Fachbereich 3
Mathematik - Informatik
Universität Bremen

# Software-Monitoring mit Hilfe eines Dashboards

Diplomarbeit am Fachbereich Mathematik und Informatik
Prof. Dr. R. Koschke
Arbeitsgruppe Softwaretechnik
Universität Bremen

von

cand. inform.
**Rebecca Tiarks**

Tag der Anmeldung: 9. November 2007
Tag der Abgabe: 9. Juli 2008

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bremen, den 09. Juli 2008

*Danksagung*

*Ich bedanke mich bei allen, die mich bei meiner Diplomarbeit unterstützt haben. In erster Linie gilt mein Dank Herrn Professor Koschke für die gute Betreuung und Zusammenarbeit während der Enstehung dieser Arbeit. Darüber hinaus danke ich den Mitarbeitern der Axivion GmbH und der gesamten AG Softwaretechnik. Im Besonderen möchte ich mich bei meiner Familie und Stefan bedanken, die mir mit jeglicher Unterstützung zur Seite standen und während der Diplomarbeit viel auf mich verzichten mussten.*

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

## 1.1 Hintergrund

Die Zahl der Softwaresysteme, die uns in unserem alltäglichen Leben umgeben, wird immer größer. Genauso wie Menschen älter werden, altert auch Software. Der Prozess der Alterung kann zwar nicht verhindert werden, aber es können Schritte unternommen werden, um verschiedene Auswirkungen einzuschränken und zu verlangsamen. Die Umsetzung dieser Maßnahmen ist das Ziel der Softwarewartung. Die Praxis zeigt, dass die meisten Ressourcen in der Softwareentwicklung nicht für die Erstellung von neuen Systemen verwendet werden sondern für die Wartung von bestehenden Programmen. Nach Boehm (1981) beträgt der Anteil der für die Wartung benötigten Ressourcen und aufgewendeten Zeit am gesamten Software-Lebens-Zyklus 50% bis 75%.

Die Qualität der Software hat dabei entscheidenden Einfluss auf die Wartung und Weiterentwicklung. Die Sicherstellung der Softwarequalität bietet eine vielversprechende Möglichkeit zur Verbesserung der Produktivität der Softwarewartung, was gleichzeitig eine Kosteneinsparung bedeutet. Da die Evolution der Software erwiesenermaßen zu einem Verfall der Qualität führt, wurden verschiedene Analysen entwickelt, um dem fortschreitenden Verfall entgegenzuwirken. Sie untersuchen ein System mit dem Ziel, Alterungserscheinungen sowohl auf Ebene des Quelltextes als auch auf Ebene der Architektur aufzudecken.

Die Ergebnisse der Analysen können als Qualitätparameter für ein Softwaresystem dienen, jedoch ist auch eine kontinuierliche Überwachung dieser Qualitätsattribute notwendig. Da aufgrund der Größe und Komplexität heutiger Software-Systeme eine manuelle Überprüfung dieser Attribute nicht praktikabel ist, besteht ein Bedarf an Werkzeugen, die den Wartungsprozess durch kontinuierliche Überwachung von Qualitätskriterien unterstützen.

Diesen Zweck erfüllen die sogenannten Software-Dashboards. Sie sind Instrumente zur ganzheitlichen Kontrolle und Steuerung der einzelnen Entwicklungsschritte und Resultate in der Softwarewartung. Sie sammeln Daten über Software-Systeme und stellen sie in einer für den Anwender geeigneten Form dar. Dadurch liefern diese komplexen Werkzeuge

allen Projektbeteiligten die wesentlichen Informationen über den Qualitätszustand einer Software.

Es existieren eine Reihe von Werkzeugen, die sich mit der Analyse von Software-Systemen befassen. Zu den Analysen gehören die Metrikerhebung, das Auffinden von dupliziertem Quelltext und die Validierung der Software-Architektur. Die Filterung und Auswertung der Daten muss jedoch oft manuell durchgeführt werden und auch die grafische Darstellung der Resultate wird in vielen Fällen nicht unterstützt. Ziel dieser Diplomarbeit ist die Automatisierung der Weiterverarbeitung der Analysedaten, um diese anschließend grafisch angemessen zu repräsentieren und somit dem Anwender Aufschluss über den Qualitätszustand einer Software zu geben.

## 1.2 Aufgabenstellung

In dieser Diplomarbeit soll ein Software-Dashboard im Rahmen des Bauhaus-Projekts entwickelt werden, dass die von den verschiedenen Analyse-Werkzeugen generierten Daten sammelt und in geeigneter Form visualisiert. Hierzu werden verschiedene Informationen in einer Datenbank abgelegt und anschließend aufbereitet, gefiltert und zusammengefasst. Zusätzlich zur Betrachtung eines Softwaresystems zu einem bestimmten Zeitpunkt, soll der zeitliche Verlauf der Messgrößen verfolgt werden können.

Das web-basierte Dashboard soll flexibel gestaltet sein, um die Integration weiterer Datenquellen zu ermöglichen. Sowohl Wartungsprogrammierer als auch Projektleiter sollen bei ihrer täglichen Arbeit durch das Dashboard unterstützt werden. Ziel ist es die Benutzeroberfläche möglichst interaktiv und personalisierbar zu gestalten, sodass sie die Eigenschaften einer Portalsoftware umsetzt.

Natürlich kann im Rahmen dieser Arbeit nicht das gesamte Themengebiet der Software-Dashboards abgedeckt werden. Es sollen Lösungsansätze und Verbesserungsmöglichkeiten für die Visualisierung der Analysedaten innerhalb des Bauhaus-Projektes aufgezeigt werden.

## 1.3 Aufbau der Arbeit

Nach der Einleitung in diesem Kapitel werden in Kapitel 2 die Grundlagen beschrieben, die zur thematischen Einordnung und zum Verstehen der Arbeit notwendig sind. In Kapitel 3 wird das Bauhaus-Projekt mit seinen Werkzeugen und Analysen erläutert. Anschließend werden die Schwachstellen und die fehlenden Funktionalitäten von Bauhaus aufgezeigt. Kapitel 4 identifiziert zuerst die Anforderungen and die zu entwickelnde Software und erläutert dann die Technologien die bei der Umsetzung verwendet wurden. Danach folgt die Beschreibung der Umsetzung bevor in Kapitel 5 eine Bewertung der Arbeit stattfindet. In einem abschließend Fazit in Kapitel 6 werden die Ergebnisse beurteilt und zusammengefasst.

# 2 Grundlagen

In diesem Kapitel soll eine Einführung und thematische Einordnung von Software-Dashboards erfolgen. Zuerst wird in Abschnitt 2.1 der Begriff der Softwarewartung definiert. In Abschnitt 2.1.1 werden anschließend Aspekte zur Softwarequalität erläutert, bevor in Abschnitt 2.2 näher auf die Begriffe Dashboards und Software-Dashboards, eingegangen wird.

## 2.1 Softwarewartung

Der Begriff der Wartung im technischen Sinne beschreibt Maßnahmen zur Erhaltung eines Sollzustandes von technischen Systemen oder Einrichtungen. Da Software jedoch keine physischen Eigenschaften besitzt und keine Verschleißteile beinhaltet, unterliegt sie keinerlei Abnutzung. Bei der Softwarewartung wird im Gegensatz zur technischen Wartung nicht der Ursprungszustand wieder hergestellt, sondern das Produkt verändert.

Nach Parnas (1994) altert Software in einem Prozess, der dem Älterwerden der Menschen sehr ähnelt. Gründe für die Software-Alterung sind zum einen, dass Produkteigentümer es versäumen notwendige Anpassungen der Software an neue Anforderungen und Techniken vorzunehmen und zum anderen sind es die notwendigen Anpassungen und Fehlerkorrekturen selber, die zur Alterung beitragen. Letzteres ist der Fall, wenn der Wartungsingenieur Änderungen an einem System vornimmt, ohne Kenntnis über die ursprüngliche Architektur zu besitzen, was somit zu Inkonsistenz zwischen Entwurfsentscheidung und Änderung führt. Nach einer Anpassung muss zum Verstehen des Produktes nicht nur der Entwurf bekannt sein, sondern auch die später hinzu gekommenen Ausnahmen in Betracht gezogen werden. Durch viele solcher Modifikationen versteht mit der Zeit auch der ursprüngliche Entwickler die Architektur nicht mehr.

Eick u. a. (2001) beschreiben weitere Gründe, die zur Software-Alterung beitragen. Dazu gehören Zeitdruck, ungeeignete Programmierwerkzeuge und ungenau formulierte Anforderungen. Außerdem sieht er die Gründe nicht allein in der Software selbst, sondern bezieht Umgebungsfaktoren mit ein. So kann ein Unternehmensumfeld, in dem eine niedrige Moral oder schlechte Kommunikation herrscht ebenso zur Alterung der Software beitragen wie die individuell unterschiedliche Produktivität von Wartungsingenieuren.

Symptome der Software-Alterung sind Abnahme der Performanz durch den strukturellen Verfall sowie die sinkende Zuverlässigkeit aufgrund der steigenden Fehleranzahl. Weiterhin stehen Produkteigentümer vor dem Problem, nicht mehr mit dem Markt „mithalten“ zu können, da die Anpassung von komplexer werdender Software immer schwieriger und somit zeitintensiver wird.

Zwar ist die Alterung von Software nach Parnas unvermeidbar, jedoch kann der Prozess durch eine Anpassung der Software weitgehend kompensiert und verzögert werden. Zu den vorbeugenden Maßnahmen zählen gute Dokumentation, Überprüfung der Entwurfsentscheidungen durch mehrere Entwickler und ein Entwurf, der mögliche Anpassungen bei der Planung mit einbezieht.

Nach IEEE 1219 (1998) ist der Begriff Softwarewartung definiert als „die Änderung eines Software Produkts nach dessen Auslieferung, um Fehler zu korrigieren, um die Performanz oder andere Attribute zu verbessern oder um Anpassungen an geänderte Umgebungen vorzunehmen.“

Swanson (1976) teilt die Aktivitäten der Softwarewartung nach den Gründen, die zu einer Wartungsaktivität führen können, in drei Kategorien ein: *Korrektive Wartung*, *Adaptive Wartung* und *Perfektive Wartung*. Anpassungen aufgrund von Fehlern in der Software werden als *Korrektive Wartung* bezeichnet. Wird eine Software an sich ändernde Techniken oder Anforderungen angepasst, bezeichnet man das als *Adaptive Wartung* und unter der letzen Kategorie *Perfektive Wartung* versteht man die Verbesserung der Wartbarkeit und Leistung einer Software.

Die Einteilung nach IEEE 1219 (1998) erweitert die Kategorien *Korrektive Wartung*, *Adaptive Wartung* und *Perfektive Wartung* noch um eine weitere namens *Notfall*. Wartungsaktivitäten, die in der Kategorie *Notfall* zusammengefasst werden, sind unvorhergesehene, außerplanmäßige Änderungen, die durchgeführt werden müssen, um die Software in einem betriebsbereiten Zustand zu halten.

Ein weiterer Aspekt der Wartung der von Eick u. a. (2001) aufgegriffen wird, ist die Speicherung der Änderungshistorie. Er definiert die Anpassung als „jegliche Veränderung der Software, die in der Versionsdatenbank aufgezeichnet wird “.

### 2.1.1 Softwarequalität

Nach ISO/IEC 9126 (2001) ist Softwarequalität die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

In der Praxis jedoch hängt die Definition von Softwarequalität vom Betrachter ab. Ein Softwareentwickler wird die Qualität des Produktes eventuell nach Attributen des Codes wie Kopplung und Kohäsion beurteilen, wohingegen für den Manager Ziele auf Projektebene wie z.B. Zuverlässigkeit und Änderbarkeit die Produktqualität definieren.

Qualitätsmodelle zerlegen und konkretisieren Softwarequalität, sodass diese messbar wird. Das Qualitätsmodell nach ISO/IEC 9126 (2001) bezieht sich auf die Produktqualität einer Software. Die Norm definiert sechs Qualitätsmerkmale, welche sich durch weitere Merkmale verfeinern lassen, wodurch ein hierarchisches Modell entsteht (siehe Abbildung 2.1).

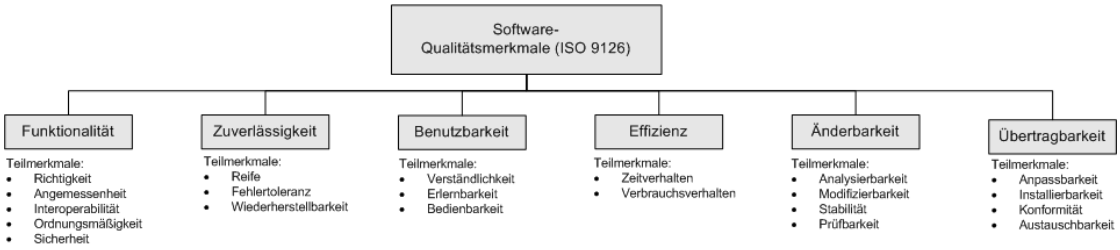


Abbildung 2.1: Softwarequalitätsmerkmale nach ISO 9126

Das Merkmal Funktionalität beschreibt das Vorhandensein von Funktionen mit festgelegten Eigenschaften. Die Funktionen erfüllen die definierten Anforderungen. Die Zuverlässigkeit beschreibt die Fähigkeit der Software, ihr Leistungsniveau unter festgelegten Bedingungen in einem festgelegten Zeitraum zu halten. Unter Benutzbarkeit versteht man den Aufwand, der zur Benutzung erforderlich ist und die individuelle Beurteilung der Benutzung durch eine festgelegte oder vorausgesetzte Gruppe. Das Merkmal Effizienz beschreibt das Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen. Änderbarkeit ist definiert durch den Aufwand, der nötig ist, um Änderungen durchzuführen. Änderungen können Korrekturen, Verbesserungen oder Anpassungen an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen. Das letzte Merkmal, Übertragbarkeit, beschreibt die Eignung der Software von einer Umgebung in eine andere übertragen zu werden. Umgebung kann organisatorische Umgebung, Hardware- oder Softwareumgebung einschließen.

### 2.1.2 Softwaremetriken

Ein Maß beschreibt im allgemeinen die Abbildung von Eigenschaften von Objekten der realen Welt, auf Zahlen oder Symbole. Diese Eigenschaften von Objekten nennt man auch Merkmal. Bei der Anwendung eines Maßes, dem Messen, kann jedes Merkmal verschiedene Werte annehmen. Diese Werte nennt man Merkmalsausprägungen oder nur kurz Ausprägungen. Ein Merkmal stellt die interessierende Größe der Messung dar und der Wert des Merkmals die Ausprägung. Als Metrik bezeichnet man in der Mathematik eine Distanzfunktion zur Bestimmung der Distanz von Größen einer quantitativen Skala. In der Softwaretechnik ist eine Metrik ein Maß zur Ausprägung eines Merkmals.

Der Standard 1061 des Institutes of Electrical and Electronics Engineers (IEEE) definiert eine Metrik wie folgt:

> Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Softwareeinheit.
>
> (IEEE 1061, 1992)

Fenton und Pfleeger (1998) teilen die Eigenschaften, die gemessen werden können, in drei Kategorien ein: Prozessmetriken, Produktmetriken und Ressourcenmetriken. Im Bereich der Wartung sind dabei besonders die Prozess- und Produktmetriken von Bedeutung. Die Prozessmetriken messen, wie der Name schon sagt, Eigenschaften von Prozessen. Dazu gehören Kriterien wie Kostenverteilung, Produktivität oder Dauer von Arbeitsschritten. Prozessmetriken dienen also zur Optimierung des Wartungsprozesses.

Eigenschaften der Software selbst werden durch die Produktmetriken gemessen. Die Produktmetriken lassen sich weiter unterteilen in externe und interne Produktmetriken. Die externen Eigenschaften sind im Wesentlichen die Eigenschaften, die im Qualitätsmodell der *ISO* beschrieben sind, beispielsweise Funktionalität, Zuverlässigkeit, Benutzbarkeit und Effizienz (siehe Abschnitt 2.1.1). Als interne Eigenschaften werden Merkmale einer Software wie Größe, Komplexität oder Struktur bezeichnet.

