viewboxY"
             select="((phaseNumber mod 2) + phaseNumber) * 400 div ($row)"/>
    <xsl:variable name="viewboxX"
              select="((phaseNumber + 1) mod 2) * 400 + 90"/>
    <xsl:variable name="width"
                  select="800 div ($row)"/>
    <xsl:variable name="hrefString"
                  select="concat('#svgView(viewBox(',
                                 ($viewboxX),',',
                                 ($viewboxY),',',
                                 ($width),',',
                                 ($width),'))')"/>
    <xsl:variable name="onclickString"
                  select="concat('phase_click(',
                                 phaseNumber,
                                 ',',($row),')')"/>
    <a xlink:href="{$hrefString}">
      <text onclick="{$onclickString}"
            x="{$tx}" y="609"
            style="font-size:10pt;"
            transform="rotate(45,{$tx},609)">
        <xsl:value-of select="phaseName"/>
      </text>
    </a>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================  -->
<!-- The template prints the upper border of the current series.   -->
<!-- This template needs the scaling factor in order to calculate  -->
<!-- the right position of the points. Additionally it needs the   -->
<!-- number of phase whose value is to be printed. Because a line  -->
<!-- is printed from value to value the proceeding coordinates are -->
<!-- needed, too.                                                  -->
<!-- The templates calls itself recursivly, this is necessary for  -->
<!-- the proceeding values.                                        -->
<!-- ============================================================  -->
<xsl:template name="UpperBorder phases">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <!-- Find the right entry in all entries. -->
  <xsl:for-each select="//*[@id='Upper Border']/data/entry">
    <xsl:variable name="ts"
                  select="realts div ($factor)"/>

    <!-- Test if the entry is the right one -->
    <xsl:if test="((realts mod ($factor)) = 0)
                   and (($ts) = ($count))">
     <xsl:variable name="x"
            select="($ts) * 500 div ($size) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                   select="600 - vl * 500 div ($scale)"/>

      <!-- Test if the entry is the first one -->
     <xsl:if test="not($count=1)">
       <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
              style="stroke-width:3; stroke:blue;"/>
     </xsl:if>

      <!-- Test if another recursive call is necessary -->
      <xsl:if test="not(($size) = ($count))">
        <xsl:call-template name="UpperBorder phases">
          <xsl:with-param name="scale">
            <xsl:value-of select="($scale)"/>
          </xsl:with-param>
          <xsl:with-param name="count">
            <xsl:value-of select="($count) + 1"/>
          </xsl:with-param>
          <xsl:with-param name="oldX">
            <xsl:value-of select="($x)"/>
```

```
          </xsl:with-param>
          <xsl:with-param name="oldY">
            <xsl:value-of select="($y)"/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>

    </xsl:if>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template marks the points of the current series that are    -->
<!-- too high.                                                       -->
<!-- This template needs the scaling factor in order to calculate    -->
<!-- the right position of the points.                               -->
<!-- ============================================================   -->
<xsl:template name="TooHighValues phases">
  <xsl:param name="scale"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <xsl:for-each select="//*[@id='Values (above)']/data/timeseries/entry">

    <!-- Test if the entry is of the phase -->
    <xsl:if test="(ts mod ($factor)) = 0">
      <xsl:variable name="ts"
                select="floor(ts div ($factor))"/>
      <xsl:variable name="x"
               select="$ts * 500 div ($size) - 500 div ($size) div 2 + 50"/>
      <xsl:variable name="y"
                select="600 - value * 500 div ($scale)"/>
      <circle cx="{$x}" cy="{$y}" r="10"
              style="stroke-width:2; stroke:maroon; fill:none;"/>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template prints the current series.                         -->
<!-- This template needs the scaling factor in order to calculate    -->
<!-- the right position of the points. Additionally it needs the     -->
<!-- number of phase whose value is to be printed. Because a line    -->
<!-- is printed from value to value the proceeding coordinates are   -->
<!-- needed, too.                                                    -->
<!-- The templates calls itself recursivly, this is necessary for    -->
<!-- the proceeding values.                                          -->
<!-- ============================================================   -->
<xsl:template name="Controlled series phases">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <!-- Find the right entry in all entries. -->
  <xsl:for-each select="//*[@id='Controlled Series']/data/entry">
    <xsl:variable name="ts"
                  select="realts div ($factor)"/>

    <!-- Test if the entry is the right one -->
    <xsl:if test="((realts mod ($factor)) = 0) and
                  (($ts) = ($count))">

     <xsl:variable name="x"
            select="($ts) * 500 div ($size) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                  select="600 - vl * 500 div ($scale)"/>
     <circle cx="{$x}" cy="{$y}" r="5"
             style="stroke-width:2 ; stroke: red; fill:red;"/>

      <!-- Test if the entry is the first one -->
     <xsl:if test="not($count=1)">
       <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
             style="stroke-width:3; stroke:red;"/>
     </xsl:if>

      <!-- Test if another recursive call is necessary -->
      <xsl:if test="not(($size) = ($count))">
        <xsl:call-template name="Controlled series phases">
          <xsl:with-param name="scale">
            <xsl:value-of select="($scale)"/>
          </xsl:with-param>
```

```
          <xsl:with-param name="count">
            <xsl:value-of select="($count) + 1"/>
          </xsl:with-param>
          <xsl:with-param name="oldX">
            <xsl:value-of select="($x)"/>
          </xsl:with-param>
          <xsl:with-param name="oldY">
            <xsl:value-of select="($y)"/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>

    </xsl:if>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template marks the points of the current series that are    -->
<!-- too low.                                                        -->
<!-- This template needs the scaling factor in order to calculate    -->
<!-- the right position of the points.                               -->
<!-- ============================================================   -->
<xsl:template name="TooLowValues phases">
  <xsl:param name="scale"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <xsl:for-each select="//*[@id='Values (below)']/data/timeseries/entry">

    <!-- Test if the entry is an right of the phases -->
    <xsl:if test="(ts mod ($factor)) = 0">
      <xsl:variable name="ts"
                    select="floor(ts div ($factor))"/>
      <xsl:variable name="x"
               select="$ts * 500 div ($size) - 500 div ($size) div 2 + 50"/>
      <xsl:variable name="y"
                    select="600 - value * 500 div ($scale)"/>
      <circle cx="{$x}" cy="{$y}" r="10"
              style="stroke-width:2; stroke:maroon; fill:none;"/>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template prints the lower border of the current series.     -->
<!-- This template needs the scaling factor in order to calculate    -->
<!-- the right position of the points. Additionally it needs the     -->
<!-- number of phase whose value is to be printed. Because a line    -->
<!-- is printed from value to value the proceeding coordinates are   -->
<!-- needed, too.                                                    -->
<!-- The templates calls itself recursivly, this is necessary for    -->
<!-- the proceeding values.                                          -->
<!-- ============================================================   -->
<xsl:template name="LowerBorder phases">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="size"
          select="//*[@id='Phases and Activities']/data/numberOfPhases"/>
  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <!-- Find the right entry in all entries. -->
  <xsl:for-each select="//*[@id='Lower Border']/data/entry">
    <xsl:variable name="ts"
                  select="realts div ($factor)"/>

    <!-- Test if the entry is the right one -->
    <xsl:if test="((realts mod ($factor) ) = 0) and
                  (($ts) = ($count))">
     <xsl:variable name="x"
            select="($ts) * 500 div ($size) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                   select="600 - vl * 500 div ($scale)"/>

      <!-- Test if the entry is the first one -->
     <xsl:if test="not($count=1)">
       <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
             style="stroke-width:3; stroke:green;"/>
     </xsl:if>

      <!-- Test if another recursive call is necessary -->
      <xsl:if test="not(($size) = ($count))">
        <xsl:call-template name="LowerBorder phases">
          <xsl:with-param name="scale">
```

```
          <xsl:value-of select="($scale)"/>
        </xsl:with-param>
        <xsl:with-param name="count">
          <xsl:value-of select="($count) + 1"/>
        </xsl:with-param>
        <xsl:with-param name="oldX">
          <xsl:value-of select="($x)"/>
        </xsl:with-param>
        <xsl:with-param name="oldY">
          <xsl:value-of select="($y)"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>

  </xsl:if>
</xsl:for-each>
</xsl:template>

<!-- ============================================================  -->
<!-- The template labels the x-axis with the names of the different -->
<!-- activities of a phase. This template needs the number of the   -->
<!-- phase in order to calculate the right position of the text.    -->
<!-- ============================================================  -->
<xsl:template name="label x axis with activities">
  <xsl:param name="phase"/>
  <xsl:variable name="size"
        select="./activities/numberOfActivities"/>

  <!-- For each name first calculate the corresponding position and -->
  <!-- then print the text.                                         -->
  <xsl:for-each select="./activities/entry">
    <xsl:variable name="x"
                  select="activityNumber * 500 div ($size) + 50"/>
    <xsl:variable name="tx"
                  select="($x) - 500 div ($size) div 2 - 5"/>
   <line x1="{$x}" y1="605" x2="{$x}" y2="595"
         style="stroke-width:2; stroke:black;"/>

    <text x="{$tx}" y="609"
          style="font-size:10pt;"
          transform="rotate(45,{$tx},609)">
      <xsl:value-of select="activityName"/>
    </text>
  </xsl:for-each>
</xsl:template>
```

```
<!-- ============================================================  -->
<!-- The template prints the upper border of the current series     -->
<!-- of a given phase (parameter).                                  -->
<!-- This template needs the scaling factor in order to calculate   -->
<!-- the right position of the points. Additionally it needs the    -->
<!-- number of activity whose value is to be printed. Because a line -->
<!-- is printed from value to value the proceeding coordinates are  -->
<!-- needed, too.                                                   -->
<!-- The templates calls itself recursivly, this is necessary for   -->
<!-- the proceeding values.                                         -->
<!-- ============================================================  -->
<xsl:template name="UpperBorder activities">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="phase"/>
  <xsl:param name="size"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <!-- Find the right entry in all entries. -->
  <xsl:for-each select="//*[@id='Upper Border']/data/entry">
    <xsl:variable name="ts"
                  select="realts div ($factor)"/>
    <xsl:variable name="ts"
                  select="floor($ts)"/>
    <xsl:variable name="tshelp"
                  select="realts mod ($factor)"/>