Die Messung externer Eigenschaften eines Softwareproduktes ist nur schwer umsetzbar. Nach Bommer u. a. (2008) stehen im Idealfall die internen und externen Merkmale in Beziehung zueinander, sodass sich von den internen auf die externen Eigenschaften schließen lässt.

Ein Problem bei der Messung von Softwarequalität stellt das Finden des geeigneten Maßes da. Oft werden zu viele Daten gesammelt und alle Merkmale, die es ermöglichen, gemessen (Nance und Arthur, 2002). Durch die Menge an unübersichtlichen Ergebnissen, werden die wichtigen Daten verdeckt. Die Auswertung von Messdaten, die ohne jeden Zweck erhoben worden sind, liefert keine hilfreichen Erkenntnisse zur Verbesserung der Softwarequalität. Bevor etwas gemessen wird, sollte das Ziel der Messung klar sein. Der *Goal-Question-Metric* Ansatz ist ein Konzept, das aus Zielen geeignete Metriken ableitet.

### 2.1.3 Goal-Question-Metric

Nach Basili u. a. (1994) ist eine Messung von Softwarequalität nur sinnvoll und effektiv, wenn sie sich nach bestimmten Zielen richtet. Diese Art der Software-Messung bezeichnet man auch als „Zielorientiertes Messen“. Die Vorgehensweise ist dabei *Top-Down* („von oben nach unten“), da man von Zielen ausgehend zu Metriken gelangt (siehe Abbildung 2.2).

Der Goal-Question-Metric (GQM) Ansatz ist eine Methode zur Entwicklung eines Systems zur Messung von Softwarequalität. Ausgehend von Qualitätszielen werden Fragen

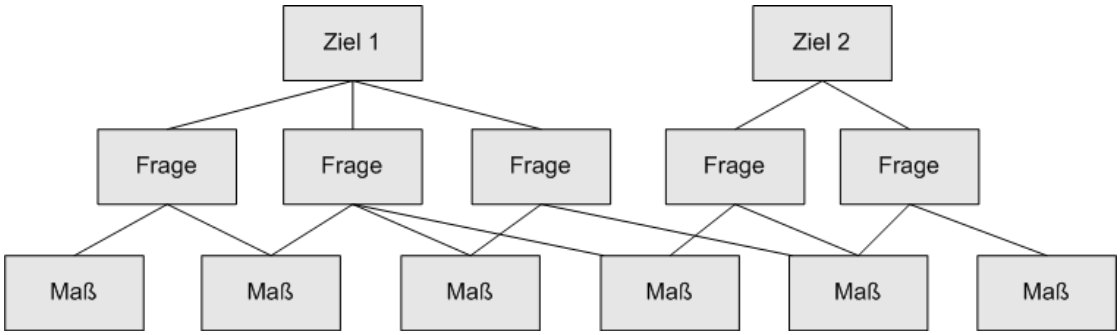


Abbildung 2.2: Der Goal-Question-Metric Ansatz

formuliert, deren Beantwortung zur Zielüberprüfung dient. Für den Grad der Erfüllung wird unter Verwendung von geeigneten Metriken eine Messgröße ermittelt.

Das Resultat der Anwendung der Goal-Question-Metric Methode ist ein Messplan, der aus drei Ebenen besteht:

**Konzeptionelle Ebene (Ziel):** Ein Ziel bezieht sich auf ein Objekt, das beobachtet werden soll. Zur Definition eines Ziels gehören der Zweck und der Qualitätsaspekt, unter dem ein Objekt untersucht werden soll, die Personen, für die die Ergebnisse der Erstellung des Messplans von Interesse sind und der Kontext, in dem ein Messplan erarbeitet wird. Mögliche Objekte sind Produkte, Prozesse und Betriebsmittel.

> **Produkte:** Gegenstände, Dokumente und Ergebnisse, die im Entwicklungsprozess entstehen, wie z.B. die Spezifikation.
>
> **Prozesse:** Software-bezogene Aktivitäten, in der Regel in zeitlicher Abhängigkeit, wie z.B. Entwerfen, Testen oder Spezifizieren.
>
> **Betriebsmittel:** Mittel, die im Entwicklungsprozess verwendet werden, um zu einem Ergebnis zu gelangen. Darunter fallen z.B. Personen, Hardware oder Büroräume.

**Operationelle Ebene (Fragen):** Eine Menge von Fragen beschreibt den Weg, auf dem ein bestimmtes Ziel erreicht werden soll. Die Fragen charakterisieren das zu untersuchende Objekt unter Berücksichtigung des Qualitätsaspektes und beurteilen die Qualität aus einem bestimmten Blickwinkel heraus.

**Quantitative Ebene (Maße):** Mit jeder Frage werden Daten assoziiert, die die Frage quantitativ beantworten. Die Daten können entweder objektiv oder subjektiv sein. Objektiv bedeutet, dass die Daten nur vom untersuchten Objekt und nicht vom Blickwinkel

abhängen, wo hingegen bei subjektive Daten sowohl das Objekt als auch der Blickwinkel Einfluss auf die Daten haben.

Nachdem man zu geeigneten Metriken gelangt ist, wird ein Mechanismus für die Messung und Datensammlung der Maße entworfen. Bevor die eigentliche Messung erfolgt muss die Validität der Messwerte überprüft werden, also ob das Maß tatsächlich das misst, was es vorgibt zu messen. Erst danach erfolgt die eigentliche Messung. Die Ergebnisse müssen anschließend interpretiert werden.

### 2.1.4 Skalenniveau

Die möglichen Ergebnisse einer Messung können je nach Datentyp klassifiziert werden. Diese Klassifizierung nennt sich Skalenniveau. Man unterscheidet zwischen *qualitativen* (*kategorischen*) und *quantitativen* (*numerischen*) Merkmalen. Die verschiedenen Eigenschaften von Objekten werden bei *kategorischen* Merkmalen durch eine wertmäßige Angabe „mit Worten“ beschrieben. Die Mess- und Zählbaren werden durch die *numerischen* Merkmale beschrieben. Sie werden durch eine mengenmäßige Angabe, meist eine reelle Zahl, repräsentiert.

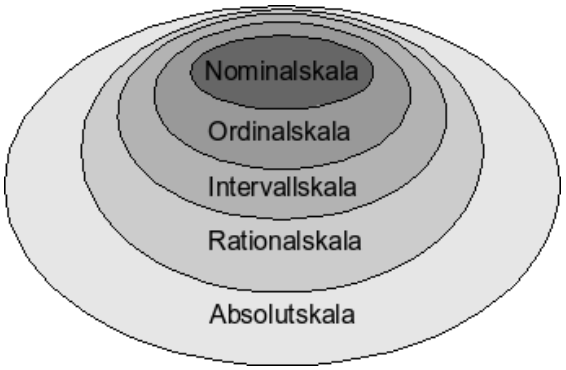


Abbildung 2.3: Skalenniveau

Je nach Skalenniveau werden die Ergebnisse dann einer bestimmten Messskala, oder kurz Skala, zugeordnet. Bei *qualitativen* Merkmalen wird eine Nominal- oder Ordinalskala verwendet und bei *quantitativen* eine Intervall-, Rational- oder Absolutskala. Das Skalenniveau bestimmt, welche Operationen mit den entsprechenden Merkmale zulässig sind. Die Skalen bilden die Operationen betreffend eine Ordnung (Abbildung 2.3). Angefangen von der Nominalskala, die die wenigsten Operationen erlaubt, bis zur Absolutskala, bei der die meisten Operationen anwendbar sind, werden die Skalen immer mächtiger. Die Operationen einer Skala sind ebenso auf allen höheren Skalen durchführbar.

### Nominalskala

Die Nominalskala ist das niedrigste Skalenniveau. Ein Merkmal ist nominal, wenn die Ausprägungen Namen oder Kategorien sind, die sich den Einheiten zuordnen lassen. Die Ausprägungen unterscheiden sich, jedoch existiert keine weitere Relation, wie zum Beispiel eine Ordnungsrelation. Die einzige Aussage, die sich von einer Nominalskala ableiten lässt, ist die Entscheidung über Gleich- oder Ungleichheit. Die einzig zulässige Operation ist die Prüfung auf Identität. Statistisch kann nur der häufigste Wert, der Modus, ausgewertet werden. Beispiele für nominal skalierte Merkmale finden sich in Tabelle 2.1.

| **Nominalskala** | Kategorien, Namen (ohne Ordnung) |
|---|---|
| **Operationen** | Identität |
| **Bedingungen** | Symmetrie, Reflexivität, Transitivität |
| **Statistik** | Häufigkeit |
| **Beispiele** | Geschlecht : weiblich $\neq$ männlich |
| | Haarfarbe : blond $\neq$ braun |

Tabelle 2.1: Nominalskala

### Ordinalskala

Die nächst höhere Skala bildet die Ordinalskala. Die Ordinalskala erweitert die Nominalskala um eine vollständige Ordnung, d.h. die Ausprägungen lassen sich in eine Rangfolge bringen. Die Abstände sind jedoch nicht interpretierbar. Als zusätzliche Operation ermöglicht die Ordinalskala den Vergleich der Kategorien. Statistisch kann der Median gebildet werden, der aussagt, welcher Wert im Mittel gemessen wurde.

| **Ordinalskala** | vollständige Ordnung |
|---|---|
| **Operationen** | Vergleich der Kategorien |
| **Bedingungen** | Transitivität, Konnexivität |
| **Statistik** | Median |
| **Beispiele** | Einkommen: niedrig $<$ mittel $<$ hoch |

Tabelle 2.2: Ordinalskala

### Intervallskala

Bei der Intervallskala lassen sich die Abstände zwischen den Ausprägungen durch eine Distanzfunktion sinnvoll definieren. Dadurch erlaubt die Intervallskala Rechenoperation wie die Addition und die Subtraktion auf den ermittelten Werten. Es existiert kein natürlicher Nullpunkt. Sowohl die numerische Größe der Einheit, also der Abstand, als

auch der Nullpunkt können frei gewählt werden. Ein Beispiel hierfür ist die Temperatur in °C, wie in Tabelle 2.3 zu sehen ist, da der Nullpunkt von 0°C willkürlich festgesetzt wurde.

| **Intervallskala** | Zahlen |
|---|---|
| **Operationen** | Addition, Subtraktion |
| **Statistik** | Standardabweichungen, Mittelwert |
| **Beispiele** | Temperatur: 10°C + 10°C = 20°C |

Tabelle 2.3: Intervallskala

### Rationalskala

Im Gegensatz zu der Intervallskala existiert bei der Rationalskala ein natürlicher Nullpunkt. Dadurch werden die Multiplikation und die Division als Rechenoperation sinnvoll und ermöglichen Aussagen über Verhältnisse von Messwerten. Beispiele für Merkmale auf einer Rationalskala sind Messungen physikalischer Größe wie Gewicht oder Länge, wie Tabelle 2.4 veranschaulicht.

| **Rationalskala** | absoluter Nullpunkt |
|---|---|
| **Operationen** | Division, Multiplikation |
| **Statistik** | Korrelation, geometrisches Mittel |
| **Beispiele** | Gewicht: 20kg · 3 = 60kg |

Tabelle 2.4: Rationalskala

### Absolutskala

Das letzte Skalenniveau beschreibt die Absolutskala. Anders als bei der Rationalskala, ist die Maßeinheit der Absolutskala natürlich gegeben und somit eindeutig festgelegt.

| **Absolutskala** | natürlich gegeben |
|---|---|
| **Operationen** | absoluter Vergleich |
| **Beispiele** | Anzahl von Menschen in einem Raum |

Tabelle 2.5: Absolutskala

### Standardmetriken

**Lines of Code:** Die Metrik *Lines of Code* misst die Anzahl der Zeilen in einem bestimmten Bereich. Die Erhebung kann auf verschiedene Einheiten des Quelltextes erhoben werden, wie beispielsweise auf Module, Funktionen oder Anweisungsblöcken.

**McCabe Complexity:** Die *Mc CabeComplexity* ist ein Maß für die Komplexität eines Quelltextbereichs. Sie wird aus dem Kontrollflussgraphen berechnet durch:

$$McCabe(G) = \#Kanten(G) - \#Knoten(G) + 2$$

oder einfacher

$$McCabe(G) = \#Binrverzweigungen + 1$$

**Nesting:** Durch den Nesting-Wert wird die Schachtelungstiefe eines Quelltextabschnittes beschrieben. Ineinander verschachtelte Verzweigungen verschlechtern die Verständlichkeit des Quelltextes erheblich. Im Zusammenhang mit der McCabe Komplexität, kann entschieden werden ob Anweisungen in verschachtelter Form vorliegen oder nacheinander kommen.

### 2.1.5 Visualisierung

Das Gebiet der Visualisierung lässt sich in zwei Bereiche unterteilen, in die *wissenschaftliche Visualisierung* und die *Informationsvisualisierung*. Die wissenschaftliche Visualisierung befasst sich mit der Visualisierung von gemessenen Daten oder Simulationsergebnissen, denen physikalische Prozesse zugeordnet werden. Bei der Informationsvisualisierung hingegen werden Daten visualisiert, die nicht unmittelbar mit physikalischen Zuständen oder Vorgängen zusammenhängen. Diese Daten werden auch als *abstrakte* Daten bezeichnet. Algorithmen sind nach Diehl (2007) eine Art von Informationen, wodurch das Gebiet der Softwarevisualiserung einen Teilbereich der Informationsvisualisierung darstellt.

#### Informationsvisualisierung

Unter dem Begriff der Informationsvisualisierung versteht man Konzepte, Methoden und Tools zur visuellen Darstellung von Informationen aus Datenbanken, Bibliotheken oder Dokumentensammlungen. Bevor die abstrakten Daten visuell dargestellt werden, erfolgt eine computergestützte Aufbereitung und Strukturierung der Informationen. Durch die Informationsvisualisierung können spezifische Daten aufgefunden werden und Relationen erkannt werden. Sie liefert verschiedene Sichten auf einen identischen Datenbestand und setzt diesen in Kontext zu anderen Informationen. „Durch angemessene Visualisierungen, treten Fehler und Messartefakte besonders hervor. Aus diesem Grund sind sie für die Qualitätskontrolle unerlässlich.“ (Ware, 2000)

### 2.1.6 Softwarevisualisierung

In Abschnitt 2.1 wurden mögliche Wartungsarbeiten in die Kategorien korrigierende Wartung, adaptive Wartung und perfektive Wartung eingeteilt. Für die effektive und effiziente Durchführung dieser Arbeiten ist die werkzeugbasierte Unterstützung der Aufgaben, des Programmverstehens und der Programmanpassung unerlässlich. Für das Programmverstehen können zahlreiche Artefakte eines Softwaresystems die Grundlage bilden, unter anderem Dokumente, Modelle, Quellcode, Datenbanktabellen, Testdaten und weitere. In den seltensten Fällen stimmen jedoch Modelle und Dokumentation mit der aktuellen Version der Software überein, sodass gilt:

> Der Quelltext ist oft der einzige Informationsträger, der ein Altsystem wirklich verlässlich beschreibt.
>
> (Eisenbarth u. a., 1999)

Dadurch bildet der Quelltext die Basis des Programmverstehens. Nach einer Studie von Fjeldstad und Hamlen (1997) wird im Rahmen einer Wartungsaufgabe in etwa die Hälfte der Zeit für das Programmverstehen verbraucht. Damit der Wartungsprozess schneller und damit kosteneffizienter abläuft, muss die zeitintensive Phase des Programmverstehens durch geeignete Repräsentation unterstützt werden. Mit dieser Aufgabe befasst sich der Bereich der Softwarevisualiserung, der im nächsten Abschnitt beschrieben werden soll.

Als Visualisierung bezeichnet man im Allgemeinen den Prozess der Transformation von Daten, Informationen und Wissen (Däßler, 1999). Durch die visuelle Darstellung können Beziehungen und verschiedene Aspekte auf einen Blick sichtbar gemacht werden. Ziel der Visualisierung ist es, komplexe Prozessabläufe und Objekte zu veranschaulichen oder zu vereinfachen. Durch Strukturierung und Klassifikation der Informationen wird der kognitive Zugang zu den Messdaten erleichtert. Die vorliegenden Daten können besser interpretiert werden und versteckte Trends können erkannt werden.

Software unterscheidet sich von anderen gefertigten Produkten, da sie nicht sichtbar bzw. greifbar ist. Der Code ist zwar in Form von Dateien auf der Festplatte vorhanden, jedoch müssen Entwickler nicht nur über die Eigenschaften eines Systems urteilen, sondern auch das Verhalten zur Laufzeit verstehen. Mit der Softwarevisualisierung soll Software durch grafische Repräsentation sichtbar gemacht werden. Die grafische Darstellung hat zum Ziel, das Verständnis und die Beurteilung des zugrunde liegenden Systems zu erleichtern. Man unterscheidet drei verschiedene Arten von Softwarevisualisierungen, basierend auf den Systemeigenschaften, die sie darstellen.

**Softwarestruktur:** Sie bezieht sich auf die Visualisierung von Daten der statischen Programmanalyse, also Informationen über Elemente und ihre Abhängigkeiten in einem Softwaresystem. Dazu gehören z.B. der Quelltext oder der Aufrufgraph eines Systems.

**Laufzeitverhalten:** Informationen, die das Laufzeitverhalten betreffen, werden z.B. durch Algorithmenanimation, Datenfluss-, oder Kontrollflussdiagramme dargestellt.

**Softwareevolution:** Diese Art der Visualisierung befasst sich mit der Darstellung der Veränderungen eines Softwaresystems. Veränderungen können beispielsweise durch Anpassungen oder Fehlerkorrekturen hervorgerufen wurden.

### Visualisierungs-Pipeline

Die Visualisierungs-Pipeline beschreibt den Weg von der Datenerfassung bis zu Bildgenerierung (Diehl, 2007). Sie besteht aus drei aufeinanderfolgenden Schritten, bei denen die Informationen des vorhergehenden als Eingabe für den nächsten Schritt dienen.

**Datenerfassung:** Die Datenerfassung extrahiert die Informationen aus verschiedenen Artefakten. Dazu gehören unter anderem die Dokumentation, der Quelltext und die Testergebnisse eines Softwaresystems.

**Datenanalyse:** Die erfassten Daten sind zu umfangreich, um direkt dem Nutzer präsentiert werden zu können. Durch eine Reihe von Methoden, wie beispielsweise der statischen Programmanalyse oder der Filterung, werden die Daten auf die wichtigen Aspekte reduziert. Diese Aufbereitung der Daten beruht auf einem Informationsmodell. In einem Informationsmodell ist der Sachverhalt, den man mit der Visualisierung sehen möchte, korrekt modelliert.

**Visualisierung:** Die verdichteten Daten werden auf ein visuelles Modell abgebildet, z.B. durch die Umwandlung in grafische oder geometrische Informationen. In einem geometrischen Modell werden die Daten so gespeichert, dass sie gerendert werden können. Rendern bezeichnet die Transformation des geometrischen Modells in den Darstellungsraum. Der letze Schritt der Visualisierungspipeline ist also die Ausgabe der Daten in Form eines Bildes auf einem Bildschirm oder einem anderen Ausgabemedium.

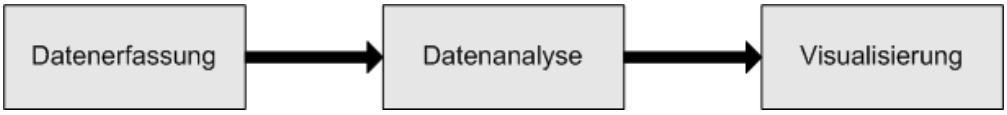


Abbildung 2.4: Visualisierungspipeline

Die grafische Darstellung kann vom Betrachter beeinflusst werden, z.B. durch Vergrößern. Der Schritt, den Diehl (2007) unter *Visualisierung* zusammenfasst, wird in der Literatur oft in die zwei Prozesse, *Transformation in das geometrische Modell* und das *Rendern* aufgeteilt (Card u. a., 1999).

Werkzeuge, die zur Softwarevisualisierung eingesetzt werden, müssen diese Schritte der Visualisierungs-Pipeline ausführen, um die Eigenschaften einer Software grafisch darzustellen. Maletic u. a. (2002) klassifizieren die Softwarevisualisierung in fünf Dimensionen. Diese Dimensionen beschreiben das *warum*, *wer*, *was*, *wie* und *wo*, der Softwarevisualisierung. Mit Hilfe dieser Taxonomie können Softwarevisualisierungs-Systeme klassifiziert werden.

**Aufgaben:** *Warum* ist die Softwarevisualiserung nötig?

**Zielgruppe:** *Wer* wird die Softwarevisualisierung verwenden?

**Vorgaben:** *Was* für eine Datenquelle soll repräsentiert werden?

**Repräsentation:** *Wie* soll die Datenquelle dargestellt werden?

**Form:** *Wo* (auf welchem Medium) soll sie visualisiert werden?

Die Aufgabendimension beschreibt, warum eine Visualisierung gebraucht wird. Die verschiedenen Aufgaben der Softwaretechnik umfassen unter anderem Implementieren, Fehlersuche, Testen, Wartung, Fehlerdiagnose, Re-Engineering, Reverse-Engineering, Prozess Management und Marketing. Je nach Aufgabenbereich muss die Abstraktionsebene und die Art der Information unterschiedlich sein.

Die Dimension der Zielgruppe beschreibt die spätere Gruppe von Nutzern, z.B. Tester, Wartungsprogrammierer oder Entwickler. Dabei können außerdem die Fähigkeiten eine entscheidende Rolle spielen.

In der Dimension der Vorgaben wird beschrieben, welche Daten visualisiert werden sollen, jedoch auf sehr niedriger Abstraktionsebene. Mögliche Vorgaben zur Visualisierung wären z.B. die Architektur, Ausführungsinformationen, der Quelltext, Messdaten und Metriken, Dokumentation und Prozessinformationen. In der detaillierten Beschreibung umfasst diese Dimension außerdem Aspekte zur Datensammlung, also mit welcher Methode die Sammlung geschehen soll und wie viel Zeit die Sammlung der Daten in Anspruch nimmt. Sehr wichtig ist ebenso die Frage der Skalierbarkeit. Aufgaben in der Softwaretechnik erfordern meistens die Visualisierung von großen Systemen und somit großen Datenmengen, für die geeignete Darstellungen und Metriken gefunden werden müssen.

Die Repräsentationsebene beschreibt wie die Visualisierung umgesetzt werden soll. Je nach Zielgruppe, Vorgabe und verfügbarem Medium muss eine Repräsentation definiert werden, die die ausgewählte Information am besten dem Nutzer zugänglich macht. Zwei Kriterien sind hierbei die Ausdrucksfähigkeit und die Effektivität. Die Relation zwischen visuellen Parametern und Datenwerten muss gleich sein, da andernfalls mehr als ein Wert auf das gleiche visuelle Element abgebildet wird. Mit der Effektivität ist die der Visualisierung zugrunde liegende Metapher gemeint, da im Falle von mengenmäßigen Daten nicht nur die Anzahl der visuellen Parameter entscheidend ist, sondern die Parameter müssen in der Lage sein, die Daten richtig zu repräsentieren.

Die letzte Dimension beschreibt das Medium der Visualisierung, also wo die Visualisierung dargestellt wird, z.B. auf einem Monitor oder in einer virtuellen Umgebung.

Young und Munro (1998) beschreiben eine Reihe von Eigenschaften, die eine gute Visualisierung ausmachen. Diese Eigenschaften sollten beim Entwurf von Visualisierungen bedacht werden, jedoch schließen sich einige dieser Kriterien gegenseitig aus, sodass Kompromisse gemacht werden müssen.

### Einfache Navigation und minimale Verwirrung

Die Visualisierung sollte gut strukturiert sein und Hilfestellung leisten bei der Navigation durch die Darstellung. Das Risiko, dass der Nutzer die Orientierung verliert, sollte möglichst gering gehalten werden.

### Hoher Informationsgehalt

Visualisierungen sollten soviel Informationen präsentieren wie möglich ohne den Nutzer zu überfordern. Dieses Kriterium widerspricht allerdings der Eigenschaft der niedrigen visuellen Komplexität, sodass zwischen diesen beiden ein Kompromiss gefunden werden muss.

### Niedrige visuelle Komplexität und gute Strukturierung

Zwar hängt die visuelle Komplexität von der Komplexität der Informationen ab, die visualisiert werden, jedoch sollte soviel wie möglich getan werden, um die visuelle Komplexität möglichst gering zu halten und durch eine gute Strukturierung der Daten die Navigation für den Anwender zu erleichtern.

### Verschiedene Detailebenen

Die Detailstufe und der Inhalt der präsentierten Informationen sollten so gewählt werden, dass sie den Anforderungen des Anwenders entsprechen. So erwartet ein Nutzer beispielsweise nach dem Start der Anwendung zuerst eine Visualisierung der Struktur des Gesamtsystems. Sobald er diese verstanden hat, möchte er dann Detailinformationen zu einzelnen Bereichen, um diese genauer zu untersuchen.

### Sinnvolle Verwendung von visuellen Metaphern

Durch Metaphern werden die Visualisierung dem Nutzer durch ihm bereits bekannte Konzepte näher gebracht und erleichtern ihm somit das Verstehen der präsentierten Informationen.

#### Geeignete Nutzungsschnittstelle

Die Nutzungsschnittstelle sollte flexibel genug gestaltet sein, um dem Anwender eine einfache Navigation und Bedienung zu ermöglichen.

#### Interaktion mit anderen Informationsquellen

Visualisierungen bieten eine andere Sicht auf die Informationen, die sie repräsentieren, jedoch können sie meistens diese Informationen nicht völlig ersetzen. Im Idealfall sollten sich die Ausgangsdaten oder Sichten auf die Ausgangsdaten mit der Visualisierung in Zusammenhang bringen lassen.

#### Sinnvoller Einsatz von Interaktion

Eine Visualisierung die dem Anwender eine Möglichkeit zur Interaktion zur Verfügung stellt, erleichtert das Verständnis der Darstellung. Sie unterstützt den Anwender außerdem bei der Suche nach Informationen.

#### Möglichkeit zur Automatisierung

Die Visualisierung sollte sich relativ einfach, dynamisch erstellen lassen. Der Hauptteil der Visualisierung sollte automatisch generiert werden mit geringem Einfluss durch den Nutzer.

### 2.1.7 Diagramme

Diagramme stellen Sachverhalte, Informationen oder Daten grafisch dar. Sie sollen den Zusammenhang zwischen zwei oder mehr voneinander abhängenden Messgrößen oder Werten verdeutlichen. Durch grafische Darstellungen, lassen sich Sachverhalte besser auf „einen Blick“ erfassen. Vielen Diagrammformen liegt das kartesische Koordinatensystem zugrunde, welches durch die horizontale X-Achse und die vertikale Y-Achse gebildet wird. Von den vier Quadranten wird zur Darstellung meistens der rechte obere verwendet.

#### Punkt- und Blasendiagramm

Bei einem Punktdiagramm (auch Streudiagramm) werden Wertpaare als XY-Koordinaten in das kartesische Koordinatensystem eingetragen. Diese Wertpaare werden als Punkte (Kreuze oder Kreise) dargestellt. Ziel eines Punktdiagrammes ist es, Abhängigkeitsstrukturen der beiden Merkmale durch das Muster der Punkte zu erkennen. Diese Beziehung

zwischen zwei oder mehr statistischen Variablen nennt man Korrelation. Ein Beispiel für häufig auftretende Korrelation sind z.B. Ballungen (Cluster).

Ein drittes metrisches Merkmal kann über die Größe der Punkte abgebildet werden. Dadurch wird das Punktdiagramm zu einem Blasendiagramm erweitert.

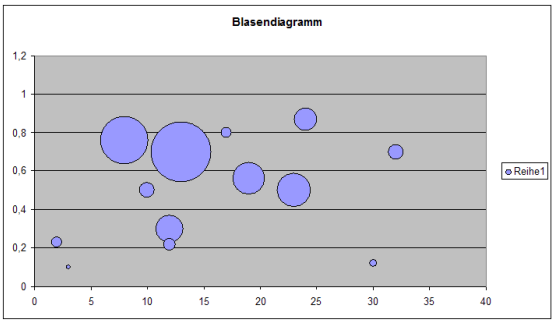


Abbildung 2.5: Punktdiagramm

### Kreisdiagramm

Das Kreisdiagramm, auch Torten- oder Kuchendiagramm genannt, ist kreisförmig und weist jeder Ausprägung des Merkmals einen Kreissektor zu. Ein Sektor wird durch zwei Radiuslinen von der Mitte zum Radius definiert. Der Winkel und somit die Fläche des Sektors stellt die relative Häufigkeit des Auftretens des Merkmals dar. Der Winkel des Sektors kann ermittelt werden, indem die relative Häufigkeit mit 360° multipliziert wird. Der Radius des Diagrammes ist dabei beliebig, da er keinen Einfluss auf die berechneten Winkel hat. Das Kreisdiagramm ist gut geeignet, um Verteilungen oder Anteile grafisch darzustellen. Es sollten jedoch nicht mehr als sieben bis neun Kategorien oder Klassen vorliegen. Viele kleine Teilwerte erschweren ebenso die Lesbarkeit und sollten deshalb zu einem Segment, z.B. „Sonstige“zusammengefasst werden. Ein Kreisdiagramm vermittelt einen guten Gesamtüberblick über einen Sachverhalt.

### Liniendiagramm

Das Liniendiagramm stellt eine Reihe von Messwerten als Linienzug dar. Wenn genügend Daten bei einer Messung gesammelt wurden, können die in das Koordinatensystem eingetragenen Wertpaare mit einer Linie verbunden werden. Diese Diagrammart ist besonders gut geeignet zur die Darstellung von Datenpunkten im Zeitverlauf. Hierbei werden z.B. Trends besser sichtbar als beim Punktdiagramm.

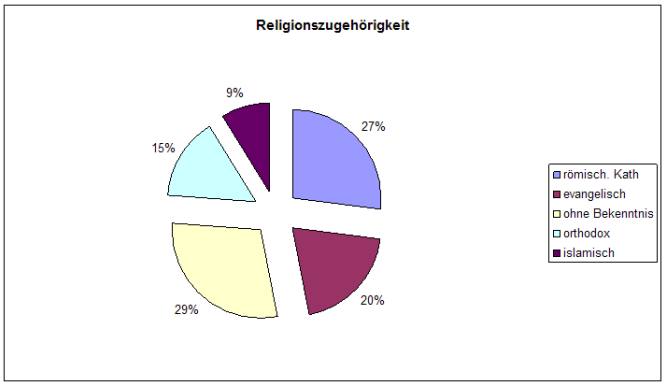


Abbildung 2.6: Kreisdiagramm

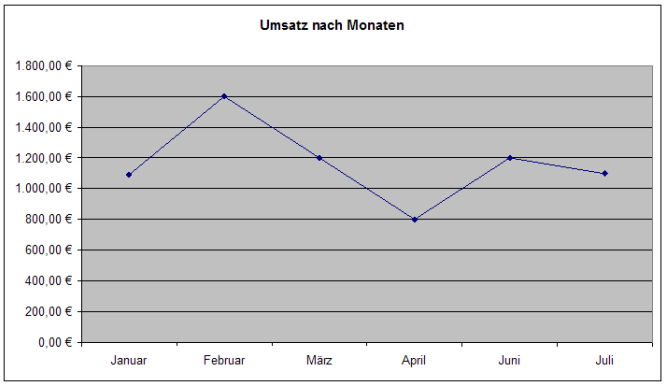


Abbildung 2.7: Liniendiagramm

### Säulen- und Balkendiagramm

Bei einem Säulendiagramm werden die Ausprägungen von Messwerten durch senkrecht auf der X-Achse eines kartesischen Koordinatensystems stehende Rechtecke repräsentiert. Bei qualitativen Daten kann die Unterteilung der X-Achse frei gewählt werden. Bei quantitativen Daten sind die Abstände gegeben. Die Fläche und die Höhe der gleichbreiten Säulen verhält proportional zur Häufigkeit (relativ oder absolut). Die Y-Achse wird entsprechend der vorkommenden Häufigkeiten skaliert. Sind die Daten ordinal skaliert, muss die Reihenfolge auf der Abszisse willkürlich gewählt werden, was eventuell zu Interpretationen führt, die nicht in den Daten vorhanden sind. Deshalb eignen sich mindestens ordinalskalierte Daten besser für diese Darstellungsform. Damit das Diagramm übersichtlich bleibt, sollte die Anzahl der dargestellten Merkmale möglichst gering sein. Ist das Koordinatensystem um 90 Grad gedreht, bezeichnet man diese Diagrammart als Balkendiagramm.