    <!-- Test if the entry is the right one -->
    <xsl:if test="(($tshelp) = ($count)) and
                  (($ts) = ($phase))">
     <xsl:variable name="x"
                   select="($tshelp) * 500 div ($size)"/>
     <xsl:variable name="x"
                   select="($x) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                   select="600 - vl * 500 div ($scale)"/>

      <!-- Test if the entry is the first one -->
     <xsl:if test="not($count=1)">
       <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
```

```
            style="stroke-width:3; stroke:blue;"/>
      </xsl:if>

       <!-- Test if another recursive call is necessary -->
       <xsl:if test="not(($size) = ($count))">
         <xsl:call-template name="UpperBorder activities">
           <xsl:with-param name="scale">
             <xsl:value-of select="($scale)"/>
           </xsl:with-param>
           <xsl:with-param name="count">
             <xsl:value-of select="($count) + 1"/>
           </xsl:with-param>
           <xsl:with-param name="phase">
             <xsl:value-of select="($phase)"/>
           </xsl:with-param>
           <xsl:with-param name="size">
             <xsl:value-of select="($size)"/>
           </xsl:with-param>
           <xsl:with-param name="oldX">
             <xsl:value-of select="($x)"/>
           </xsl:with-param>
           <xsl:with-param name="oldY">
             <xsl:value-of select="($y)"/>
           </xsl:with-param>
         </xsl:call-template>
       </xsl:if>

     </xsl:if>
   </xsl:for-each>
</xsl:template>

<!-- ============================================================  -->
<!-- The template marks the points of a given phase (parameter) of  -->
<!-- the current series that are too high, of a given phase.        -->
<!-- This template needs the scaling factor in order to calculate   -->
<!-- the right position of the points.                              -->
<!-- ============================================================  -->
<xsl:template name="TooHighValues activities">
  <xsl:param name="scale"/>
  <xsl:param name="phase"/>

  <xsl:variable name="size"
                select="./activities/numberOfActivities"/>
  <xsl:variable name="factor"
                select="//parameter[@id='Phases Factor']/@value"/>
```

```
  <xsl:for-each select="//*[@id='Values (above)']/data/timeseries/entry">
    <xsl:variable name="tsmod"
                  select="ts mod ($factor)"/>
    <xsl:variable name="ts"
                  select="floor(ts div ($factor))"/>

    <!-- Test if the entry is one of an activity of the phase -->
    <xsl:if test="(($ts) = ($phase)) and (not(($tsmod) = 0))">
     <xsl:variable name="x"
                   select="($tsmod) * 500 div ($size)"/>
     <xsl:variable name="x"
                   select="($x) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                   select="600 - value * 500 div ($scale)"/>
      <circle cx="{$x}" cy="{$y}" r="10"
              style="stroke-width:2; stroke:maroon; fill:none;"/>
    </xsl:if>

  </xsl:for-each>
</xsl:template>

<!-- ============================================================  -->
<!-- The template prints the values of the current series of a      -->
<!-- phase (parameter).                                             -->
<!-- This template needs the scaling factor in order to calculate   -->
<!-- the right position of the points. Additionally it needs the    -->
<!-- number of activity whose value is to be printed. Because a line -->
<!-- is printed from value to value the proceeding coordinates are  -->
<!-- needed, too.                                                   -->
<!-- The templates calls itself recursivly, this is necessary for   -->
<!-- the proceeding values.                                         -->
<!-- ============================================================  -->
<xsl:template name="Controlled series activities">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="phase"/>
  <xsl:param name="size"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <!-- Find the right entry in all entries. -->
```

```
<xsl:for-each select="//*[@id='Controlled Series']/data/entry">

  <xsl:variable name="ts"
                select="realts div ($factor)"/>
  <xsl:variable name="ts"
                select="floor($ts)"/>
  <xsl:variable name="tshelp"
                select="realts mod ($factor)"/>

  <!-- Test if the entry is the right one -->
  <xsl:if test="(($tshelp) = ($count)) and
                (($ts) = ($phase))">
   <xsl:variable name="x"
                 select="($tshelp) * 500 div ($size)"/>
   <xsl:variable name="x"
                 select="($x) - 500 div ($size) div 2 + 50"/>
   <xsl:variable name="y"
                 select="600 - vl * 500 div ($scale)"/>
   <circle cx="{$x}" cy="{$y}" r="5"
           style="stroke-width:2 ; stroke: red; fill:red;"/>

    <!-- Test if the entry is the first one -->
   <xsl:if test="not($count=1)">
     <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
           style="stroke-width:3; stroke:red;"/>
   </xsl:if>

    <!-- Test if another recursive call is necessary -->
    <xsl:if test="not(($size) = ($count))">
      <xsl:call-template name="Controlled series activities">
        <xsl:with-param name="scale">
          <xsl:value-of select="($scale)"/>
        </xsl:with-param>
        <xsl:with-param name="count">
          <xsl:value-of select="($count) + 1"/>
        </xsl:with-param>
        <xsl:with-param name="phase">
          <xsl:value-of select="($phase)"/>
        </xsl:with-param>
        <xsl:with-param name="size">
          <xsl:value-of select="($size)"/>
        </xsl:with-param>
        <xsl:with-param name="oldX">
          <xsl:value-of select="($x)"/>
        </xsl:with-param>
        <xsl:with-param name="oldY">
          <xsl:value-of select="($y)"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>

  </xsl:if>
 </xsl:for-each>
</xsl:template>

<!-- ==========================================================  -->
<!-- The template marks the points of a given phase (parameter) of  -->
<!-- the current series that are too low, of a given phase.         -->
<!-- This template needs the scaling factor in order to calculate   -->
<!-- the right position of the points.                              -->
<!-- ==========================================================  -->
<xsl:template name="TooLowValues activities">
  <xsl:param name="scale"/>
  <xsl:param name="phase"/>

  <xsl:variable name="size"
        select="./activities/numberOfActivities"/>
  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>
  <xsl:for-each select="//*[@id='Values (below)']/data/timeseries/entry">
    <xsl:variable name="ts"
                  select="floor(ts div ($factor))"/>
    <xsl:variable name="tsmod"
                  select="ts mod ($factor)"/>

    <!-- Test if the entry is one of an activity of the phase -->
    <xsl:if test="(($ts) = ($phase)) and (not(($tsmod) = 0))">
     <xsl:variable name="x"
                   select="($tsmod) * 500 div ($size)"/>
     <xsl:variable name="x"
                   select="($ts) * 500 div ($size)"/>
     <xsl:variable name="x"
                   select="($x) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                   select="600 - value * 500 div ($scale)"/>
      <circle cx="{$x}" cy="{$y}" r="10"
              style="stroke-width:2; stroke:maroon; fill:none;"/>
    </xsl:if>

  </xsl:for-each>
```

```
</xsl:template>

<!-- ============================================================   -->
<!-- The template prints the lower border of the current series     -->
<!-- of a given phase (parameter).                                  -->
<!-- This template needs the scaling factor in order to calculate   -->
<!-- the right position of the points. Additionally it needs the    -->
<!-- number of activity whose value is to be printed. Because a line -->
<!-- is printed from value to value the proceeding coordinates are  -->
<!-- needed, too.                                                   -->
<!-- The templates calls itself recursivly, this is necessary for   -->
<!-- the proceeding values.                                         -->
<!-- ============================================================   -->
<xsl:template name="LowerBorder activities">
  <xsl:param name="scale"/>
  <xsl:param name="count"/>
  <xsl:param name="phase"/>
  <xsl:param name="size"/>
  <xsl:param name="oldX"/>
  <xsl:param name="oldY"/>

  <xsl:variable name="factor"
          select="//parameter[@id='Phases Factor']/@value"/>

  <!-- Find the right entry in all entries. -->
  <xsl:for-each select="//*[@id='Lower Border']/data/entry">
    <xsl:variable name="ts"
                  select="realts div ($factor)"/>
    <xsl:variable name="ts"
                  select="floor($ts)"/>
    <xsl:variable name="tshelp"
                  select="realts mod ($factor)"/>

    <!-- Test if the entry is the right one -->
    <xsl:if test="(($tshelp) = ($count)) and
                  (($ts) = ($phase))">
     <xsl:variable name="x"
                   select="($tshelp) * 500 div ($size)"/>
     <xsl:variable name="x"
                   select="($x) - 500 div ($size) div 2 + 50"/>
     <xsl:variable name="y"
                   select="600 - vl * 500 div ($scale)"/>

      <!-- Test if the entry is the first one -->
     <xsl:if test="not($count=1)">
      <line x1="{$oldX}" y1="{$oldY}" x2="{$x}" y2="{$y}"
            style="stroke-width:3; stroke:green;"/>
     </xsl:if>

      <!-- Test if another recursive call is necessary -->
      <xsl:if test="not(($size) = ($count))">
        <xsl:call-template name="LowerBorder activities">
          <xsl:with-param name="scale">
            <xsl:value-of select="($scale)"/>
          </xsl:with-param>
          <xsl:with-param name="count">
            <xsl:value-of select="($count) + 1"/>
          </xsl:with-param>
          <xsl:with-param name="phase">
            <xsl:value-of select="($phase)"/>
          </xsl:with-param>
          <xsl:with-param name="size">
            <xsl:value-of select="($size)"/>
          </xsl:with-param>
          <xsl:with-param name="oldX">
            <xsl:value-of select="($x)"/>
          </xsl:with-param>
          <xsl:with-param name="oldY">
            <xsl:value-of select="($y)"/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>