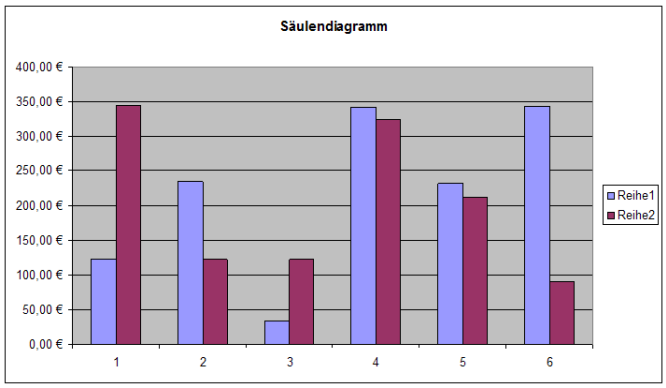


Abbildung 2.8: Säulendiagramm

### Histogramm

Das Histogramm ist eine Sonderform des Säulendiagramms und eine grafische Darstellung für die Häufigkeitsverteilung von Messwerten. Um Daten in einem Histogramm darstellen zu können, müssen diese klassiert werden, indem die nach der Größe geordneten Daten in k Klassen eingeteilt werden. Die Größe der Intervalle kann dabei unterschiedlich sein. Auf der X-Achse werden die Klassengrenzen aufgetragen und über den Klassenintervallen wird ein Rechteck errichtet. Bei unterschiedlichen Klassengrößen repräsentiert die Fläche der Säule die Häufigkeit und die Höhe die Häufigkeitsdichte. Es können sowohl die absolute als auch die relative Häufigkeit in einem Histogramm dargestellt werden.

## Netzdiagramm

Beim Netzdiagramm besitzt jedes Merkmal eine eigene Größenachse. Die Achsen gehen vom Mittelpunkt aus und sind kreisförmig angeordnet. Die Werte der Datenreihen werden über Linien verbunden. Das von den Linien eingeschlossene Gebiet ist der aussagekräftige Teil der Darstellung. Die Linien oder die gesamte Fläche werden oft farblich gekennzeichnet. Das Diagramm bietet einen guten Vergleich, wenn Objekte nach unterschiedlichen Kriterien beurteilt werden sollen. Die Anzahl von Merkmalen sollte zwischen fünf und sieben liegen und zehn nicht überschreiten. Mindestens drei werden benötigt, um eine sichtbare Verbindung zu erhalten.

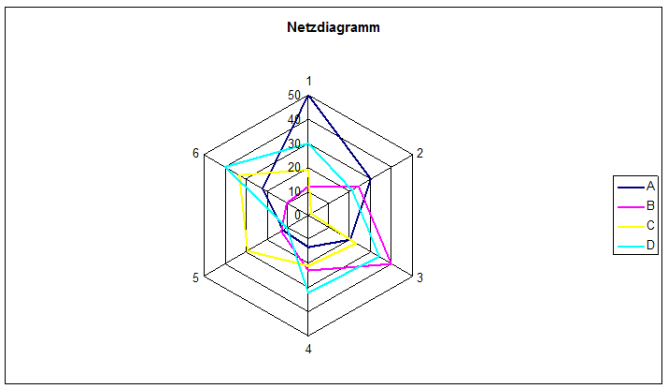


Abbildung 2.9: Netzdiagramm

## Hierarchien

Hierarchien sind oft verwendete Datenstrukturen. Sie bieten die Möglichkeit komplexe Daten vereinfacht darzustellen. Sie werden meistens in Diagrammen als Bäume dargestellt. Knoten werden durch einen Kreis, ein Rechteck oder eine Ellipse repräsentiert und die Kanten durch Linien. Diese Darstellungen verschwendet jedoch im oberen Bereich des Diagramms viel Platz da sich die Anzahl der Kinder mit der Tiefe des Baumes erhöht und diese im unteren Teil abgebildet werden müssen.

Eine Alternative die eine besser verteilte Darstellung von Hierarchien bietet ist das Kacheldiagramm, das auch als Tree Map bezeichnet wird. Die hierarchische Struktur wird in der Tree Map durch ineinander verschachtelte Rechtecke abgebildet. Die Fläche der darzustellenden Dateneinheit ist proportional zu ihrer Größe. Im einfachsten Fall meint das die Summe aller im Teilgraphen enthaltenen Elemente.

## 2.2 Dashboards

Das Wort Dashboard kommt aus dem Englischen und bedeutet übersetzt *Armaturenbrett*. Der Begriff ist in der Literatur im Zusammenhang mit Informationstechnologie nicht klar definiert. Eine Eigenschaft, die sich jedoch häufig finden lässt, ist die grafische Darstellung von Kennzahlen in Form von Ampeln oder Instrumenten, die denen in einem Flugzeugcockpit ähneln. Eine allgemeine Definition findet sich in dem Buch von Few (2006):

> Ein Dashboard ist eine visuelle Darstellung der wichtigsten Informationen, die nötig sind, um ein oder mehr Ziele zu erreichen, zusammengefasst und angeordnet auf einem Bild, sodass die Information auf einen Blick erfasst werden kann.
>
> (Few, 2006)

Ein Dashboard ist meistens auf einen bestimmten Arbeitsplatz oder eine Personengruppe zugeschnitten. Die relevanten Informationen werden aufgabenangemessen aggregiert und visualisiert, sodass ein Anwender wesentliche Fakten schnell erfassen kann.

### Kategorisierung von Dashboards

Die Einteilung von Dashboards in Kategorien kann anhand einer Vielzahl von Kriterien erfolgen. Dazu zählen der Aufgabenbereich der zugrunde liegenden Daten, das Einsatzgebiet und viele weitere Eigenschaften. Tabelle 2.6 zeigt die von Few (2006) aufgelisteten Merkmale und ihre möglichen Ausprägungen.

In der Praxis wird die grundlegende Ausrichtung meistens durch das Aufgabengebiet und den Zweck festgelegt, wodurch sich wiederum die Ausprägungen der anderen Kriterien ergeben. Man unterscheidet in *strategische Dashboards*, *operative Dashboards* und *analytische Dashboards*.

### Strategische Dashboards

Am meisten verbreitet sind die *strategischen Dashboards*, die auch „executive dashboards“ genannt werden. Sie erlauben einen schnellen Überblick über die Zielerreichung durch verdichtete Kennzahlen auf relativ hohem Abstraktionsniveau. Die Zielgruppe sind vornehmlich Manager oder andere Entscheidungsträger eines Unternehmens. Klare, einfache Visualisierungen stellen die wichtigen Informationen dar, die in Bezug zu Vorgabezielen und Ergebnissen vorhergehender Messungen gesetzt werden. Die Möglichkeit zur Interaktion ist sehr begrenzt oder nicht vorhanden, da tiefer gehende Detailinformationen für einen Entscheidungsträger nicht von Interesse sind und von den wesentlichen Zielen ablenken. Diese Art von Dashboards erfordert keine Verarbeitung von Echtzeit-Daten,

| Merkmal | Ausprägung |
|---|---|
| Aufgabenbereich | Strategisch |
| | Taktisch |
| | Operational |
| Datentyp | Qualitativ |
| | Quantitativ |
| Einsatzgebiet | Vertrieb |
| | Finanzen |
| | Marketing |
| | Personal |
| | Produktion |
| Messgrößen | Allgemeines Kennzahlensystem |
| | Balanced Scorecard |
| | Spezifisches Kennzahlensystem |
| Datenspanne | Unternehmensweit |
| | Abteilungsweit |
| | Individuell |
| Aktualisierungshäufigkeit | Monatlich |
| | Wöchentlich |
| | Täglich |
| | Stündlich |
| | Echtzeit oder fast Echtzeit |
| Interaktionsdesign | Statische Anzeige |
| | Interaktiv (Navigation, Filtern etc.) |
| Visualisierungsform | Vorwiegend Text |
| | Vorwiegend Grafik |
| | Integration von Grafik und Text |
| Portalfunktionalität | keine Portalfunktionalität |
| | Portalfunktionalität |

Tabelle 2.6: Merkmale und mögliche Ausprägungen von Dashboards

sondern basiert auf Informationen, die in regelmäßigen Abständen zu einem bestimmten Zeitpunkt aus dem System extrahiert wurden.

### Analytische Dashboards

Dashboards, die zur Analyse von Daten und Sachverhalten dienen, gehören zu den *analytischen Dashboards*. Sie erfordern bei der Darstellung von Informationen einen höheren Detailgrad. Damit Zusammenhänge deutlicher werden, verfügen sie über ergiebige Vergleiche, feingranulare Statistiken über bestimmte Zeiträume und komplexe Funktionen zum Auswerten von Resultaten. Die Datenbasis besteht aus Ergebnissen der statischen Analyse. Das Ziel von *analytischen Dashboards* ist es, Trends auszumachen sowie versteckte Problemursachen aufzudecken. Die Benutzungsschnittstelle ist anspruchsvoller als die eines *strategischen Dashboards* und erfordert mehr Einarbeitungsaufwand, da sie komplexere Datenstrukturen darstellt. Außerdem ermöglicht die Oberfläche verschiedene Formen der Interaktion, die zur Untersuchung von Detailinformationen nötig sind, wie z.B. die Navigation in hierarchischen Daten (*drill-down*). Dadurch kann ein Anwender die Ursachen für ein Problem nachvollziehen, und korrigieren.

### Operative Dashboards

Die dritte Kategorie bilden die *operativen Dashboards*. Sie bewerten und visualisieren die operativen Prozesse in einem Unternehmen und ermöglichen das Steuern und Überwachen der täglichen Aufgaben. Der Status von Unternehmensabläufen wird auf zeitnah erhobenen Daten analysiert und auf Abweichungen vom Normalfall untersucht. Durch die Informationen in Echtzeit oder „fast“ Echtzeit können Entscheidungsträger schnell reagieren und steuernd in den Prozess eingreifen.

## 2.2.1 Software-Dashboards

Auch in der Softwareentwicklung und Wartung ist ein Instrument zur ganzheitlichen Kontrolle und Steuerung der einzelnen Entwicklungsschritte und Resultate notwendig. In der Literatur wird diese Art von Werkzeugen auch als Software-Leitstand, Software-Dashboard oder Projektleistand-Cockpit bezeichnet. Rötschke (2006) definiert Software-Dashboards als „komplexe Werkzeuge zur Qualitätssicherung, die es erlauben, Daten über Software-Systeme zu sammeln und diese in geeigneter Form für verschiedene Anwender darzustellen“.

Das Hauptziel von Software-Dashboards ist es also, allen Projektbeteiligten kontinuierlich die wesentlichen Informationen über den Qualitätszustand eines Softwareproduktes zur Verfügung zu stellen. Projektbeteiligte Personen sind dabei Entwickler, Architekten, Projektleiter und Qualitätsverantwortliche. Neben den inneren Qualitätseigenschaften (wie Wartbarkeit und Strukturiertheit) spielen ebenso externe Qualitätseigenschaften (wie

Performanz und Funktionsabdeckung) sowie prozessbezogene Fortschrittseigenschaften (wie Budget-und Zeiteinhaltung) eine Rolle bei der Analyse von Softwaresystemen (siehe Abschnitt 2.1.2).

Da die Software-Qualität einen entscheidenden Einfluss auf die Produktivität der Wartung und Weiterentwicklung von Softwaresystemen hat, bietet die Sicherstellung der Qualität eine wesentliche Möglichkeit zur Kostenersparnis durch Verbesserung der Produktivität der Wartung (Deißenböck und Seifert, 2006). Da die Weiterentwicklung und Wartung von Softwaresystemen zu einem Verfall der Qualität führt (Eick u. a., 2001), wächst in den Unternehmen der Bedarf an unterstützenden Werkzeugen zur Kontrolle von relevanten Qualitätskriterien. Mit Hilfe einer zeitnahen und kontinuierlichen Überwachung helfen Software-Dashboards Probleme und Fehler während des Entwicklungsprozesses möglichst früh aufzudecken und zu beheben. Der Begriff *Monitoring* (engl. Beobachtung) beschreibt dabei die systematische Erfassung eines Prozesses oder eines Vorgangs. Das Ziel ist es dabei, in den Prozess oder Vorgang steuernd einzugreifen, falls dieser bestimmte Schwellwerte über- oder unterschreitet.

### 2.2.2 Aufbau von Software-Dashboards

Abbildung 2.10 zeigt ein allgemeines Prinzip von Software-Dashboards, auf dem die meisten existierenden Ansätze beruhen. Die Basis bildet eine Datenbank, in der Informationen über das Softwaresystem zentral abgelegt werden können. Die wichtigen Informationen über das Softwareprodukt werden parallel zur Entwicklung zu definierten Zeitpunkten extrahiert und zu einem konsistenten Datenbestand integriert. Zu diesen Informationen zählen z.B. Modelle und Quelltexte. Im nächsten Schritt werden unwichtige Daten herausgefiltert und Detailinformationen aggregiert. Anhand von vordefinierten Regeln können anschließend die Ergebnisse bewertet werden. Hierzu zählt auch die Überwachung von Schwellwerten. Die Darstellung erfolgt in angemessener Form, je nach Rolle und Informationsbedarf des Betrachters. Die Daten werden auf verschiedenen Aggregationsebenen angeboten, durch die der Anwender navigieren kann.

Im Gegensatz zu Werkzeugen, die nur die Analyse und Darstellung eines Software-Systems zu einem bestimmten Zeitpunkt als Ziel haben, schließen Software-Dashboards außerdem noch die Betrachtung des Verlaufs von Messgrößen über einen gewissen Zeitraum mit ein. Dadurch lassen sich z.B. Trends erkennen und vorhersagen.

### 2.2.3 Funktionalität von Software-Dashboards

Die von Giesecke u. a. (2006) entwickelte Taxonomie für Software-Dashboards unterscheidet den Betrieb und die Entwicklung von Software und definiert entsprechend Software-Entwicklungs-Dashboards und Software-Betriebs-Dashboards. In der Taxonomie wird anstelle des Begriffs Software-Dashboard, Software-Leitstand verwendet, der nachfolgend aus Gründen der Konsistenz ersetzt worden ist. Für die Kategorie der

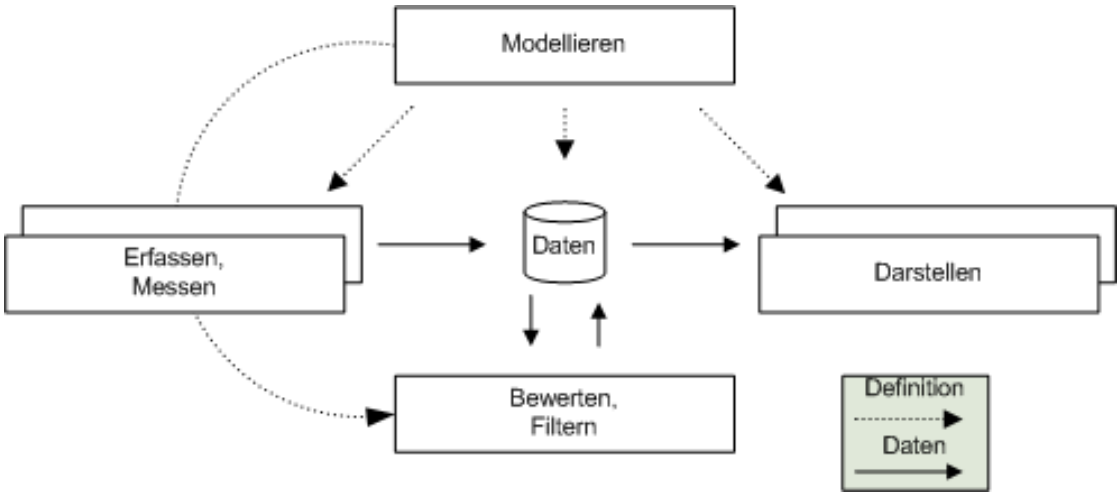


Abbildung 2.10: Grundlage von Software-Dashboards

Software-Betriebs-Dashboards beschreiben die Autoren ein Schichtenmodell der Funktionalität. Da aufgrund der besonderen Eigenschaften von Software jedoch eine scharfe Trennung von Entwicklung und Betrieb nicht sinnvoll ist, lässt sich ihr Schichtenmodell ebenso auf einen Software-Dashboard übertragen, das die Funktionalität beider Arten in sich vereinigt.