    </xsl:if>
  </xsl:for-each>

</xsl:template>
</xsl:stylesheet>
```

B.3 Fehlerverteilung auf Phasen und Affiliation mit zugehörigem Aufwand (*Defect View*)

```
<!-- ============================================================  -->
<!-- The stylesheet creates a bar chart with the phases along the x-   -->
<!-- axis and an additional partitioning of each phase into the        -->
<!-- affiliation of the defects. The axis is only labeled with the      -->
<!-- phases, the affiliation are distinguished with different colors    -->
<!-- that are explained in the legend.                                  -->
<!-- The stylesheet has a maximum number of affiliation that could be   -->
<!-- drawn, if the number is greater, the according template has to be  -->
<!-- adapted or an additional input is needed, that specify the color.  -->
<!-- ============================================================  -->
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="svg"
    doctype-public="-//W3c//DTD SVG 1.0//EN"
    doctype-system="http://w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"/>

  <!-- ============================================================  -->
  <!-- The "master" template, sets the viewbox and calls the other     -->
  <!-- templates that do the work.                                     -->
  <!-- ============================================================  -->
  <xsl:template match="globaldata">

    <svg xml:space="preserve"
      xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      height="16cm" width="15cm"
      viewBox="0 0 755 700">

      <xsl:apply-templates
           select="//*[@id='Reported Defects']/data/entry/phase/entry/effort">
        <xsl:sort data-type="number" order="descending"/>
      </xsl:apply-templates>

      <xsl:call-template name="make coordinate system"/>

      <xsl:call-template name="PhasesAndActivities"/>

      <xsl:call-template name="makelegend"/>

      <xsl:call-template name="title"/>

      <xsl:call-template name="unit axis"/>

    </svg>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- Makes the coordinate system of the diagramm.                    -->
  <!-- ============================================================  -->
  <xsl:template name="make coordinate system">
    <!-- X-axis -->
    <line x1="60" y1="550" x2="560" y2="550"
          style="stroke-width:2 ; stroke: black;"/>
    <!-- Y-axis (greater zero)-->
    <line x1="60" y1="550" x2="60" y2="50"
          style="stroke-width:2 ; stroke: black;"/>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- The template searches the highest value that occures therefor   -->
  <!-- it is called with the sorted values. Then the template calcu-   -->
  <!-- lates the maximum to be 10 percent greater. With that value     -->
  <!-- the other templates are called.                                 -->
  <!-- ============================================================  -->
  <xsl:template
       match="//*[@id='Reported Defects']/data/entry/phase/entry/effort">
    <xsl:if test="position() = 1">
      <xsl:variable name="max"
                    select="."/>
      <xsl:variable name="max"
                    select="($max) * 1.1"/>
      <xsl:variable name="max"
                    select="ceiling( ($max) ) +1"/>

      <xsl:variable name="dist"
                    select=" ($max) div 10"/>

      <!-- scaling y-axis -->
      <xsl:call-template name="yaxis">
        <xsl:with-param name="scale">
          <xsl:value-of select="($max)"/>
        </xsl:with-param>
        <xsl:with-param name="dist">
          <xsl:value-of select="($dist)"/>
        </xsl:with-param>
        <xsl:with-param name="deltadist">
```

```
        <xsl:value-of select="($dist)"/>
      </xsl:with-param>
    </xsl:call-template>

    <xsl:call-template name="ReportedDefectsSorting">
      <xsl:with-param name="scale">
        <xsl:value-of select="($max)"/>
      </xsl:with-param>
    </xsl:call-template>

  </xsl:if>
</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the y-axis with the corresponding values.   -->
<!-- It needs the scale value (the maximum value), the distance      -->
<!-- between x-axis and the last value and the distance between two  -->
<!-- values of the scale bar.                                        -->
<!-- The template calls itself recursive until the maximum value is  -->
<!-- reached (scale = dist).                                         -->
<!-- ============================================================   -->
<xsl:template name="yaxis">
  <xsl:param name="scale"/>
  <xsl:param name="dist"/>
  <xsl:param name="deltadist"/>

  <xsl:variable name="dy"
                select="550 - ($dist) * 500 div ($scale)"/>
  <line x1="55" y1="{$dy}" x2="65" y2="{$dy}"
        style="stroke-width:2 ; stroke: black;"/>
  <xsl:variable name="dyt"
                select="($dy) + 6"/>
  <text x="30" y="{$dyt}"
        style="font-size:12pt;text-anchor:middle">
    <xsl:value-of select="($dist)"/>
  </text>

  <!-- Preparation for recursive call -->
  <xsl:variable name ="newdist"
                select="($dist) + ($deltadist)"/>
  <xsl:variable name ="newdist"
                select="floor( ($newdist) * 100)"/>
  <xsl:variable name ="newdist"
                select=" ($newdist) div 100"/>

  <!-- Test if recursiv call is necessary -->
  <xsl:if test="not( ($scale) &lt; ($newdist) )">
    <xsl:call-template name="yaxis">
      <xsl:with-param name="scale">
        <xsl:value-of select="($scale)"/>
      </xsl:with-param>
      <xsl:with-param name="dist">
        <xsl:value-of select="($newdist)"/>
      </xsl:with-param>
      <xsl:with-param name="deltadist">
        <xsl:value-of select="($deltadist)"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:if>
</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the x-axis with the names of the different  -->
<!-- phases.                                                         -->
<!-- ============================================================   -->
<xsl:template name="PhasesAndActivities">
  <xsl:variable name="size"
            select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <xsl:for-each select="//*[@id='Phases and Activities']/data/entry">
    <xsl:variable name="x"
                  select="phaseNumber * 500 div ($size) + 60"/>
    <xsl:variable name="tx"
                  select="($x) - 500 div ($size) div 2 + 5"/>
   <line x1="{$x}" y1="555" x2="{$x}" y2="545"
         style="stroke-width:2 ; stroke: black;"/>
    <text x="{$tx}" y="559"
          style="font-size:10pt;"
          transform="rotate(45,{$tx},559)">
      <xsl:value-of select="phaseName"/>
    </text>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template prints the bars for each phase and affiliation.    -->
<!-- The template has to be changed if the number of affiliation     -->
<!-- changes.  In a later version it is possible to have an input    -->
<!-- that specify the colors that are to be used.                    -->
<!-- ============================================================   -->
```

```
<xsl:template name="ReportedDefectsSorting">
  <xsl:param name="scale"/>
  <xsl:variable name="size"
            select="//*[@id='Phases and Activities']/data/numberOfPhases"/>

  <!-- the distance between two bars -->
  <xsl:variable name="maxA"
               select="4"/>
  <xsl:variable name="width"
               select="(500 div ($size) - 10) div ($maxA)"/>

  <!-- for each phase -->
  <xsl:for-each select="//entry/phase/entry">
    <xsl:variable name="phase"
                  select="../phaseNumber"/>
    <xsl:variable name="affiliation"
                  select="./defectAffiliationNumber"/>
    <xsl:variable name="x"
                  select = "65 + 500 div ($size)* (($phase) - 1)"/>
    <xsl:variable name="x"
                  select = "($x) + ($width)*($affiliation)"/>
    <xsl:variable name="x"
                  select = "($x)- ($width) div 2"/>
    <xsl:variable name="effort"
                  select="./effort"/>
    <xsl:variable name="y"
                  select="550 - (500 * ($effort) div ($scale) )"/>
    <xsl:variable name="barwidth"
                  select="($width) -2"/>

    <!-- decide which color is to be used. -->
    <xsl:choose>
      <xsl:when test="($affiliation) = 1">
        <line x1="{$x}" y1="550" x2="{$x}" y2="{$y}"
              style="stroke-width:{$barwidth} ; stroke: green;"/>
      </xsl:when>
      <xsl:when test="($affiliation) = 2">
        <line x1="{$x}" y1="550" x2="{$x}" y2="{$y}"
              style="stroke-width:{$barwidth} ; stroke: red;"/>
      </xsl:when>
      <xsl:when test="($affiliation) = 3">
        <line x1="{$x}" y1="550" x2="{$x}" y2="{$y}"
              style="stroke-width:{$barwidth} ; stroke: blue;"/>
      </xsl:when>
      <xsl:when test="($affiliation) = 4">
        <line x1="{$x}" y1="550" x2="{$x}" y2="{$y}"
              style="stroke-width:{$barwidth} ; stroke: orange;"/>
      </xsl:when>
    </xsl:choose>
    <xsl:variable name="tx"
                  select="($x)"/>
    <xsl:variable name="ty"
                  select="($y) - 10"/>
    <xsl:variable name="nod"
                  select="./numberOfDefects"/>
    <xsl:if test="not( ($nod)=0)">
      <text x="{$tx}" y="{$ty}"
            style="text-anchor: middle; font-size:10pt; fill:maroon;">
        <xsl:value-of select="($nod)"/>
      </text>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template makes the legend of the diagramm.                 -->
<!-- ============================================================   -->
<xsl:template name="makelegend">
  <text x="560" y="50"
        style="font-size:14pt;">Defect Sources</text>
  <line x1="570" y1="65" x2="570" y2="85"
        style="stroke-width:8 ; stroke: green;"/>
  <text x="587" y="83"
        style="font-size:12pt;">
    <xsl:value-of
         select="//*[@id='Defect Affiliation']/categories/category[1]/title"/>
  </text>
  <line x1="570" y1="90" x2="570" y2="110"
        style="stroke-width:8 ; stroke: red;"/>
  <text x="587" y="108"
        style="font-size:12pt;">
    <xsl:value-of
        select="//*[@id='Defect Affiliation']/categories/category[2]/title"/>
  </text>
  <line x1="570" y1="115" x2="570" y2="135"
        style="stroke-width:8 ; stroke: blue;"/>
  <text x="587" y="133"
        style="font-size:12pt;">
    <xsl:value-of
        select="//*[@id='Defect Affiliation']/categories/category[3]/title"/>
```

```
    </text>
    <line x1="570" y1="140" x2="570" y2="160"
          style="stroke-width:8 ; stroke: orange;"/>
    <text x="587" y="158"
          style="font-size:12pt;">
      <xsl:value-of
          select="//*[@id='Defect Affiliation']/categories/category[4]/title"/>
    </text>
    <text x="568" y="183"
          style="font-size:10pt; fill:maroon;">#</text>
    <text x="587" y="183"
          style="font-size:12pt;">No. of defects</text>
  </xsl:template>

  <!-- ============================================================   -->
  <!-- Prints the title of the diagramm.                              -->
  <!-- ============================================================   -->
  <xsl:template name="title">
    <text x="320" y="30"
          style="text-anchor: middle; font-size:14pt;">
      <xsl:value-of select="parameter[@id='Title']/@value"/>
    </text>
  </xsl:template>

  <!-- ============================================================   -->
  <!-- The template prints the unit of each axis.                     -->
  <!-- ============================================================   -->
  <xsl:template name="unit axis">
    <text x="60" y="30"
          style="text-anchor: middle; font-size:12pt;">
      <xsl:value-of select="parameter[@id='y-Axis Title']/@value"/>
    </text>
    <text x="600" y="557"
          style="text-anchor: middle; font-size:12pt;">
      <xsl:value-of select="parameter[@id='x-Axis Title']/@value"/>
    </text>
  </xsl:template>
</xsl:stylesheet>
```

B.4 Zeitlicher Projektverlauf (*Project Plan*)

```
<!-- ============================================================  -->
<!-- The stylesheet creates a diagramm that represents a project plan. -->
<!-- It ist possible to show a reference plan and the actual plan.    -->
<!-- The height of the diagramm is calculated automatically, this is  -->
<!-- because of the same distance between the phases (activities)     -->
<!-- along the y-axis. If the distance is varied the labeling of the  -->
<!-- axis could not be read by increasing number of entries. By       -->
<!-- clicking on a line the start and end of the phase (activity) is  -->
<!-- shown in an alert box.                                           -->
<!-- ============================================================  -->
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="svg"
    doctype-public="-//W3c//DTD SVG 1.0//EN"
    doctype-system="http://w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"/>

  <!-- ============================================================  -->
  <!-- The "master" template, sets the viewbox and calls the other    -->
  <!-- templates that do the work.                                    -->
  <!-- ============================================================  -->
  <xsl:template match="globaldata">

    <xsl:variable name="rowNumber"
                  select="count(//*[@id='Phases and Activities']/data/entry)"/>

    <xsl:variable name="height" select="90 + $rowNumber * 65"/>
    <xsl:variable name="realHeight" select="$height * 15 div 800"/>
    <svg xml:space="preserve"
          xmlns="http://www.w3.org/2000/svg"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          width="20cm" height="{$realHeight}cm"
          viewBox="0 0 1200 {$height}"
          id="master">

      <!-- ==========================================================  -->
      <!-- ECMAScript, that opens a alert box with the begin and end   -->
      <!-- of a phase (activity).                                      -->
      <!-- ==========================================================  -->
      <script type="text/ecmascript"> <![CDATA[
        function mousemove( begin, end){
          var text = " begin: " + begin.toString()
                     + "; end: " + end.toString();
          alert(text);
        }
      ]]> </script>

      <xsl:call-template name="scale x-axis">
        <xsl:with-param name="call">
          <xsl:value-of select="1"/>
        </xsl:with-param>
      </xsl:call-template>
      <xsl:call-template name="title"/>
      <xsl:call-template name="legend"/>
      <xsl:call-template name="label y axis"/>
      <xsl:call-template name="print axis"/>
      <xsl:call-template name="print unit axis"/>
      <xsl:call-template name="print actual curve"/>
      <xsl:call-template name="print reference curve"/>
    </svg>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- The template prints the axis.                                  -->
  <!-- ============================================================  -->
  <xsl:template name="print axis">
    <line x1="200" x2="1000" y1="50" y2="50"
          style="stroke-width:2 ; stroke: black;"/>
    <xsl:variable name="maxY"
       select="count(//*[@id='Phases and Activities']/data/entry)"/>
    <xsl:variable name="y"
                  select="($maxY) * 65 + 60 "/>
    <line x1="200" x2="200" y1="50" y2="{$y}"
          style="stroke-width:2 ; stroke: black;"/>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- The template prints the unit of each axis.                     -->
  <!-- ============================================================  -->
  <xsl:template name="print unit axis">
    <text x="1010" y="54"
          style="font-size:14pt;">
      <xsl:value-of select="//parameter[@id='x-Axis Title']/@value"/>
    </text>
    <xsl:variable name="maxY"
       select="count(//*[@id='Phases and Activities']/data/entry)"/>
    <xsl:variable name="y"
                  select="($maxY) * 65 + 80 "/>
```

```
  <text x="200" y="{$y}"
        style="text-anchor: middle; font-size:14pt;">
    <xsl:value-of select="//parameter[@id='y-Axis Title']/@value"/>
  </text>
</xsl:template>

<!-- ============================================================  -->
<!-- The template scales the x-axis.                               -->
<!-- ============================================================  -->
<xsl:template name="scale x-axis">
  <xsl:param name="call"/>

  <xsl:variable name="max"
         select="//*/timeseries[last()]/entry[last()]/value"/>
  <xsl:variable name="x"
                select="(($call) div 10)  * 800 + 200"/>
  <line x1="{$x}" x2="{$x}" y1="45" y2="55"
        style="stroke-width:2 ; stroke: black;"/>
  <text x="{$x}" y="45"
        style="text-anchor: middle; font-size:14pt;">
    <xsl:value-of select="($call) div 10 * ($max)"/>
  </text>

  <xsl:variable name="maxY"
     select="count(//*[@id='Phases and Activities']/data/entry)"/>
  <xsl:variable name="y"
                select="($maxY) * 65 + 80 "/>
  <line x1="{$x}" x2="{$x}" y1="50" y2="{$y}"
        style="stroke-width:2 ; stroke: lightgrey;"/>
  <xsl:if test="($call) &lt; 10">
    <xsl:call-template name="scale x-axis">
      <xsl:with-param name="call">
        <xsl:value-of select="($call) + 1"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:if>
</xsl:template>

<!-- ============================================================  -->
<!-- The template prints the title of the diagramm and the informa-  -->
<!-- tion how to get further information about an entry.           -->
<!-- ============================================================  -->
<xsl:template name="title">
  <text x="600" y="-20" style="text-anchor: middle; font-size:18pt;">
    <xsl:value-of select="parameter[@id='Title']/@value"/>
  </text>
  <text x="1030" y="265"
        style="font-size:14pt;">Click on a line</text>
  <text x="1030" y="290"
        style="font-size:14pt;">for details.</text>
</xsl:template>

<!-- ============================================================  -->
<!-- The template prints the lines for the actual curve.            -->
<!-- ============================================================  -->
<xsl:template name="print actual curve">
  <xsl:variable name="maxX"
                select="//*[@id='Actual Plan']/data/timeseries[last()]/
                        entry[last()]/value"/>
   <xsl:for-each select="//*[@id='Actual Plan']/data/timeseries">
    <xsl:variable name="y"
              select="position() * 65 + 30 "/>
    <xsl:variable name="count"
              select="position()"/>

    <xsl:for-each select="./entry">
      <xsl:variable name="x1"
                    select="ts div ($maxX) * 800 + 200"/>
      <xsl:variable name="x2"
                    select="value div ($maxX) * 800 + 200"/>
      <xsl:variable name="y1"
                    select="($y) - 3"/>
      <xsl:variable name="y2"
                    select="($y) + 3"/>

      <xsl:variable name="yt"
                    select="($y) - 5"/>
      <xsl:variable name="xt"
                    select="(($x1) + ($x2)) div 2"/>
      <xsl:variable name="mouseover"
                    select="concat('mousemove(',
                                   ts,',',
                                   value,')')"/>

      <a><line onclick="{$mouseover}"
            x1="{$x1}" x2="{$x2}" y1="{$y}" y2="{$y}"
            style="stroke-width:4 ; stroke: blue;"/></a>
      <line x1="{$x1}" x2="{$x1}" y1="{$y1}" y2="{$y2}"
            style="stroke-width:3 ; stroke: blue;"/>
      <line x1="{$x2}" x2="{$x2}" y1="{$y1}" y2="{$y2}"
```

```
            style="stroke-width:3 ; stroke: blue;"/>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template prints the lines for the reference curve.         -->
<!-- ============================================================   -->
<xsl:template name="print reference curve">
  <xsl:variable name="maxX"
                select="//*[@id='Reference Plan']/data/timeseries[last()]/
                        entry[last()]/value"/>
  <xsl:for-each select="//*[@id='Reference Plan']/data/timeseries">
    <xsl:variable name="y"
              select="position() * 65 + 50 "/>

    <xsl:variable name="count"
              select="position()"/>
    <xsl:for-each select="./entry">
      <xsl:variable name="x1"
                    select="ts div ($maxX) * 800 + 200"/>
      <xsl:variable name="x2"
                    select="value div ($maxX) * 800 + 200"/>
      <xsl:variable name="y1"
                    select="($y) - 3"/>
      <xsl:variable name="y2"
                    select="($y) + 3"/>

      <xsl:variable name="mouseover"
                    select="concat('mousemove(',
                                   ts,',',
                                   value,')')"/>