Die Funktionalitäten werden unterschieden in Basis-Funktionalität und Erweiterungs-Funktionalität. Ein System muss mindestens die Basis-Funktionalität zur Verfügung stellen, damit von einem Dashboard gesprochen werden kann. Zu dieser Gruppe gehören die Erfassung und die Visualisierung. Die Gruppe der optionalen Erweiterungen umfasst mindestens die Funktionalität der Überwachung, der Beeinflussung und der Alarmierung.

**Erfassung:** Die Daten müssen vom System erfasst, aufbereitet und aggregiert werden. Es soll ein Zugriff auf historische Daten ermöglicht werden.

**Visualisierung:** Die erfassten Messdaten werden mit einem Architekturmodell des zu untersuchenden Systems in Verbindung gebracht und dargestellt.

**Beeinflussung:** Das System bietet Möglichkeiten zur Beeinflussung der Systemeigenschaften, wie z.B. Reparatur und Rekonfigurationsoperationen.

**Überwachung:** Kritische Systemzustände können nach vordefinierten Regeln erkannt werden. Die Erkennung von Anomalien unterstützt den Prozess der Fehlerdiagnose und Reparatur.

**Alarmierung:** Mit Hilfe der Überwachungsfunktionalität können anwesende oder auch nicht anwesende Administratoren auf kritische Systemzustände aufmerksam gemacht werden.

Das Übertragen von Dashboard-Lösungen aus anderen Unternehmensbereichen ist nicht ohne weiteres möglich. Wichtig bei der Einführung von Software-Dashboards ist die Festlegung von Messgrößen und die Ableitung geeigneter Maße zur Erfassung der Daten, wie es bereits in Abschnitt 2.1.2 beschrieben wurde. Als Voraussetzung muss herausgefunden werden, welche Daten zur Qualitätssicherung in einem Unternehmen bereits erhoben werden und wie diese Daten zu den Messzielen beitragen können.

## 2.3 Portale

Bei einem Dashboard steht die reine Auswertung und Darstellung von Fakten und relevanten Kennzahlen im Vordergrund. Im Gegensatz dazu verfolgt der Portal-Ansatz das Ziel, unterschiedliche Inhalte unter einer gemeinsamen Oberfläche zu integrieren. Portale bieten einen zentrale Zugang zu Informationen und Diensten. Verschiedene Anwendung werden über eine Portalplattform zusammengeführt und können über die zentrale Benutzeroberfläche bedient werden. Da zur Nutzung aller verfügbaren Dienste nur eine einmalige Anmeldung des Nutzers an der Portalsoftware nötig ist, spricht man vom Prinzip des „Single Sign-On“ (Einmalanmeldung).

Portale sind wie Dashboards Personengruppen- oder Arbeitsplatzgebunden, bieten jedoch umfangreichere Navigationsmöglichkeiten zur Untersuchung des Datenbestandes auf verschiedenen Detailebenen. Die Möglichkeit zur Personalisierung der Oberfläche ist ein weiteres Merkmal von Portalen. Das bedeutet, dass der Nutzer das Portal an seine persönlichen Vorlieben und Präferenzen anpassen kann. Portale basieren meistens auf Web-Technologien, da zur Darstellung nur ein Web-Browser erforderlich ist.

### Portalarchitektur

Im Folgenden soll die von Großmann und Koschek (2005) entwickelte fachliche Referenzarchitektur vorgestellt und erläutert werden. Die Architektur beschreibt ein idealtypisches Modell der fachlichen Ebene der Portalanwendung. Sie identifiziert die fachliche Funktionalität und die Zusammenhänge des Anwendungsbereiches.

### Integrationskomponente

Die Hauptaufgabe des softwaregestützten Informationsmanagements ist es, die im Kontext relevanten Informationen zu ermitteln und diese dem Nutzer zu präsentieren. Die Schwierigkeit liegt hierbei in der Heterogenität der Unternehmensdaten, da sich diese im Bezug auf ihren Strukturierungsgrad, ihres Speicherortes, ihres Informationsgehalts und ihrer Relevanz unterscheiden.

Die Aufgabe der Integrationskomponente ist es, von den konkreten Datenquellen zu abstrahieren und somit eine virtuelle zentrale Informationsquelle darzustellen. Sie implementiert

einen zentralen Datenkatalog für alle angeschlossenen Datenquellen, Anwendungen und Dienstanbieter. Die Integrationskomponente bildet eine einheitliche Zugriffsschicht, mittels der die Komponenten der Portalanwendung auf die Informationen zugreifen können. Die Metadaten enthalten Beschreibungen über die Zusammenhänge zwischen Daten aus mehreren technischen Datenquellen, sodass diese in Bezug zueinander gesetzt werden können.

### Prozesskomponente

Die Prozesskomponente hat die Prozessautomatisierung und Prozessunterstützung als Aufgabe. Zum Erreichen von definierten Zielen koordiniert sie Benutzer und Anwendungen. Weiterhin legt sie fest, welche Rollen an einem Arbeitsschritt beteiligt sind, wie die weiteren Arbeitsschritte aussehen und wie der Arbeits- und Datenfluss für einen Geschäftsprozess ablaufen soll. Die technische Unterstützung von Prozessen ist nur dann effektiv möglich, wenn sie nach fest definierten Regeln ablaufen, sich häufig wiederholen und aus mehreren Arbeitsschritten bestehen. Weiterhin sollten sie vordefinierte Ziele besitzen und mehr als eine Person betreffen.

In der Literatur findet sich für die Prozesskomponente häufig der Begriff Workflow-Management-System. Als Workflow werden dabei die technisch unterstützen Aktivitäten des Geschäftsprozesses bezeichnet. Die Workflowsysteme unterstützen meistens die Modellierung von Geschäftsprozessen mit Hilfe von grafischen Prozessdiagrammen. Die Steuerung der Prozesse erfolgt dann anhand von Steuerungsdiagrammen, die das Workflowsystem aus den Prozessdiagrammen ableitet. Eine Simulationskomponente ermöglicht die Erkennung von Fehler und Schwachstellen in Abläufen. Zur Simulation wird das Verhalten eines Prozesses unter verschiedenen Voraussetzungen getestet unter Berücksichtigung von verschiedenen Aspekten wie z.B. Bearbeitungszeit.

### Portalapplikationen

Als Portalapplikationen bezeichnet man die webbasierten Präsentationskomponenten eines Portals. Sie sind für den Zugriff auf die Informationssysteme eines Unternehmens zuständig. Die Portalapplikationen oder sogenannten Portlets modellieren zum einen die Basisfunktionalität eines Portals wie z.B. die Suche oder die Anmeldung, und zum anderen bilden sie die unternehmensspezifischen Fachfunktionen ab.

Ein Portlet-Fenster besteht aus einer Titelleiste und einem Rahmen, der eine Schaltfläche zum Schließen enthält. Portlets verarbeiten die Benutzeranfragen indem, sie die Daten der angeforderten Portalkomponente übergeben und anschließend die erhaltenen Ergebnisse in einem Portlet-Fenster darstellen.

### Präsentationskomponente

Die Präsentationskomponente ist für die Präsentation der Daten zuständig. Sie sollte flexibel sein, um die Daten in Abhängigkeit der Rolle, des Kontexts, der persönlich Präferenzen und der Relevanz präsentieren zu können. Bei Portalsystemen teilt sich die grafische Oberfläche meistens in eine Inhalts- und eine Navigationskomponente. Die Inhaltskomponente ist aus Portlets zusammengesetzt (siehe Abschnitt 2.3 Portalapplikationen). Die Rolle eines Benutzers legt fest, welche Portlets von ihm verwendet werden dürfen. Die Inhalte und die Darstellung der Portalapplikationen lassen sich personalisieren, wodurch der Nutzer die Möglichkeit hat, das Portalsystem an seinen Arbeitsablauf und seine persönlichen Präferenzen anzupassen.

Die Präsentationskomponente ist weiterhin für die Auswahl und Anordnung der Portlets auf der Portalseite verantwortlich sowie für die Speicherung der persönlichen Einstellungen der Benutzer. Stehen mehrere Portlets in Abhängigkeit zueinander und müssen über Zustandsänderungen informiert werden, übernimmt die Präsentationskomponente die Benachrichtigung der relevanten Applikationen.

### Business Intelligence

Der Begriff Business Intelligence bezeichnet Systeme, die Unternehmensdaten sammeln, auswerten und darstellen, mit dem Ziel, die strategische Entscheidungsfindung zu unterstützen. Außerdem dienen die Systeme zur Überwachung und Optimierung der Geschäftsprozesse. Die Komponente der Business Intelligence berechnet aus den Daten Kennzahlen in Echtzeit oder stellt sie in Berichten zur Verfügung.

### Knowledge Management

Aufgabe des Knowledge Managements ist die Erschließung, Verwaltung und Recherche des Unternehmenswissens. Das Ziel ist es, die Informationen organisiert und strukturiert verfügbar zu machen. Das implizite Wissen, welches nur in den Köpfen der Mitarbeiter eines Unternehmens vorhanden ist, soll in explizites Wissen, also dokumentierbares und allgemein zugängliches Wissen, überführt werden. Die kontextabhängige Informationsrecherche und -präsentation konzentriert die Daten auf das Wesentliche und hilft dem Anwender bei der Suche nach Informationen. Die Knowledge Management Komponente innerhalb der Portalanwendung dient zur Informationsrecherche und erhöht die Qualität der Suchergebnisse.

### Benutzerverwaltung

Die verschiedenen Rollen der Benutzer des Portalsystems beschreiben fachliche Aufgaben und legen Zugriffsrechte für die verfügbaren Informationen fest. Die Nutzer des Systems

verfügen über autorisierte Benutzerkonten, denen jeweils eine oder mehrere definierte Rollen zugewiesen werden können. Die Benutzerverwaltung übernimmt dabei die Pflege der Benutzerkonten mit den dazugehörigen Rollen.

### Sicherheit

Je öffentlicher der Zugang zum Portal gestaltet ist, desto wichtiger ist der Aspekt der Sicherheit. Eine Anwendung die z.B. über das Internet erreichbar ist, erfordert entsprechende Sicherheitsmaßnahmen, um die verfügbaren Daten zu schützen. Zu den Anforderungen an die Sicherheit gehören sowohl die Authentifizierung, die Autorisierung sowie die Sicherung der Kommunikation durch Verschlüsselung.

### Programmierschnittstelle und -werkzeuge

Zur Erstellung von eigenen Portalapplikationen (siehe Abschnitt 2.3 Portalapplikationen) muss das Portalsystem eine Programmierschnittstelle zur Verfügung stellen. Diese sollte die Erweiterung der Portalkomponenten und Standardprozesse ermöglichen, sodass z.B. die Datenbasis um eigene Anforderungen ergänzt werden kann. Auch die internen Prozesse der Portalanwendung sollten soweit zugänglich sein, dass bei der Weiterentwicklung eine Fehlersuche oder Performanzoptimierungen erleichtert werden.

## 2.4 Verwandte Arbeiten

### Sotograph

*Sotograph* ist ein kommerzielles Produkt der Software Tomography GmbH[1]. Das Ziel von *Sotograph* ist die automatisierte Analyse von Softwaresystemen hinsichtlich ihrer inneren Code- und Architekturqualität. Durch einen Vergleich von Soll und Ist-Architektur eines Systems kann validiert werden, in wie weit die Architekturvorgaben in der Entwicklung eingehalten werden. Die Architekturvorgabe muss jedoch manuell erstellt und überprüft werden, sodass lediglich der Vergleich beider Architekturen automatisiert abläuft.

Der Funktionsumfang von *Sotograph* umfasst unter anderen Metrikanalyse, Zyklenerkennung, Klonerkennung sowie die Trendanalyse einer Architektur.

Derzeit unterstützt *Sotograph* die Analyse von Programmen, die in den Sprachen C, C++ oder Java geschrieben wurden. Die Extraktion der Daten erfolgt aus Quellcode oder aus Bytecode. Bei der Analyse erkennt *Sotograph* Programmstrukturen wie z. B. Klassen oder Pakete und identifiziert Abhängigkeiten zwischen ihnen. Diese Informationen werden nach Abschluss der Analyse in einer *MySQL* Datenbank gespeichert.

[1] http://www.software-tomography.com

Die durchgeführten Analysen sind statisch, d.h. die Architekturvalidierung findet zu einem bestimmten Zeitpunkt statt. Die Ergebnisse der Analysen können mit Hilfe der Erweiterungen *Sotoweb* und *Sotoreport* aggregiert und ausgegeben werden.

### ConQAT

*ConQAT*[1] (Continous Quality Assessment Toolkit) wurde als Forschungsprojekt von der TU München entwickelt und bietet ein Rahmenwerk für die Erstellung von Leitständen zur kontinuierlichen Qualitätsüberwachung von Software. *ConQAT* basiert auf einem Kompositionsmechanismus, der die Integration von verschiedenen Analysewerkzeugen ermöglicht. Dadurch können neben dem Quellcode noch weitere Systemartefakte zur Analyse eines Programms herangezogen werden.

*ConQAT* unterscheidet sich von anderen Ansätzen dadurch, dass die zu analysierende Software nicht in eine durch ein Metamodell vorgegebene Standardform überführt wird. Die erhobenen Daten werden aggregiert und anschließend über einen Webserver im HTML Format ausgegeben. Der Zusammenhang zwischen der aggregierten Sicht und der Detailinformation wird durch Querverweise hergestellt. (Deißenböck und Seifert, 2006)

### MOOSE

*MOOSE*[2] ist ein Forschungsprojekt, das 19997 von der Software Composition Group (SCG) der Uni Bern gegründet wurde. Inzwischen sind weitere Universitäten in ganz Europa an der Weiterentwicklung beteiligt. *MOOSE* ist ein Re-Engineering Werkzeug, das ein Rahmenwerk zur Integration von verschiedenen Werkzeugen zur Verfügung stellt. Dadurch ist *MOOSE* nicht allein auf die Analyse und Visualisierung von Softwaresystemen begrenzt, sondern es kann um weitere für das Re-Engineering nützliche Funktionalität, wie z.B. das Refaktorieren erweitert werden.

Der Ansatz von *MOOSE* basiert auf einem sprachunabhängigen Metamodell, welches die Elemente der zu analysierenden Software (Klassen, Pakete) und ihre Abhängigkeiten repräsentiert. Die Extraktion der Daten erfolgt aus Quellcode. Zu den unterstützen Sprachen gehören Java, C++, SMALLTALK und COBOL. (Ducasse u. a., 2005)

Durch die Integration von verschiedensten Werkzeugen hat sich der Funktionsumfang von *MOOSE* seit der Entstehung kontinuierlich vergrößert. Exemplarisch werden an dieser Stelle nur ein paar Beispiele genannt. Zu den Funktionen zählt die Klonerkennung, Analyse von Informationen von Versionsverwaltungssystemen, dynamische Programmanalyse, Abhängigkeitsanalyse, Evolutionsanalyse und weitere.

[1] http://conqat.cs.tum.edu
[2] http://moose.unibe.ch/

## 2.5 Structured Query Language

Die *Structured Query Language*, kurz *SQL*, ist eine Datenbanksprache, die sämtliche Aspekte des Datenbankzugriffs abdeckt. Dazu gehören die Datenmodifikation (*Data Modification Language*, *DML*), die Datendefinition (*Data Definition Language*, *DDL*) und die Rechteverwaltung (*DCL*, *Data Control Language*). *SQL* ist die am häufigsten in relationalen Datenbanken implementierte Datenbanksprache und ist von *ISO* und *ANSI* in mehreren Versionen standardisiert. Im Wesentlichen richten sich die gängigen Datenbanksysteme nach dem *SQL-92* Standard ISO/IEC 9075 (1992). Die aktuelle Version des Standards ist aus dem Jahr 2006, wird jedoch von den wenigsten Datenbanksystemen umgesetzt.