      <a><line onclick="{$mouseover}"
            x1="{$x1}" x2="{$x2}" y1="{$y}" y2="{$y}"
            style="stroke-width:4 ; stroke: red;"/></a>
      <line x1="{$x1}" x2="{$x1}" y1="{$y1}" y2="{$y2}"
            style="stroke-width:3 ; stroke: red;"/>
      <line x1="{$x2}" x2="{$x2}" y1="{$y1}" y2="{$y2}"
            style="stroke-width:3 ; stroke: red;"/>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- The template labels the y-axis with the names of the phases     -->
<!-- and activities.                                                 -->
<!-- ============================================================   -->
<xsl:template name="label y axis">
  <xsl:for-each select="//*[@id='Phases and Activities']/data/entry">
    <xsl:variable name="y"
              select="position() * 65 + 30 "/>

    <xsl:call-template name="realize wordwrap">
      <xsl:with-param name="text">
        <xsl:value-of select="category"/>
      </xsl:with-param>
      <xsl:with-param name="y">
        <xsl:value-of select="($y)"/>
      </xsl:with-param>
    </xsl:call-template>
    <xsl:if test="((position() mod 3) = 0) and
                  (not (position() = last()))">
      <xsl:variable name="y"
                select="position() * 65 + 72.25 "/>
      <line x1="50" x2="1000" y1="{$y}" y2="{$y}"
            style="stroke-width:3 ; stroke: lightgray"/>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

<!-- ============================================================   -->
<!-- Because svg does not support a wordwrap, this is simulated in   -->
<!-- this template. First it is tested if the text is shorter than   -->
<!-- 22 characters and than printed, otherwise it is search for the  -->
<!-- second blank, if the text till there is shorter than 22         -->
<!-- characters it is printed, otherwise the text is printed until   -->
<!-- the first blank occures. If nor blank is found the text is      -->
<!-- printed.                                                        -->
<!-- In a later version it is possible to search for a "-" and use   -->
<!-- this additionally.                                              -->
<!-- ============================================================   -->
<xsl:template name="realize wordwrap">
  <xsl:param name="text"/>
  <xsl:param name="y"/>
  <xsl:choose>
    <xsl:when test="string-length(($text)) &lt; 22">
      <xsl:call-template name="write text">
        <xsl:with-param name="text">
          <xsl:value-of
```

```
          select="($text)"/>
    </xsl:with-param>
    <xsl:with-param name="y">
      <xsl:value-of select="($y)"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:when>
<xsl:when test="contains(($text),' ')">
  <xsl:variable name="texthelp"
               select="substring-after(($text),' ')"/>
  <xsl:choose>
    <xsl:when test="contains(($texthelp),' ')">
      <xsl:variable name="textend"
                    select="substring-after(($texthelp),' ')"/>
      <xsl:choose>
        <xsl:when test="string-length(
                       substring-before( ($text),($textend) )) &lt; 22">

          <xsl:call-template name="write text">
            <xsl:with-param name="text">
              <xsl:value-of
                   select="normalize-space(
                               substring-before( ($text),($textend) ))"/>
            </xsl:with-param>
            <xsl:with-param name="y">
              <xsl:value-of select="($y)"/>
            </xsl:with-param>
          </xsl:call-template>

          <xsl:if test="string-length(($textend)) &gt; 0">
            <xsl:call-template name="realize wordwrap">
              <xsl:with-param name="text">
                <xsl:value-of select="($textend)"/>
              </xsl:with-param>
              <xsl:with-param name="y">
                <xsl:value-of select="($y) + 20"/>
              </xsl:with-param>
            </xsl:call-template>
          </xsl:if>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="write text">
            <xsl:with-param name="text">
              <xsl:value-of select="substring-before(($text),' ')"/>
            </xsl:with-param>
            <xsl:with-param name="y">
              <xsl:value-of select="($y)"/>
            </xsl:with-param>
          </xsl:call-template>

          <xsl:if test="string-length(($texthelp)) &gt; 0">
            <xsl:call-template name="realize wordwrap">
              <xsl:with-param name="text">
                <xsl:value-of select="($texthelp)"/>
              </xsl:with-param>
              <xsl:with-param name="y">
                <xsl:value-of select="($y) + 20"/>
              </xsl:with-param>
            </xsl:call-template>
          </xsl:if>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise>

      <xsl:call-template name="write text">
        <xsl:with-param name="text">
          <xsl:value-of select="substring-before(($text),' ')"/>
        </xsl:with-param>
        <xsl:with-param name="y">
          <xsl:value-of select="($y)"/>
        </xsl:with-param>
      </xsl:call-template>

      <xsl:if test="string-length(($texthelp)) &gt; 0">
        <xsl:call-template name="realize wordwrap">
          <xsl:with-param name="text">
            <xsl:value-of select="($texthelp)"/>
          </xsl:with-param>
          <xsl:with-param name="y">
            <xsl:value-of select="($y) + 20"/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>
    </xsl:otherwise>
  </xsl:choose>

</xsl:when>
<xsl:otherwise>
  <xsl:call-template name="write text">
```

```
          <xsl:with-param name="text">
            <xsl:value-of select="($text)"/>
          </xsl:with-param>
          <xsl:with-param name="y">
            <xsl:value-of select="($y)"/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- This template writes a given text at a given height along the  -->
  <!-- y-axis.                                                        -->
  <!-- ============================================================  -->
  <xsl:template name="write text">
    <xsl:param name="text"/>
    <xsl:param name="y"/>
    <text x="190"  y="{$y}"
          style="text-anchor:end; font-size:12pt;">
      <xsl:value-of select="($text)"/>
    </text>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- The template makes the legend.                                 -->
  <!-- ============================================================  -->
  <xsl:template name="legend">
    <text x="1030" y="130"
        style="font-size:14pt; fill:blue">actual curve</text>
    <text x="1030" y="160"
        style="font-size:14pt; fill:red">reference curve</text>
  </xsl:template>
</xsl:stylesheet>
```

B.5 Kreisdiagramm (*Pie Chart*)

```
<!-- ============================================================  -->
<!-- Prints a pie chart and needs a CategorySeries as input. The color -->
<!-- of each section is calculated automatically, but with more than  -->
<!-- 16 entries the algorithm does not work and the colors of the last -->
<!-- sections are equal. For more sections an alternativ algorithm is  -->
<!-- needed or probably an additional input that specifies the color.  -->
<!-- The color is given in RGB, the red part is decreased by 16 for    -->
<!-- each section. The blue part is increased by 16 for each section.  -->
<!-- The green part is calculated with a alternating function that     -->
<!-- multiplies (-1)^<position> with eight and <position - 1> and adds -->
<!-- it to 225. Position specifies the call number (or the number of   -->
<!-- the entry that is processed; are equal) of the template.          -->
<!-- ============================================================  -->
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="svg"
    doctype-public="-//W3c//DTD SVG 1.0//EN"
    doctype-system="http://w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"/>

  <!-- ============================================================  -->
  <!-- The "master" template, sets the viewbox and calls the other    -->
  <!-- templates that do the work.                                    -->
  <!-- ============================================================  -->
  <xsl:template match="globaldata">

    <svg xml:space="preserve"
      xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      viewBox="0 0 800 700">

      <xsl:call-template name="title"/>
      <xsl:call-template name="print pieChart"/>
      <xsl:call-template name="make legend"/>
    </svg>
  </xsl:template>

  <!-- ============================================================  -->
  <!-- The first call of the template pieSection, with the first      -->
  <!-- parameters.                                                    -->
  <!-- ============================================================  -->
  <xsl:template name="print pieChart">
      <xsl:variable name="totalsum" select="sum(.//entry/value/)"/>

      <xsl:call-template name="pieSection">
        <xsl:with-param name="x">
          <xsl:value-of select="400"/>
        </xsl:with-param>
        <xsl:with-param name="y">
```

```
        <xsl:value-of select="350"/>
      </xsl:with-param>
      <xsl:with-param name="mx">
        <xsl:value-of select="200"/>
      </xsl:with-param>
      <xsl:with-param name="my">
        <xsl:value-of select="350"/>
      </xsl:with-param>
      <xsl:with-param name="radius">
        <xsl:value-of select="200"/>
      </xsl:with-param>
      <xsl:with-param name="total">
        <xsl:value-of select="$totalsum"/>
      </xsl:with-param>
      <xsl:with-param name="fraction">
        <xsl:value-of select="0"/>
      </xsl:with-param>
      <xsl:with-param name="colorR">
        <xsl:value-of select="255"/>
      </xsl:with-param>
      <xsl:with-param name="colorG">
        <xsl:value-of select="255"/>
      </xsl:with-param>
      <xsl:with-param name="colorB">
        <xsl:value-of select="0"/>
      </xsl:with-param>
      <xsl:with-param name="callNumber">
        <xsl:value-of select="1"/>
      </xsl:with-param>
    </xsl:call-template>
</xsl:template>

<!-- ============================================================   -->
<!-- Prints one section of the pie chart. If there are unprinted    -->
<!-- sections, the template calls itself with modified parameters.  -->
<!-- ============================================================   -->
<xsl:template name="pieSection">
  <xsl:param name="x"/>
  <xsl:param name="y"/>
  <xsl:param name="mx"/>
  <xsl:param name="my"/>
  <xsl:param name="radius"/>
  <xsl:param name="total"/>
  <xsl:param name="fraction"/>
  <xsl:param name="colorR"/>
  <xsl:param name="colorG"/>
  <xsl:param name="colorB"/>
  <xsl:param name="callNumber"/>

  <xsl:variable name="actualPart"
            select="//data/entry[position() = ($callNumber)]/value"/>
  <xsl:variable name="fraction"
                select="($fraction) + ($actualPart)"/>