# 3 Analyse

In diesem Kapitel soll untersucht werden, welche Möglichkeiten Bauhaus zur Qualitätssicherung bietet und wie die Ergebnisse der einzelnen Analyse-Werkzeuge visualisiert werden. Nach einer allgemeinen Einführung in Bauhaus in Abschnitt 3.1, werden die Zwischenformate beschrieben, die die Grundlagen der Analysen bilden. Die grafische Oberfläche *GraVis* wird in Abschnitt 3.1.3 erläutert, bevor in Abschnitt 3.2 eine Bewertung der Werkzeuge von Bauhaus vorgenommen wird.

## 3.1 Bauhaus

Gegründet wurde Bauhaus 1996 vom Fraunhofer Institut für Experimentelles Software-Reengineering und der Universität Stuttgart. Als gemeinsames Forschungsprojekt wird es inzwischen an den Universitäten Bremen und Stuttgart weiterentwickelt und durch die als Ableger entstandene Axivion GmbH kommerziell vermarktet.

Das Ziel von Bauhaus ist es, einem Wartungsprogrammierer beim Analysieren und Verstehen von Software zu helfen, wodurch sich seine Arbeit erleichtert und er in seinen alltäglichen Aufgaben unterstützt wird. Bauhaus bietet hierfür eine Reihe von Quellcodeanalysen, wie z.B. Metrikerhebung, Klonerkennung, Architekturvalidierung und Erkennung von Zyklen im Aufrufgraph.

Damit ein System analysiert werden kann, muss es in eine Zwischendarstellung überführt werden. Dies geschieht in Bauhaus mit Hilfe des Analyse-Tools *cafeCC*, welches aus dem Quellcode die interne Zwischendarstellung *Intermediate Language (IML)* erzeugt. Die IML ist eine sprachunanbhängige Darstellung, wodurch sich die Wiederverwendbarkeit vorhandener Analysen erhöht und die Integration weiterer Analysen vereinfacht wird. Ein Programm wird in der *IML* sehr feingranular repräsentiert, da das gesamte System, einschließlich aller seiner Programmkonstrukte enthalten ist.

Für weitere Analysen kann aus der *IML* ein *Resource Flow Graph* erzeugt werden, der neben der *IML* eine weitere Zwischendarstellung in Bauhaus bildet. Im *RFG* wird von den Detailinformationen weitgehend abstrahiert, sodass nur noch die globalen Elemente eines Programms, deren Attribute und ihre Abhängigkeiten repräsentiert werden. Der

*RFG* bildet die Basis für die Visualisierungen in *GraVis* der grafischen Oberfläche von Bauhaus.

### 3.1.1 Intermediate Language

Die *IML* ist eine graphbasierte Darstellungsform, die der eines annotierten abstrakten Syntaxbaumes ähnlich ist. Durch die Erweiterung um semantische Kanten wird aus dem Baum allerdings ein Graph, weshalb die *IML*-Repräsentation eines Programms auch als *IML*-Graph bezeichnet wird.

Die Knoten in der *IML* sind von einem bestimmten Typ und besitzen Attribute. Die Attribute sind entweder Werte oder Verweise auf andere Knoten, welche als gerichtete Kanten bezeichnet werden. Bei der Generierung der *IML* werden die klassischen Analysen eines Compilier-Frontends angewendet. Zuerst wird der Quelltext in einen Tokenstrom überführt und anschließend mithilfe eines Parsers in einen abstrakten Syntaxbaum (*AST*) umgewandelt. Im nachfolgenden Schritt wird der Graph um semantische Informationen erweitert, welche als Kanten repräsentiert sind. Die semantischen Kanten beinhalten Informationen wie Datentypen oder Sprungziele, können aber ebenso dazu verwendet werden, beliebige andere Informationen zu speichern.

### 3.1.2 Ressource Flow Graph

Der *RFG* ist die zweite interne Zwischendarstellung in Bauhaus und bildet ein Programm im Vergleich zur *IML* auf einer höheren Abstraktionsebene ab. Mit Hilfe der grafischen Oberfläche *GraVis* kann ein *RFG* visualisiert und bearbeitet werden. Die Generierung eines *RFGs* erfolgt entweder aus der *IML*, wie für die Programmiersprachen Ada, C, C++ oder Java oder direkt aus dem Bytecode, wie es für .NET oder Java der Fall ist.

Der *RFG* ist ein hierarchischer Graph und besteht aus typisierten Knoten und Kanten. Die Knoten repräsentieren architekturrelevante Elemente wie z.B. Dateien, Routinen oder Typen. Die Abhängigkeiten zwischen ihnen werden durch Kanten abgebildet. Das dynamische Modell des RFG erlaubt es, Attribute und Typen frei zu definieren und diese zur Laufzeit zu verwenden. Knoten- und Kantentypen sind frei typisierbar, wo hingegen Attribute von einem der fünf vorgegebenen Datentypen boolscher Wert, Togglewert, Integerzahl, Fließkommazahl oder String sein müssen. Die Attribute repräsentieren z.B. Quelltextverweise oder Metriken.

Um die Menge der im *RFG* enthaltenen Informationen zu strukturieren, gibt es das Konzept der Sichten (*View*). Eine *View* besteht aus einer Teilmenge der Knoten und Kanten des *RFGs* und stellt bestimmte Aspekte des Gesamtsystems dar. Außerdem ermöglichen die *Views* die Durchführung von Analysen auf einem Teilgraphen. Das Ergebnis einer Analyse wird wiederum in einer neu erzeugten *View* dargestellt.

Zur Erstellung eines *RFGs*, müssen für C oder C++ zuerst die Quelltextdateien dem Tool *cafeCC* übergeben werden, das für die einzelnen Übersetzungseinheiten mehrere *IML*-Dateien generiert. Um die Darstellung des gesamten Programms zu erhalten, müssen die einzelnen Dateien mit Hilfe des Tools *imllink3* zu einer *IML* zusammengelinkt werden. Die Überführung der *IML* in einen *RFG* ist Aufgabe des Programms *iml2rfg*.

### 3.1.3 GraVis

Die grafische Oberfläche *GraVis* bietet verschiedene Möglichkeiten, um einen *RFG* anzuzeigen, zu bearbeiten und zu analysieren. Nach dem Start von *GraVis* öffnet sich das Hauptfenster, die sogenannte *workbench*, wie in Abbildung 3.1 zu sehen ist. In der Mitte des Fensters sind die *Views* aufgelistet, die in dem geladenen *RFG* enthalten sind.

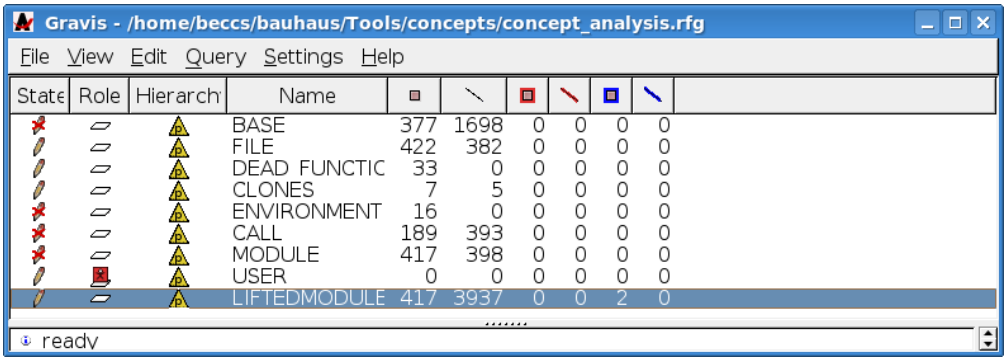


Abbildung 3.1: Das Hauptfenster von *GraVis*

Zu jeder *View* werden ihr Name, ihr Status, die Anzahl von Knoten und Kanten, sowie Informationen zur Rolle und zur Hierarchie angezeigt. Ist einer *View* eine Rolle zugewiesen, werden bestimmte Vorgabewerte bei der Architekturanalayse gesetzt. Die Hierarchiespalte gibt an welcher Kantentyp zur hierarchischen Darstellung verwendet werden soll. Eine *View* kann als schreibgeschützt gekennzeichnet werden, sodass sie nicht verändert werden kann. Ob eine *View* modifizierbar ist, zeigt die Spalte *State* am linken Rand des Fensters.

Ein Doppelklick auf den Namen einer *View* öffnet ein neues Fenster, das sogenannte *Graph Window*, das den Inhalt der *View* hierarchisch angeordnet darstellt. Dazu gehören die Knoten und Kanten sowie deren Attribute. Die unterschiedlichen Knotentypen werden durch verschiedene Formen und Farben repräsentiert. Ein Dateisymbol stellt beispielsweise ein Modul eines Systems dar. Die Farbe der Kanten ist abhängig von dem dazugehörigen Kantentyp. Zusätzlich werden noch einige Attribute der Knoten und Kanten an den entsprechenden Objekten dargestellt, wie beispielsweise die Dateinamen. Bei den Kanten kann über ein Attribut die Stärke der Pfeilline beeinflusst werden.

Die Knoten innerhalb des Fensters können vom Nutzer mit Hilfe der Maus ausgewählt und frei verschoben werden. Beim Auswählen der Knoten und Kanten wird zwischen

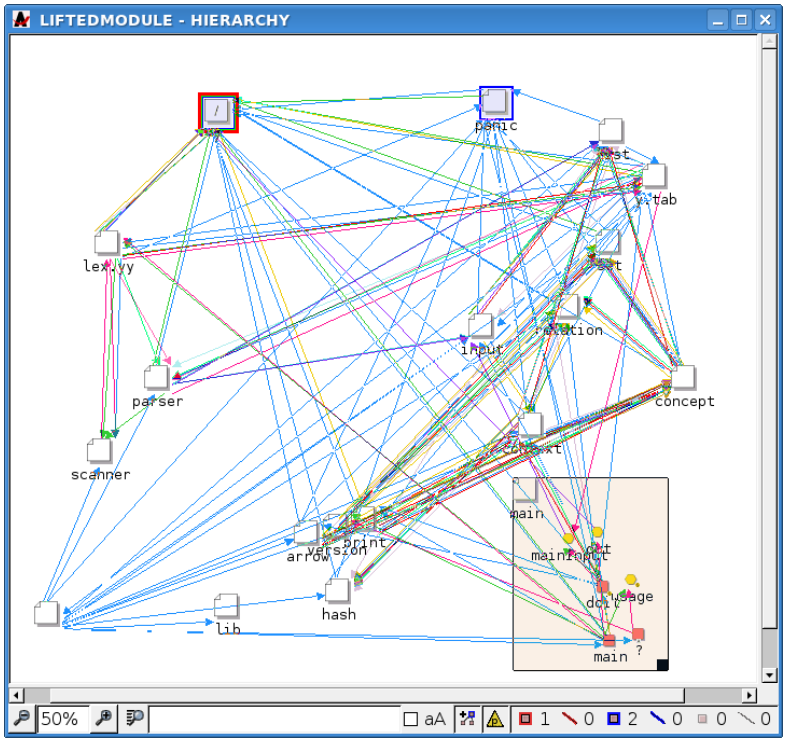


Abbildung 3.2: Das Graphfenster

*markieren* und *selektieren* unterschieden. Eine Selektion wird nur auf Elemente einer *View* angewendet, während eine Markierung in allen *Views* sichtbar ist, in denen das markierte Element vorkommt. Eine blaue Umrandung stellt eine Selektion dar und eine rote eine Markierung, wie Abbildung 3.2 zeigt. Durch einen Doppelklick können Knoten in einer hierarchischen Ansicht geöffnet werden, um die enthaltenen Kindknoten anzuzeigen. Diese werden in einem Rahmen dargestellt, wie es in der Abbildung für den Knoten *main* der Fall ist.

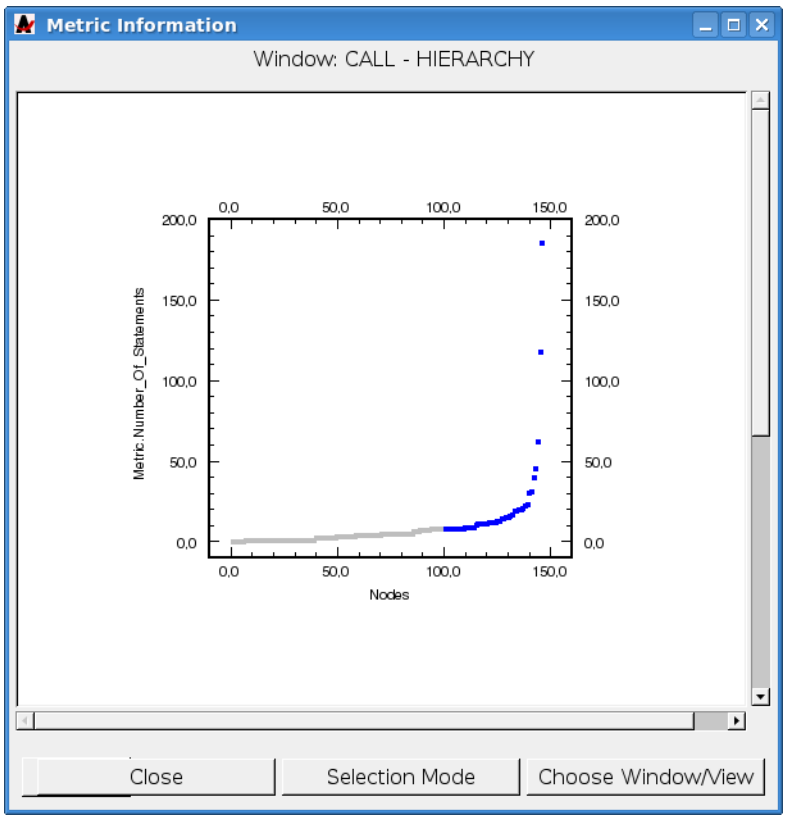


Abbildung 3.3: Das Metrikfenster

Die Analyse von Metrikwerten erfolgt über das sogenannte Metrikfenster, das in Abbildung 3.3 zu sehen ist. Ein Metrikfenster lässt sich zu einer *View* über das Kontextmenü öffnen und ist zunächst leer. Auf der freien Fensterfläche können vom Benutzer definierte Metrikboxen hinzugefügt werden. Eine Metrikbox ist ein zweidimensionales Punktediagramm (Abschnitt 2.1.7) mit dessen Hilfe man Integer- und Floatattribute der Knoten einer *View* visualisieren kann. Das Kontextmenü des Metrikfensters bietet verschiedene Metriken an, die auf den Achsen des Diagramms aufgetragen werden können. In Abbildung 3.3 sind für die Knoten der *View* auf der X-Achse die Anzahl der Anweisungen innerhalb einer Funktion auf der Y-Achse visualisiert.

Zur detaillierten Analyse von auffälligen Werten können die Knoten in dem Diagramm selektiert werden, wodurch sie automatisch auch in der zugrunde liegenden *View* ausgewählt werden. In dieser können dann die Knoten näher betrachtet werden.

### 3.1.4 Analysen

Bauhaus bietet eine Reihe von Quellcodeanalysen, die im Folgenden kurz erläutert werden sollen.

#### Metrikerhebung

Metriken bieten die Möglichkeit, Eigenschaften oder Qualitätsmerkmale von Software quantitativ zu beurteilen. Die von Bauhaus implementierten Analysen erheben Metriken sowohl auf der Ebene des Quellcodes als auch auf Ebene der Architektur. Als Grundlage zur Berechnung der Metriken dienen die beiden Bauhaus-internen Zwischendarstellungen *IML* und *RFG*.

#### Klonerkennung

Als Klon bezeichnet man eine Kopie eines Quelltextabschnittes, die mit kleinen Modifikationen an anderer Stelle wieder eingefügt worden ist. Dadurch erhält man duplizierten Quelltext. Klone erschweren die Wartung von Software, da gefundene Fehler nicht mehr nur an einer Stelle, sondern auch in allen Kopien behoben werden müssen.