  <!-- Calculates the coordinates -->
  <xsl:variable name="x1"
                select="math1:cos(($fraction) div $total * 2 * 3.1415926)"
                xmlns:math1="java.lang.Math"/>
  <xsl:variable name="x1"
                select="($x1) * ($radius)"/>
  <xsl:variable name="x1"
                select="($x1) + ($mx)"/>
  <xsl:variable name="y1"
                select="math1:sin(($fraction) div $total * 2 * 3.1415926)"
                xmlns:math1="java.lang.Math"/>
  <xsl:variable name="y1"
                select="($y1) * ($radius)"/>
  <xsl:variable name="y1"
                select="($y1) + ($my)"/>

  <!-- The color of the pie section   -->
  <xsl:variable name="command"
                select="concat('rgb(', ($colorR),
                               ',', ($colorG),
                               ', ',($colorB),')' )"/>

  <xsl:choose>
    <xsl:when test="(($actualPart) div ($total)) &lt; 0.5">
      <xsl:if test="($actualPart) &gt; 0">
        <path d="M{$x},{$y} A{$radius}, {$radius}
                 0 0 1 {$x1},{$y1} L{$mx},{$my} z"
              fill="{$command}" stroke="black" stroke-width="1" />
      </xsl:if>
    </xsl:when>
    <xsl:otherwise>
      <path d="M{$x},{$y} A{$radius}, {$radius}
                0 1 1 {$x1},{$y1} L{$mx},{$my} z"
             fill="{$command}" stroke="black" stroke-width="1" />
    </xsl:otherwise>
  </xsl:choose>
```

```
  <!-- Calls the template again, if there are nodes left -->
  <xsl:variable name="nodePosition" select="$callNumber + 1"/>
  <xsl:if test="($fraction) &lt; ($total)">
    <xsl:call-template name="pieSection">
      <xsl:with-param name="x">
        <xsl:value-of select="($x1)"/>
      </xsl:with-param>
      <xsl:with-param name="y">
        <xsl:value-of select="($y1)"/>
      </xsl:with-param>
      <xsl:with-param name="mx">
        <xsl:value-of select="($mx)"/>
      </xsl:with-param>
      <xsl:with-param name="my">
        <xsl:value-of select="($my)"/>
      </xsl:with-param>
      <xsl:with-param name="radius">
        <xsl:value-of select="($radius)"/>
      </xsl:with-param>
      <xsl:with-param name="total">
        <xsl:value-of select="($total)"/>
      </xsl:with-param>
      <xsl:with-param name="fraction">
        <xsl:value-of select="($fraction)"/>
      </xsl:with-param>
      <xsl:with-param name="colorR">
        <xsl:value-of select="($colorR) - 16"/>
      </xsl:with-param>
      <xsl:with-param name="colorG">
        <xsl:value-of select="((($colorG) + 1) mod 2) * 255  +
                (1 + (-2) * ((($colorG) +1) mod 2)) * ($callNumber) * 8"/>
      </xsl:with-param>
      <xsl:with-param name="colorB">
        <xsl:value-of select="($colorB) + 16"/>
      </xsl:with-param>
      <xsl:with-param name="callNumber">
        <xsl:value-of select="($callNumber) + 1"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:if>
</xsl:template>

<!-- ============================================================   -->
<!-- Prints the title of the diagramm.                              -->
<!-- ============================================================   -->
<xsl:template name="title">
  <text x="270" y="30" style="text-anchor: middle; font-size:14pt;">
    <xsl:value-of select="parameter[@id='Title']/@value"/>
  </text>
</xsl:template>

<!-- ============================================================   -->
<!-- The template makes the legend of the diagramm.                 -->
<!-- ============================================================   -->
<xsl:template name="make legend">
  <xsl:variable name="total" select="sum(.//entry/value/)"/>
  <text x="548" y="125" style="font-size:14pt;">
    total: <xsl:value-of select="$total"/>
  </text>
  <xsl:for-each select=".//entry/">
    <xsl:variable name="textX"
                  select="580"/>
    <xsl:variable name="textY"
                  select="position() * 35 + 125"/>
    <xsl:variable name="lineY"
                  select="position() * 35 + 120"/>
    <xsl:variable name="colorG"
                  select="(255 - (position() -1) * 8) *
                          (position() mod 2)
                          + ((position() - 1) * 8 ) *
                          ((position() + 1) mod 2)"/>
    <xsl:variable name="colorR"
                  select="255 - (position() - 1) * 16"/>
    <xsl:variable name="colorB"
                  select="0 + (position()- 1)* 16"/>
    <xsl:variable name="color"
                  select="concat('rgb(',
                                 $colorR, ',',
                                 $colorG,', ',
                                 $colorB,')' )"/>
    <line x1="540" y1="{$lineY}" x2="575" y2="{$lineY}"
          style="stroke-width:10; stroke:{$color};"/>

    <xsl:variable name="percent"
                  select="./value/"/>
    <xsl:variable name="percent"
                  select="$percent div $total * 100"/>
    <xsl:variable name="percent"
                  select="round($percent * 100) div 100"/>
```

```
      <text x="{$textX}" y="{$textY}"
            style="font-size:14pt;">
        <xsl:value-of
             select="./category/"/>: <xsl:value-of select="$percent"/>%
      </text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

C Implementierte Funktionen

Hier finden sich der Kode der Java-Klassen, die neu implementiert wurden, um die Messdaten geeignet aufzubereiten.

C.1 Convert Project Plan 2 List of Time Series

```
// Software Project Control Center
// (c) Kaiserslautern University of Technology

package de.ktech.spcc.server.functions.repository;

import de.ktech.spcc.server.dataTypes.repository.*;
import de.ktech.spcc.server.functions.*;
import de.ktech.spcc.server.functions.exceptions.*;

import java.util.*;

/**
 * Converts the DataTypeImplementation Object ProjectPlan to a
 * ListOfTimeSeries. Additionally a CategorySeries is produced containing
 * the names of the phases and activities for that an entry exists.
 *
 * @author Irina Feuerbach
 */
public class ConvertProjectPlan2ListOfTimeSeries
               extends FunctionImplementation {
  private static String input1 = "Project Plan";
  private static String input2 = "Phases and Activities";
private static String output1 = "List of Time Series";
  private static String output2 = "Entries";

  /**
   * @see de.ktech.spcc.server.functions.FunctionImplementation#compute()
   */
  public void compute() throws ComputingException {
ProjectPlan plan =
(ProjectPlan) this.getInput(input1);
PhasesAndActivities allPhases =
(PhasesAndActivities) this.getInput(input2);

if ((plan == null) || (allPhases == null))
throw new ComputingException("Some entries null");
    ListOfTimeSeries projectPlan = new ListOfTimeSeries();
    CategorySeries entries = new CategorySeries();
    TimeSeries ts;
    Vector phases = allPhases.getPhases();
    Vector activities;
    String phase, activity;
    Vector temp;
    int position;
    for (int i = 0; i < phases.size(); i++){
      phase = (String) phases.get(i);
      ts = null;
      temp = new Vector();
      // search all Entries of the phase
      for (int j = 0; j < plan.getNumberOfProjectPlanEntries(); j++){
        if ((plan.getPhase(j)).equals(phase)){
          // if the entry corresponds to the phase, store it in the time
          // series, otherwise add it to a vector and process it later
          if (plan.getActivity(j) == null){
            if (ts == null)
              ts = new TimeSeries();
            ts.set(plan.getBegin(j), plan.getEnd(j));
          }
          else
            temp.add(new Integer(j));
        }
      }
      if (ts != null){
        projectPlan.add(ts);
        ts = null;
        entries.add(phase,0);
      }

      activities = allPhases.getActivities(phase);
      for (int j = 0; j < activities.size(); j++){
        activity = (String) activities.get(j);
        // search all entries of the activity,
        for (int k = 0; k < temp.size(); k ++){
          position = ((Integer) temp.get(k)).intValue();
          if ((plan.getActivity(position)).equals(activity)){
            if (ts == null)
```

```
            ts = new TimeSeries();
          ts.set(plan.getBegin(position), plan.getEnd(position));
        }
      }

      if (ts != null){
        projectPlan.add(ts);
        ts = null;
        entries.add(activity,0);
      }

    }
  }
  this.setOutput(output1, projectPlan);
  this.setOutput(output2, entries);
 }

}
```