Man unterscheidet dabei nach verschiedenen Klontypen. Bei einem *Typ-1* Klon handelt es sich um eine exakte Kopie des Quelltextes. *Typ-2* Klone sind Kopien, bei denen nach dem duplizieren Bezeichner konsistent umbenannt wurden und *Typ-3* Klone weisen weitere Veränderungen am Code auf wie z.B. die Erweiterung einer Funktion.

Bauhaus bietet verschiedene Tools zur Klonerkennung, die entweder auf Basis der *IML* arbeiten, wie die Programme *clones* und *ccdiml*, oder auf einem eigens erzeugten *AST*, wie das Tool *clast*.

#### Zyklische Abhängigkeiten

Zwischen zwei Komponenten eines Softwaresystems liegt eine zyklische Abhängigkeit vor, wenn gilt: $(A \rightarrow B)$ und $(B \rightarrow A)$. Diese Art von Abhängigkeit ist problematisch, da bei der Wiederverwendung einer der Komponenten der gesamte Zyklus verwendet wird. Zyklen im Aufrufgraphen können dazu führen, dass ein Programm bei der Ausführung in einem Zyklus solange hängen bleibt, bis der Arbeitsspeicher voll ist.

Bauhaus ist in der Lage mit Hilfe einer Architekturanalyse, die von *GraVis* aus durchgeführt wird, Zyklen zu erkennen.

## 3.2 Problemdefinition

Nachdem im letzten Abschnitt die verschiedenen Tools von Bauhaus vorgestellt wurden, sollen nun die Schwachstellen sowie die fehlenden Funktionalitäten aufgezeigt werden.

Bauhaus richtet sich hauptsächlich an Wartungsprogrammierer und unterstützt diese bei ihren täglichen Aufgaben. Mit Hilfe der Analysen durch die verschiedenen Tools und der grafischen Oberfläche *GraVis* ist der Wartungsprogrammierer in der Lage, ein Softwaresystem sehr detailliert zu untersuchen. Dadurch lassen sich die Schwachstellen und potentiellen Probleme aufdecken und lokalisieren.

Der Umgang mit *GraVis* und die Nutzung der vielfältigen Möglichkeiten der grafischen Oberfläche erfordern fundiertes Fachwissen über die Komponenten und Abhängigkeiten einer Software. Um die Schwachstellen eines Systems zu erkennen, muss der Anwender zum einen wissen, *wo* er suchen muss, also welche Analysen auf einer bestimmten *View* zum gewünschten Ergebnis führen und zum anderen, wie die Ergebnisse zu interpretieren sind.

Entscheidungsträger eines Unternehmens, wie beispielsweise Manager, verfügen oft nicht über Fachwissen aus dem Bereich der Softwaretechnik. Für sie ist die Überwachung und Bewertung der Qualität des Produktes und der Prozesse in der Wartung relevant. *GraVis* eignet sich hierfür nur bedingt, da entscheidende Funktionalitäten zur Umsetzung dieser Aufgabe fehlen.

### Visualisierung

Eine der Hauptaufgaben von Software-Dashboards im Sinne der Qualitätssicherung ist die angemessene Darstellung von Kennzahlen. Bei *GraVis* werden dafür die Metrikboxen eingesetzt. Sie bieten zwar eine gute Möglichkeit, nach auffälligen Attributwerten von Programmelementen zu suchen, aber die Informationen über den Qualitätszustand eines Systems werden nicht auf einen Blick erkennbar. Die Metrikboxen müssen außerdem vom Anwender nach jedem Programmstart manuell erstellt werden, da sie nicht speicherbar sind. Eine Visualisierung in einer anderen Diagrammart wird nicht unterstützt.

Es kann immer nur ein *RFG* zur Zeit in *GraVis* analysiert werden, sodass der Vergleich bzw. die Analyse von mehreren Projekten nur umsetzbar ist, wenn die grafische Oberfläche mehrfach gestartet wird. Die Metrikboxen lassen sich nur für geöffnete *Views* erstellen, sodass bei zwei Projekten auf dem Bildschirm allein schon sechs Fenster offen sind für den Vergleich der Metrikwerte.

Durch das nach oben Propagieren (*liften*) der Kanten bei hierarchischen Ansichten werden die Informationen im Graphfenster auf verschiedenen Abstraktionsebenen angeboten. Der Nutzer kann durch Öffnen und Schließen der Knoten zwischen diesen Ebenen navigieren. Damit ihm diese Funktionalität zur Verfügung steht, muss der Anwender zuvor die Analyse zum Propagieren der Kanten ausführen.

### Analyse des zeitlichen Verlaufs

Sowohl Bauhaus als auch *GraVis* sind für die Analyse und Visualisierung eines Softwaresystems zu einem bestimmten Zeitpunkt ausgelegt. Um Aufschluss über die Entwicklung eines Systems zu erhalten, müssen die Daten der einzelnen *RFG* Versionen vom Anwender manuell in Beziehung zueinander gesetzt werden.

### Fazit

*GraVis* ist ein Werkzeug, das für Wartungsprogrammierer entwickelt wurde. Diese sollen mit Hilfe der grafischen Oberfläche potentielle Probleme einer Software aufdecken. Da sie über fundierte Fachkenntnisse verfügen, können sie den vollen Funktionsumfang zur Unterstützung ihrer Arbeit verwenden. Für sie sind die Detailinformationen zu einem System wichtiger als eine grobe Übersicht.

Die Menüführung und die Interaktionsmöglichkeiten von *GraVis* sind aufgabenangemessen und richten sich eher nach der Funktionalität als nach der Benutzerfreundlichkeit. Dadurch, das *GraVis* nur für die Benutzergruppe der Wartungsprgrammierer ausgelegt ist, kann man abschließend feststellen, dass die Funktionalität, die ein Dashboard ausmacht noch nicht in Bauhaus enthalten ist. Wie wichtig solch ein Werkzeug jedoch im Zuge der kosteneffizienten Wartung ist, wurde bereits in den vorangegangen Abschnitten erläutert.

# 4 Umsetzung

Dieses Kapitel beschreibt die Lösungsideen und deren Umsetzung. Hierfür werden in Abschnitt 4.1 die Anforderungen an das zu entwickelnde System beschrieben. In Abschnitt 4.2 werden die dazu verwendeten Technologien erläutert. Die Beschreibung der Datenbank findet sich in Abschnitt 4.3. Auf die Architektur und die Struktur des umgesetzten Systems wird anschließend in Abschnitt 4.4 eingegangen.

## 4.1 Anforderungen

Nachdem im letzten Kapitel die Schwachstellen und die fehlende Funktionalität von Bauhaus dargelegt wurden, sollen nun die sich daraus ergebenden Anforderungen für das zu entwickelnde System beschrieben werden.

### 4.1.1 Anwendungsfälle

Die Zielgruppe der Software sind Unternehmen, die in der Software-Entwicklung tätig sind. Die potentiellen Anwender sind alle projektbeteiligten Personen, hauptsächlich jedoch die Entwickler und die Projektleiter. Zur Nutzung des Portalsystems werden deshalb die Rollen *Entwickler* und *Projektleiter* definiert. Diese Rollen können den Mitarbeitern des Unternehmens zugeordnet werden.

Der erste Anwendungsfall der Software ist im Rahmen der Wartung von Software-Systemen zu sehen. Durch Anpassungen oder Fehlerkorrekturen verändert sich die Software im Laufe der Zeit. Die einzelnen Arbeitsschritte innerhalb der Wartung und deren Resultate erfordern eine Kontrolle und Überwachung, um die Qualität der Software sicherzustellen. Ein Projektleiter, der für verschiedene Projekte zuständig ist, muss über den aktuellen Zustand der einzelnen Systeme informiert sein, um bei Problemen steuernd einzugreifen. Die Informationen werden auf einer hohen Abstraktionsebene in verdichteter Form dargestellt. Durch Diagramme erkennt der Projektleiter die wichtigen Fakten auf einen Blick. Dem Projektleiter ist dabei wichtig, auf eventuelle Schwachstellen hingewiesen zu werden, wie beispielsweise die Überschreitung von Schwellwerten. Weiterhin sind Daten

über den Verlauf der Kennzahlen für ihn interessant. Das können entweder normale Statusinformationen sein, die im zeitlichen Verlauf dargestellt werden oder Daten über neu aufgetretene Probleme. Dazu zählen z.B. neue Klone und neue Style-Verstöße. Aufgrund dieser Daten kann der Projektleiter seine Entscheidungen treffen. Weitere Detaillinformationen sind für ihn nicht von Bedeutung. Die Oberfläche soll für den Projektleiter einfach zu bedienen sein und bei Bedarf kann er weitere Analysen und Darstellungsformen auswählen.

Der zweite Anwendungsfall findet sich innerhalb der täglichen Arbeit eines Wartungsprogrammierers. Er muss unter anderem Anpassungen vornehmen, Fehler finden, diese korrigieren und testen. Das Portalsystem kann ihn bei diesen Aufgaben unterstützen, indem es potentielle Probleme auffindet und visualisiert. Der Programmierer kann mit Hilfe der Detailinformationen zu den Problemen, diese lokalisieren und beheben. Außerdem möchte er gerne wissen, welche Verstöße er verursacht hat und für welche Probleme er zuständig ist.

### 4.1.2 Allgemeine Anforderungen

Aus den Anwendungsfällen ergeben sich die allgemeinen Anforderungen, die im Folgenden beschrieben sind. Es soll ein Web-basiertes System entwickelt werden, das die Eigenschaften eines Portals besitzt. Die Software soll über Auswertungs- und Analysefunktionen verfügen, um aus den Daten der zentralen Informationsquelle Kennzahlen zu berechnen. Die Daten sollen dem Nutzer auf verschiedenen Abstraktionsebenen angeboten werden, sodass er von Kennzahlen zu den Detailinformation navigieren kann. Das Portalsystem muss in der Lage sein, den aktuellen Zustand einer zu untersuchenden Software sowie den zeitlichen Verlauf der Kennzahlen abzubilden. Die Systemarchitektur muss so gestaltet sein, dass eine Integration von weiteren Datenquellen einfach umsetzbar ist. Weiterhin soll eine zentrale Zugriffsschicht existieren, die den Zugriff auf angebundene Informationsquellen kapselt, sodass alle Komponenten der Portalanwendung nur über die Zugriffsschicht an die Daten gelangen können. Das System muss verschiedene Rollen und die damit verbundenen Zugriffsrechte abbilden. Die Passwörter der Nutzer müssen verschlüsselt in der Datenbank gespeichert werden. Die Strukturierung der Datenbank soll konform mit dem *SQL*-Standard sein.

#### Personalisierung

Die grafische Oberfläche des Systems soll die Möglichkeit zur Personalisierung bieten. Personalisierung bedeutet, dass sich das Portal nicht für alle Benutzer mit derselben Gestaltung der grafischen Benutzungsoberfläche und demselben Funktions- und Informationsangebot präsentiert. Mit der Rolle eines Benutzers sind Zugriffsrechte auf Informationen und Anwendungen verbunden, sodass nur solche Informationen angezeigt werden, die der Rolle entsprechen und zur Erledigung der jeweiligen Aufgabe dienen. Dadurch sind Daten nur für bestimmte Nutzergruppen sichtbar und damit vor unbefugtem

Zugang geschützt. Weiterhin soll dadurch der Aufwand der Informationssuche für die Anwender minimiert werden, da die Datenmenge durch die Filterung reduziert wird. Die Informationsdarstellung und -aufbereitung erfolgt nach Angaben des Benutzers. Es soll die Möglichkeit geben den Inhalt bestimmter Portlets festzulegen und deren Anordnung innerhalb der Oberfläche nach persönlichen Präferenzen zu gestalten. Dadurch kann die Portaloberfläche so konfiguriert werden, dass sie den persönlichen Vorlieben und der Arbeitsweise des Anwenders entspricht. Die Nutzerangaben müssen gespeichert werden können, sodass sich die grafische Oberfläche beim nächsten Start im gleichen Zustand präsentiert. Diese Anforderung entspricht der im DIN EN ISO 9241 (1996) festgelegten Anforderung der Individualisierbarkeit.

### Web-Anwendung

Das Layout und die Struktur der Seiten sollte getrennt werden, sodass das Layout zentral definiert und bearbeitet werden kann. Die Portalseiten sollen dynamisch erstellt werden und nicht in Form von statischen HTML Seiten vorliegen. Die Daten und Inhalte werden von der Präsentation durch Schablonen (*Templates*) getrennt, die mit dynamisch generierten Daten gefüllt werden.

### Performanz

Die Anforderungen an die Performanz bezieht sich auf die Antwortzeit der Portalanwendung, also die durchschnittliche Reaktionszeit der Portalanwendung auf eine Interaktion. Die Zeit einer Interaktion erstreckt sich vom Auslösen der Aktion durch den Anwender bis zur Anzeige der resultierenden Inhalte auf der Portalseite. Diese ist von einer Vielzahl von Komponenten abhängig, jedoch hat die Größe der Datenbank den stärksten Einfluss auf die Performanz, sodass die Anfragen unter diesem Gesichtspunkt optimal umgesetzt werden müssen. Nach Dahm (2006) gilt für die erwartete Antwortzeit, dass ein Nutzer bis zu einer Sekunde die Antwort als *unmittelbar*, bis zu fünf Sekunden als *verzögert* und bei bis zehn Sekunden als *stark verzögert* empfunden wird.

## 4.2 Verwendete Technologien

Dieser Abschnitt soll die Technologien beschreiben, die bei der Implementierung des Prototypen zum Einsatz kamen. Es werden kurz die Grundlagen zu den einzelnen Technologien erläutert und der Zusammenhang der einzelnen Komponenten dargelegt. Da die Entwicklung des Prototypen in Zusammenarbeit mit der Axivion GmbH erfolgte, wurde die Auswahl einiger Technologien nicht von mir getroffen sondern von der Axivion GmbH. Weiterhin baut eine Webanwendung auf Grundbausteinen auf, dessen Verwendung von mir somit nicht beeinflusst werden konnte. Deshalb wird nur bei einigen der beschriebenen Technologien die Auswahlentscheidung begründet.

## Python

*Python*[1] ist eine frei verfügbare objektorientierte Skriptsprache. Bei der Entwicklung von *Python* wurde viel Wert auf eine einfache Syntax gelegt. Die Standardimplementierung von *Python* wurde in portablem *ANSI C* geschrieben und läuft praktisch auf jeder gebräuchlichen Plattform. Neben dem Sprachinterpreter selber sind auch die Module, die zur Standardbibliothek gehören, so portabel wie möglich implementiert. Dadurch können Programme, die die Kernsprache und die Standardbibliotheken benutzen, plattformunabhängig eingesetzt werden. Die Datentypen in *Python* werden dynamisch verwaltet, d.h. es sind keine Typdeklarationen im Programm notwendig. Die Freigabe von nicht mehr verwendeten Speicherbereichen erfolgt automatisch.

Für die Entwicklung der Software im Rahmen wäre neben *Python* noch die Programmiersprache *Ada95* in Frage gekommen, in der große Teile von Bauhaus implementiert sind. Die Entscheidung fiel auf *Python*, da für den Prototypen eine Reihe weiterer Technologien eingebunden werden mussten und es für *Python* umfangreiche Bibliotheken und Schnittstellen zu diesen Technologien gibt. Ein weiterer Grund war, dass die Entwicklung unabhängig von dem restlichen Bauhaus-Projekt stattfinden konnte und Änderungen im Bauhaus-Projekt keinen Einfluss auf meine Programmierung hatten.

## Cherrypy

Zur Entwicklung des Systems wird *Cherrypy*[2] als Web-Server eingesetzt. *Cherrypy* ist ein in Python geschriebenes Programmiergerüst zur Entwicklung von dynamischen Webanwendungen, ein sogenanntes *Web Application Framework*. Cherrypy kann sowohl als eigenständiger Web-Server eingesetzt werden, als auch über einen bestehenden Server wie z.B. Apache laufen. Das hat den Vorteil, dass die entwickelte Anwendung sehr portabel ist und auf verschiedenen Webservern zum Einsatz kommen kann. Voraussetzung ist hierfür allerdings ein Server, der mit dem Python *Web Server Gateway Interface* (*WSGI*) kompatibel ist, das eine einheitliche Schnittstelle für Web-Server und Webanwendungen definiert.