C.2 Defect Processing

```
// Software Project Control Center
// (c) Kaiserslautern University of Technology

package de.ktech.spcc.server.functions.repository;

import de.ktech.spcc.server.dataTypes.repository.*;
import de.ktech.spcc.server.functions.*;
import de.ktech.spcc.server.functions.exceptions.*;

import java.util.*;

/**
 * This class processes all found defects dependent on the parameter.
 * It is possible to sort the defects according to their state or their class.
 * The result is of the type CategorySeries with the name of the state (or
 * the class) and the corresponding number of defects.
 *
 * @author Irina Feuerbach
 * @author Stefan Knieling
 */
public class DefectsProcessing extends FunctionImplementation {
  private static String input1 = "Reported Defects";
  private static String input2 = "State";
  private static String input3 = "Class";
  private static String parameter = "Sorting Criteria";
  private static String outputID = "Sorted Defects";

  /**
   * @see de.ktech.spcc.server.functions.FunctionImplementation#compute()
   */
  public void compute() throws ComputingException {
    CategorySeries sortedDefects = new CategorySeries();
    String param = this.getParameter(parameter);
    if (param.equals("state"))
      sortedDefects = sortAccordingStates();
    else if (param.equals("class"))
      sortedDefects = sortAccordingClasses();
```

```
    this.setOutput(outputID, sortedDefects);
  }

  /** Sorts the defects according to their class and puts the result into a
   *  CategorySeries. The value corresponds to the total number of
   *  defects with this class.
   * @exception ComputingException.
   * @return CategorySeries. The series containing the sorted values.
   */
  private CategorySeries sortAccordingClasses() throws ComputingException {
    // retrieve values from wrapper
    NumberOfFoundDefects allFoundDefects =
              (NumberOfFoundDefects) this.getInput(input1);

 Categories categories = (Categories) this.getInput(input3);
   CategorySeries sortedDefects = new CategorySeries();

    if ((allFoundDefects == null) ||
        (categories == null)) {
      throw new ComputingException(
        "NullPointerException, an input wasn't set or the name " +
"of the input was spelled incorrect");
    }

Vector classes = categories.getEntries();
    int count = 0;
    String id = "";
    String title = "";

    /* Here we need to compare by id because it is the id that is
     * given in the corresponding xml-file and not the class itself.
     */

    for (int i = 0; i < classes.size(); i++) {
      count = 0;
      id = ( (Categories.Entry) classes.elementAt(i) ).getId();
      title = ( (Categories.Entry) classes.elementAt(i) ).getTitle();
      for (int j = 0; j < allFoundDefects.getNumberOfFoundDefect(); j++) {
       if (id.equals(allFoundDefects.getDefectClass(j))){
              count++;
        }
      }
      sortedDefects.add(title, count);
    }
return sortedDefects;
  }

  /** Sorts the defects according to their state and puts the result into a
   *  CategorySeries. The value corresponds to the total number of
   *  defects with this state.
   * @exception ComputingException.
   * @return CategorySeries. The series containing the sorted values.
   */
  private CategorySeries sortAccordingStates() throws ComputingException{
    // retrieve values from wrapper
    NumberOfFoundDefects allFoundDefects =
              (NumberOfFoundDefects) this.getInput(input1);
Categories categories = (Categories) this.getInput(input2);
    CategorySeries sortedDefects = new CategorySeries();

    if ((allFoundDefects == null) ||
        (categories == null)) {
      throw new ComputingException(
        "NullPointerException, an input wasn't set or the name " +
"of the input was spelled incorrect");
    }

Vector states = categories.getEntries();
    int count = 0;
    String title = "";

    for (int i = 0; i < states.size(); i++) {
      count = 0;
  title = ( (Categories.Entry) states.elementAt(i) ).getTitle();
      for (int j = 0; j < allFoundDefects.getNumberOfFoundDefect(); j++) {
        if ( title.equals(allFoundDefects.getStatus(j)) ){
          count++;
        }
      }
      sortedDefects.add(title, count);
    }
    return sortedDefects;
  }
}
```

C.3 Defect Sorting (Affiliation and Phase)

```
// Software Project Control Center
// (c) Kaiserslautern University of Technology

package de.ktech.spcc.server.functions.repository;

import de.ktech.spcc.server.dataTypes.repository.*;
import de.ktech.spcc.server.functions.*;
import de.ktech.spcc.server.functions.exceptions.*;

import java.util.*;

/**
 * This function sorts all found defects according to the phases and defect
 * affiliations. The effort is added if there is more than one defect in the
 * same phase and affiliation and the number of defects is counted. The
 * result is of the type Defects.
 *
 * @author Irina Feuerbach
 * @author Stefan Knieling
 */
public class DefectSortingAccordingAffiliationAndPhase
extends FunctionImplementation {
  private static String input1 = "Reported Defects";
  private static String input2 = "Defect Affiliation";
  private static String input3 = "Phases";
  private static String outputID = "Sorted Defects";

  /**
   * @see de.ktech.spcc.server.functions.FunctionImplementation#compute()
   */
  public void compute() throws ComputingException {
    // retrieve values from wrapper
    NumberOfFoundDefects allFoundDefects =
     (NumberOfFoundDefects) this.getInput(input1);
    Categories categories = (Categories) this.getInput(input2);
    PhasesAndActivities allPhases =
     (PhasesAndActivities) this.getInput(input3);
    Defects allDefects = new Defects();

    if ((allFoundDefects == null) ||
        (allPhases == null) ||
        (categories == null)) {
      throw new ComputingException(
        "NullPointerException, an input wasn't set or the name " +
" of the input was spelled incorrect");
    }

    Vector phase = allPhases.getPhases();
Vector affiliations = categories.getEntries();
int count = 0;
double effort = 0;
String title = "";

for (int i = 0; i < phase.size(); i++) {
for (int j = 0; j < affiliations.size(); j++) {
        count = 0;
        effort = 0;
        title = ( (Categories.Entry) affiliations.elementAt(j) ).getTitle();

        for (int k = 0; k < allFoundDefects.getNumberOfFoundDefect(); k++) {
          if ((((String) phase.get(i)).equals(allFoundDefects.getPhase(k))) &&
              (title.equals(allFoundDefects.getAffiliation(k)))) {
         count++;
             effort += allFoundDefects.getEffort(k);
          }
        }
        allDefects.newEntry((String) phase.get(i), count, title, effort);
    }
}
this.setOutput(outputID, allDefects);
  }
}
```

C.4 Effort Processing

```
// Software Project Control Center
// (c) Kaiserslautern University of Technology

package de.ktech.spcc.server.functions.repository;

import de.ktech.spcc.server.dataTypes.repository.*;
import de.ktech.spcc.server.functions.*;
import de.ktech.spcc.server.functions.exceptions.*;

import java.util.*;

/**
 * The class converts the DataTypeImplementation Object Effort to a
 * ListOfTimeSeries or a TimeSeries. With a parameter it is possible to
 * differ between the outputs. The other outputs are null.
 * The function could convert the effort of the phases into a timeseries or
 * additionally the effort of the activities. In this case a factor is needed,
 * that helps to decode the timestamp in order to differ between effort values
 * of phases and effort values of activities.
 *
 * @author Irina Feuerbach
 */
public class EffortProcessing extends FunctionImplementation {
private static String input1 = "Effort";
private static String input2 = "Phases and Activities";
private static String output1 = "Time Series (only Phases)";
private static String output2 = "List of Time Series";
private static String output3 = "Time Series";
private static String parameter1 = "Output Type";
private static String parameter2 = "Phases Factor";

/**
 * @see de.ktech.spcc.server.functions.FunctionImplementation#compute()
 */
public void compute() throws ComputingException {
Effort effort =
(Effort) this.getInput(input1);
PhasesAndActivities allPhases =
(PhasesAndActivities) this.getInput(input2);

if ((effort == null) || (allPhases == null))
throw new ComputingException("Some entries null");
String param = this.getParameter(parameter1);
String factor = (String) this.getParameter(parameter2);
int phaseFactor;
try {
phaseFactor = Integer.parseInt(factor);
} catch (NumberFormatException nfe) {
throw new ComputingException("Error reading parameter: Phases factor");
}

if (param.equals("phasesAsTS"))
convertPhaseEffort2TimeSeries(effort, allPhases);
if (param.equals("asLoTS"))
convertEffort2ListOfTimeSeries(effort, allPhases);
if (param.equals("asTS"))
convertEffort2TimeSeries(effort, allPhases, phaseFactor);
}

private void convertPhaseEffort2TimeSeries(
Effort effort,
PhasesAndActivities allPhases) {
TimeSeries phaseEffortAsTS = new TimeSeries();
Vector phasesNames = allPhases.getPhases();
String phase;
for (int i = 0; i < phasesNames.size(); i++) {
phase = (String) phasesNames.get(i);

if (effort.getEffort(phase) != -1)
phaseEffortAsTS.set(
allPhases.getPhaseNumber(phase),
effort.getEffort(phase));
}

this.setOutput(output1, phaseEffortAsTS);
this.setOutput(output2, null);
this.setOutput(output3, null);
}

private void convertEffort2ListOfTimeSeries(
     Effort effort,
     PhasesAndActivities allPhases) {
ListOfTimeSeries effortAsLoTS = new ListOfTimeSeries();
Vector phasesNames = allPhases.getPhases();
String phase;
Vector activities;
    TimeSeries ts;
```

```
// Creates the time series containing the effort value of
// the activities
double tmpEffort;
for (int i = 0; i < phasesNames.size(); i++) {
ts = new TimeSeries();
phase = (String) phasesNames.get(i);
activities = allPhases.getActivities(phase);
for (int j = 0; j < activities.size(); j++) {
tmpEffort = effort.getEffort(phase, (String) activities.get(j));
if (tmpEffort != -1)
ts.set(j, tmpEffort);
}
effortAsLoTS.add(ts);
}

this.setOutput(output1, null);
this.setOutput(output2, effortAsLoTS);
this.setOutput(output3, null);
}

private void convertEffort2TimeSeries (
Effort effort,PhasesAndActivities allPhases,
int phaseFactor) {
TimeSeries effortAsTS = new TimeSeries();
Vector phasesNames = allPhases.getPhases();
String phase;
Vector activities;
int timestamp;
double tmpEffort;
for (int i = 0; i < phasesNames.size(); i++) {
phase = (String) phasesNames.get(i);

if (effort.getEffort(phase) != -1)
effortAsTS.set(
allPhases.getPhaseNumber(phase) * phaseFactor,
effort.getEffort(phase));
activities = allPhases.getActivities(phase);
for (int j = 0; j < activities.size(); j++) {
tmpEffort = effort.getEffort(phase, (String) activities.get(j));
if (tmpEffort != -1) {
timestamp = allPhases.getPhaseNumber(phase) * phaseFactor;
timestamp = timestamp + j + 1;
effortAsTS.set(timestamp, tmpEffort);
}
}
}

this.setOutput(output1, null);
this.setOutput(output2, null);
this.setOutput(output3, effortAsTS);
}
}
```

C.5 Reported Effort Processing

```
// Software Project Control Center
// (c) Kaiserslautern University of Technology

package de.ktech.spcc.server.functions.repository;

import de.ktech.spcc.server.dataTypes.repository.*;
import de.ktech.spcc.server.functions.*;
import de.ktech.spcc.server.functions.exceptions.*;

import java.util.*;

/**
 * The class converts the DataTypeImplementation Object ReportedEffort to a
 * ListOfTimeSeries or a TimeSeries. With a parameter it is possible to
 * differ between the outputs. The other outputs are null.
 * The function could convert the effort of the phases into a timeseries or
 * additionally the effort of the activities. In this case a factor is needed,
 * that helps to decode the timestamp in order to differ between effort values
 * of phases and effort values of activities.
 *
 * @author Irina Feuerbach
 */
public class ReportedEffortProcessing extends FunctionImplementation {
  private static String input1 = "Reported Effort";
  private static String input2 = "Phases and Activities";
  private static String parameter1 = "Maximum Phase Number";
  private static String parameter2 = "Output Type";
  private static String parameter3 = "Phases factor";
  private static String output1 = "Time Series (only Phases)";
  private static String output2 = "List of Time Series";
  private static String output3 = "Time Series";
  private static String output4 = "Maxima";
  private static String output5 = "Effort Assessment";
  private static String output6 = "Difficulty";

  /**
   * @see de.ktech.spcc.server.functions.FunctionImplementation#compute()
   */
  public void compute() throws ComputingException {
    ReportedEffort effort = (ReportedEffort) this.getInput(input1);
    PhasesAndActivities allPhases = (PhasesAndActivities) this.getInput(input2);

    if ((effort == null) || (allPhases == null))
      throw new ComputingException("Some entries null");

    String value = this.getParameter(parameter1);
    String param = this.getParameter(parameter2);
    int maxPhaseNumber = 0;

    try {
      maxPhaseNumber = Integer.parseInt(value);

      if (maxPhaseNumber < 0)
        maxPhaseNumber = Integer.MAX_VALUE;
    } catch (NumberFormatException nfe) {
      maxPhaseNumber = Integer.MAX_VALUE;
    }

    if (param.equals("phasesAsTS"))
      computeReportedEffortSumForPhases(effort, allPhases, maxPhaseNumber);

    if (param.equals("asLoTS"))
      computeReportedEffortSumAsListOfTimeSeries(effort, allPhases,
        maxPhaseNumber);

    if (param.equals("asTS"))
      computeReportedEffortSumForPhasesAndActivitiesAsTimeSeries(effort,
        allPhases, maxPhaseNumber);

    computeMeanEffortAssessment(effort, allPhases, maxPhaseNumber);
    computeMeanDifficulty(effort, allPhases, maxPhaseNumber);
  }

  private void computeReportedEffortSumForPhases(ReportedEffort effort,
    PhasesAndActivities allPhases, int maxPhaseNumber)
    throws ComputingException {
    TimeSeries effortTS = new TimeSeries();
    TimeSeries maximumTS = new TimeSeries();

    Vector phasesNames = allPhases.getPhases();
    String phase;
    int ts = 0;
    double sum;
    double max = 0;
    int endingPhaseNumber;

    if (phasesNames.size() < maxPhaseNumber)
      endingPhaseNumber = phasesNames.size();
```

```
  else
    endingPhaseNumber = maxPhaseNumber;

  for (int i = 0; i < endingPhaseNumber; i++) {
    sum = 0;
    phase = (String) phasesNames.get(i);
    ts = allPhases.getPhaseNumber(phase);

    for (int j = 0; j < effort.getNumberOfEntries(); j++) {
      if (phase.equals(effort.getPhase(j))) {
        sum += effort.getEffort(j);
      }
    }

    effortTS.set(ts++, sum);

    if (max < sum)
      max = sum;
  }

  maximumTS.set(0, max);
  this.setOutput(output1, effortTS);
  this.setOutput(output2, null);
  this.setOutput(output3, null);
  this.setOutput(output4, maximumTS);
}

private void computeReportedEffortSumForPhasesAndActivitiesAsTimeSeries(
  ReportedEffort effort,
  PhasesAndActivities allPhases,
  int maxPhaseNumber) throws ComputingException {
  TimeSeries effortTS = new TimeSeries();
  TimeSeries maximaTS = new TimeSeries();
  int phaseFactor;
  double maxPhases = 0;
  double maxActivities = 0;
  String factor = (String) this.getParameter(parameter3);

  try {
    phaseFactor = Integer.parseInt(factor);
  } catch (NumberFormatException nfe) {
    throw new ComputingException("Error reading parameter: Phases factor");
  }

  Vector phases = allPhases.getPhases();
  String phase;
  String activity;
  int ts = 0;
  int endingNumber;
  double sum;

  if (phases.size() < maxPhaseNumber)
    endingNumber = phases.size();
  else
    endingNumber = maxPhaseNumber;

  for (int i = 0; i < endingNumber; i++) {
    sum = 0;
    phase = (String) phases.get(i);
    ts = allPhases.getPhaseNumber(phase) * phaseFactor;

    //first the effort of the phase
    for (int j = 0; j < effort.getNumberOfEntries(); j++) {
      if (phase.equals(effort.getPhase(j))) {
        sum += effort.getEffort(j);
      }
    }

    if (maxPhases < sum)
      maxPhases = sum;

    effortTS.set(ts, sum);

    //second the effort of the activities
    Vector activityNames = allPhases.getActivities(phase);

    for (int j = 0; j < activityNames.size(); j++) {
      sum = 0;
      activity = (String) activityNames.get(j);
      ts = allPhases.getPhaseNumber(phase) * phaseFactor;
      ts = ts + j + 1;

      for (int k = 0; k < effort.getNumberOfEntries(); k++) {
        if (phase.equals(effort.getPhase(k)) &&
            activity.equals(effort.getActivity(k))) {
          sum += effort.getEffort(k);
        }
      }

      effortTS.set(ts, sum);
```

```
      if (maxActivities < sum)
        maxActivities = sum;
    }

    maximaTS.set(i + 1, maxActivities);
    maxActivities = 0;
  }

  maximaTS.set(0, maxPhases);
  this.setOutput(output1, null);
  this.setOutput(output2, null);
  this.setOutput(output3, effortTS);
  this.setOutput(output4, maximaTS);
}

private void computeReportedEffortSumAsListOfTimeSeries(
  ReportedEffort effort,
  PhasesAndActivities allPhases,
  int maxPhaseNumber) throws ComputingException {
  ListOfTimeSeries effortLoTS = new ListOfTimeSeries();
  TimeSeries maximaTS = new TimeSeries();
  TimeSeries ts;
  double maxActivities = 0;

  Vector phases = allPhases.getPhases();
  String phase;
  String activity;
  int endingNumber;
  double sum;

  if (phases.size() < maxPhaseNumber)
    endingNumber = phases.size();
  else
    endingNumber = maxPhaseNumber;

  for (int i = 0; i < endingNumber; i++) {
    sum = 0;
    phase = (String) phases.get(i);

    ts = new TimeSeries();

    Vector activityNames = allPhases.getActivities(phase);

    for (int j = 0; j < activityNames.size(); j++) {
      sum = 0;
      activity = (String) activityNames.get(j);

      for (int k = 0; k < effort.getNumberOfEntries(); k++) {
        if (phase.equals(effort.getPhase(k)) &&
            activity.equals(effort.getActivity(k))) {
          sum += effort.getEffort(k);
        }
      }

      ts.set(j, sum);

      if (maxActivities < sum)
        maxActivities = sum;
    }

    effortLoTS.add(ts);
    maximaTS.set(i, maxActivities);
    maxActivities = 0;
  }

  this.setOutput(output1, null);
  this.setOutput(output2, effortLoTS);
  this.setOutput(output3, null);
  this.setOutput(output4, maximaTS);
}

/**
 * Computes the mean of all effort assessment values per phase.
 *
 * @param effort The reported effort.
 * @param allPhases The phases and activities.
 * @param maxPhaseNumber The maximum number of phases.
 */
private void computeMeanEffortAssessment(ReportedEffort effort,
  PhasesAndActivities allPhases, int maxPhaseNumber) {
  String phase;
  TimeSeries timeSeries = new TimeSeries();
  double[] sum = new double[maxPhaseNumber];
  int[] number = new int[maxPhaseNumber];
  int max = allPhases.getPhases().size();

  if (max > maxPhaseNumber)
    max = maxPhaseNumber;
```

```
  // Compute mean:
  for (int j = 0; j < effort.getNumberOfEntries(); j++) {
    for (int i = 0; i < max; i++) {
      phase = (String) allPhases.getPhases().get(i);

      if (phase.equals(effort.getPhase(j))) {
        sum[i] += effort.getEffortAssessment(j);
        number[i]++;
      }
    }
  }

  // Set time series:
  for (int i = 0; i < max; i++) {
    if (number[i] > 0)
      timeSeries.set(i, sum[i] / (double) number[i]);
    else
      timeSeries.set(i, 0);
  }

  this.setOutput(output5, timeSeries);
}

/**
 * Computes the mean of all difficulty values per phase.
 *
 * @param effort The reported effort.
 * @param allPhases The phases and activities.
 * @param maxPhaseNumber The maximum number of phases.
 */
private void computeMeanDifficulty(ReportedEffort effort,
  PhasesAndActivities allPhases, int maxPhaseNumber) {
  String phase;
  TimeSeries timeSeries = new TimeSeries();
  double[] sum = new double[maxPhaseNumber];
  int[] number = new int[maxPhaseNumber];
  int max = allPhases.getPhases().size();

  if (max > maxPhaseNumber)
    max = maxPhaseNumber;

  // Compute mean:
  for (int j = 0; j < effort.getNumberOfEntries(); j++) {
    for (int i = 0; i < max; i++) {
      phase = (String) allPhases.getPhases().get(i);

      if (phase.equals(effort.getPhase(j))) {
        sum[i] += effort.getDifficulty(j);
        number[i]++;
      }
    }
  }

  // Set time series:
  for (int i = 0; i < max; i++) {
    if (number[i] > 0)
      timeSeries.set(i, sum[i] / (double) number[i]);
    else
      timeSeries.set(i, 0);
  }

  this.setOutput(output6, timeSeries);
}
}
```