## Genshi

*Genshi*[3] ist eine Python-Bibliothek zur Erzeugung von Inhalten für Webseiten. Diese Art von Programmen wird *Template Engine* genannt, da sie eine Vorlage (Template) nach einem bestimmten Vorgehen mit Inhalten füllen. *Genshi* ist in der Lage, generierte Inhalte in XML-basierte Sprachen, wie z.B. HTML einzufügen. Eine *Template Engine* bietet den großen Vorteil, dass Inhalte für eine Webseite nicht im Python Programm selber mit String-Verkettung erzeugt werden müssen.

[1] http://www.python.org
[2] http://www.cherrypy.org
[3] http://genshi.edgewall.org

### HTML

*HTML* (hypertext markup language) steht für eine textbasierte Auszeichnungssprache für Dokumente im World Wide Web. Sie dient zur Strukturierung von Inhalten wie Text und Bildern und sie ermöglicht die Definition von Verknüpfung (Hyperlinks) zwischen Dokumenten.

### Cascading Style Sheets

*Cascading Style Sheets* befassen sich mit der Gestaltung von *HTML* Dokumenten. Das Aussehen eines Dokumentes wird dabei in einer separaten Datei gespeichert, wodurch eine Trennung von Struktur und Layout erreicht wird. Weiterhin erhöht sich die Anpassungsfähigkeit des Aussehens an spezielle Präferenzen oder Bedürfnisse.

### JavaScript

*JavaScript* ist eine objektbasierte Skriptsprache, die eine dynamische Gestaltung von Webseiten ermöglicht. Dadurch können *HTML*-Dokumente während der Anzeige im Browser durch Nutzerinteraktion verändert werden. *JavaScript* ist eine clientseitige Skriptsprache, d.h. Skripte, die in *JavaScript* geschrieben sind, werden vom Browser des Benutzers interpretiert und ausgeführt.

### XML

Der Begriff *XML* [4] steht für *Extensible Markup Language*, eine Auszeichnungssprache, die zur Strukturierung von Daten im Textformat verwendet wird. Durch das Definieren von eigenen Elementen und Attributen können verschiedene Informationen, wie beispielsweise Text oder Grafiken beschrieben werden. Mit Hilfe von *XML* werden die Daten von der Darstellung getrennt. So können die Daten aus einem *XML*-Dokument in verschiedenen Formen ausgegeben werden.

### JSON

*JSON*[5] (Java Script Object Notation) ist ein programmiersprachenunabhängiges Textformat zum Austausch von Daten zwischen Anwendungen. Die Syntax ist auf ein Minimum reduziert und sehr kompakt. Dadurch wird das übertragene Datenvolumen gering gehalten.

[4] http://www.w3.org/XML
[5] http://www.json.org

Obwohl *JSON* auf einer Untermenge der Programmiersprache *JavaScript* basiert und häufig in Zusammenhang mit dieser verwendet wird, ist es ein unabhängiges Format, für das Implementierungen in vielen weiteren Sprachen existieren. Unter anderem in C, Java, JavaScript, Python und Smalltalk.

In *JSON* gibt es vier primitive Datentypen (Zeichenketten, Zahlen, boolsche Werte, Null) und zwei Datenstrukturen (Array, Objekte). Eine Zeichenkette ist eine Folge von Zeichen im Unicode Format. Ein Objekt ist eine ungeordnete Liste von Schlüssel/Wert-Paaren. Ein Schlüssel ist immer eine Zeichenkette und ein Wert kann vom Typ Zeichenkette, Zahl, boolscher Wert oder Null sein. Näheres zur Syntax findet sich im Standard RFC 4627.

## AJAX

*AJAX* (Asynchronous JavaScript and XML) bezeichnet eine Kombination von Techniken zur asynchronen Datenübertragung zwischen einem Client und einem Server. Die Anfrage läuft dabei im Hintergrund, ohne das der Anwender diese bemerkt. Die Seite wird dabei nicht neu geladen, es werden lediglich die relevanten Inhalte mit den angeforderten Daten aktualisiert.

Die Grundbausteine von *AJAX* beruhen auf den folgenden Techniken (Gamperl, 2007):

**XHTML und CSS:** zur Formatierung einer Webseite.

**DOM:** (*Document Object Model*) für den Zugriff auf den Dokumentenbaum.

**XML oder JSON:** als Datenaustauschformat.

**XMLHttpRequest:** zur asynchronen Client/Server-Kommunikation.

**JavaScript:** als Schnittstelle all dieser Komponenten.

### Yahoo Interface Library

Die *Yahoo Interface Library* [6] oder kurz *YUI* ist eine frei verfügbare, in *JavaScript* geschriebene Bibliothek zur Entwicklung von asynchronen Webseiten und Web-Anwendungen. *YUI* bietet eine Reihe von gut aufeinander abgestimmten Funktionen, Vorlagen und Komponenten zur Erstellung von Webseiten mit dynamischen Inhalten.

Zu den Komponenten von *YUI* gehören unter anderem vordefinierte Menüs, Schaltflächen, Diagramme und Navigationskomponenten. Außerdem bietet *YUI* noch eine Reihe von Layoutvorlagen in Form von *CSS*-Dokumenten. Der Funktionsumfang von *YUI* unterstützt alle Grundbausteine von *AJAX*, beispielsweise die Client/Server-Kommunikation mit Hilfe von *XMLHttpRequest*-Objekten, den dynamischen Zugriff auf den Dokumentenbaum mittels *DOM*, sowie die Operationen für die Verwendung von *JSON*

[6] http://developer.yahoo.com/yui

Die Entscheidung, *YUI* zu verwenden, wurde hauptsächlich von der Axivion GmbH voran getrieben, da *YUI* die nötigen Komponenten für eine Portalsoftware liefert und der Prototyp als eine solche implementiert werden sollte.

### AMCharts

*AMCharts*[7] ist ein frei verfügbares Set von Flash-Diagrammen, die mit Hilfe einer Skriptsprache wie *JavaScript* konfiguriert und manipuliert werden können. Es enthält vier verschiedene Diagrammtypen. Kreisdiagramme, Linien- und Flächendiagramme, Balken- und Säulendiagramme, sowie Punkt- und Blasendiagramme (siehe Abschnitt 2.1.7). Die darzustellenden Daten können aus XML- oder CSV-Dateien geladen werden oder dynamisch generiert werden.

*AMCharts* wurde zusätzlich in der Implementierung verwendet, da die Diagramme von *YUI* verschiedene Diagrammtypen, wie etwa das Blasendiagramm nicht unterstützen. Außerdem befindet sich die Diagramm-Komponente von *YUI* noch in der Entwicklung, wodurch sich Fehler in der Darstellung ergaben. Ein weiterer Grund *AMCharts* zu integrieren, war die fehlende Möglichkeit, die Diagramme zu beschriften.

### SQLite

Die Programmbibliothek *SQLite*[8] wurde für den eingebetteten Einsatz entworfen und enthält ein relationales Datenbanksystem. Dadurch lassen sich *SQLite*-Datenbanken in Anwendungen integrieren, ohne das ein zusätzlicher Serverdienst laufen muss. Für viele Programmier- und Skriptsprachen existieren passende Datenbankschnittstellen. Die Datenbank wird in mehreren oder bei Bedarf in einer einzigen Datei gespeichert. So entfällt der administrative Aufwand für das Anlegen einer Berechtigungsstruktur für verschiedene Nutzer, da die Zugriffsrechte über die Rechte des Dateisystems verwaltet werden. Die Speicherung in nur einer Datei ermöglicht die einfache Portierung der Datenbank.

Anders als im *SQL*-Standard, wo bei der Tabellenerstellung der Datentyp angegeben werden muss, verwendet *SQLite* keine Datentypen. Alle Daten sind als null-termininierte Strings gespeichert und nicht als binäre Repräsentation. Aus Gründen der Kompatibilität oder wenn die Daten sortiert werden sollen, können bei der Tabellenerstellung die Typen mit angegeben werden. *SQLite* unterscheidet intern nur bei der Sortierung zwischen Integer und String Datentypen.

Für das Portalsystem wurde *SQLite* als Datenbanksystem ausgewählt, da es einfach aufzusetzen ist, kein weiterer Serverdienst notwendig ist und die Anfragen aufgrund der fehlenden Client-Server Kommunikation sehr performant sind. Durch den Einsatz als eingebettete Datenbank kann sie direkt mit der Portalsoftware ausgeliefert werden und muss nicht beim Kunden installiert und administriert werden.

[7] http://www.amcharts.com/
[8] http://www.sqlite.org

## 4.3 Datenbank

Wie in Abschnitt 3.2 erläutert, bildet der *RGF* ein Programm in der Version ab, in der es bei der Analyse vorlag. Bei Beginn der Diplomarbeit existierte keine Möglichkeit, die Daten aus mehreren *RFG* Versionen in Beziehung zueinander zu setzen, außer durch den manuellen Vergleich. Die Speicherung von mehreren Versionen in der bestehenden Struktur des *RFGs* war nicht umsetzbar, sodass die Wahl auf den Einsatz einer Datenbank fiel, wie es auch bei anderen gängigen Qualitätsüberwachungssystemen zu finden ist (Abschnitt 2.4 Seite 30).

Eine Datenbank bietet zum einen den Vorteil, dass große Datenmengen dauerhaft gespeichert werden können und zum anderen, können Nutzer mit Hilfe der *Structured Query Language (SQL)* die Daten um eigene Metriken erweitern oder eigene Analysen durch Abfragen realisieren.

Das verwendete Datenbankmodell entstand während der Diplomarbeit und wurde von mir in Zusammenarbeit mit der Axivion GmbH entwickelt. Als Datenbanksystem kommt *SQLite* zum Einsatz.

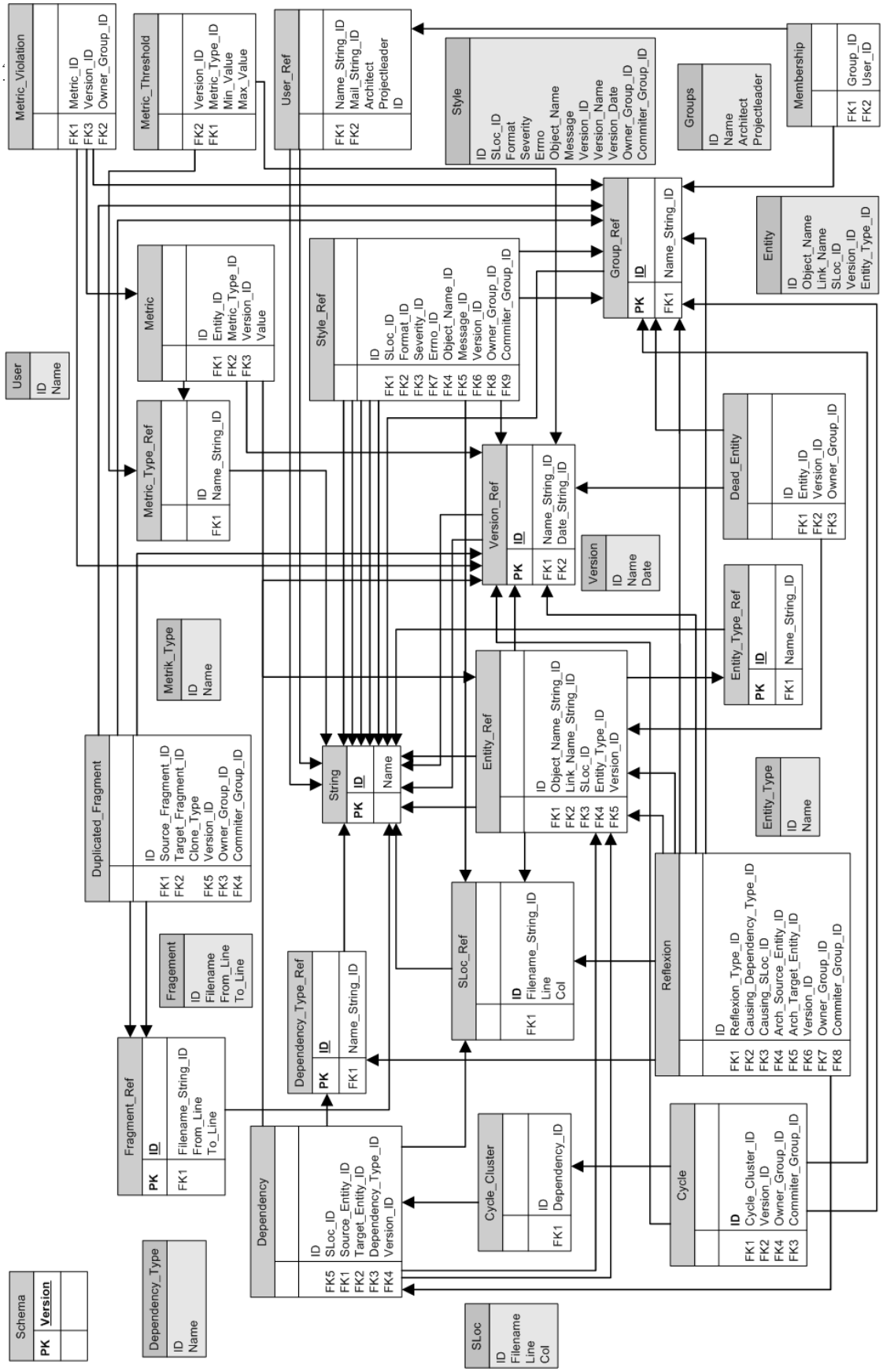


Abbildung 4.1: Datenbankschema

### 4.3.1 Datenbankschema

Eine Anforderung der Axivion GmbH an das Datenbankschema war die Konformität mit dem *SQL* Standard, sodass die Umstellung auf ein anderes Datenbanksystem ohne großen Aufwand möglich ist. Abbildung 4.1 zeigt das Schema der Datenbank. Im Folgenden werden die Tabellen und ihre Bedeutung kurz erläutert.

In der Tabelle `Schema` wird die Versionsnummer des Datenbankschemas gespeichert. Alle Strings, die in den Daten vorkommen, werden zentral in der `String` Tabelle abgelegt. Dadurch, dass alle anderen Datensätze nur noch Referenzen auf diese enthalten, werden Redundanzen vermieden und damit Speicherplatz eingespart. Alle Tabellen, in denen solch eine Referenz vorkommt, haben an ihrem Namen die Endung `_Ref` angehängt. Da bei einer Anfrage an diese Tabellen immer die Verknüpfung zur `String` Tabelle hergestellt werden müsste, sind zur Vereinfachung Sichten definiert, die die Zusammenführung beider Tabellen repräsentieren. In Abbildung 4.1 sind die Sichten als graue Entitäten dargestellt.

### 4.3.2 Versionierung

In der Tabelle `Version_Ref` wird der Name und das Datum einer Version gespeichert. Alle Einträge, die über mehrere Versionen identifizierbar sind, verfügen über eine Referenz auf eine Version. Dadurch erfolgt die Versionierung des *RFGs*, die es ermöglicht, z.B. die Entwicklung von Metrikwerten oder Klonen über einen bestimmten Zeitpunkt zu beobachten.

Die Probleme und Schwächen, die die Analysen aufdecken, wie beispielsweise Metrikverstöße oder Klone, werden bestimmten Nutzern zugeordnet. Diese sind in der Tabelle `User_Ref` gespeichert. Mehrere Nutzer werden zu Gruppen aus `Group_Ref` zusammengefasst. Die Zuordnung von Gruppen zu Nutzern geschieht über die Tabelle `Membership`

Quelltextpositionen (*source locations SLoc*) finden sich in `SLoc_Ref`. Sie besitzen keine Referenz auf eine Version, da sie ihre Positionen ändern und nicht mehr eindeutig sind, ebenso wie die in `Fragment_Ref` gespeicherten Quelltextbereiche (*source ranges SLoc_Range*). Eine *SLoc* wird durch eine Zeile und eine Spalte beschrieben und eine *SLoc_Range* durch eine Start- und eine Endzeile.

Die Knoten eines *RFGs* werden in `Entity_Ref` repräsentiert und die dazugehörigen Typen in *Entity_Type_Ref*. Zu einem Knoten werden *Object_Name*, *Link_Name*, die *SLoc* und sein Typ gespeichert. Die Tabelle `Dependency` speichert die Kanten und `Dependency_Type_Ref` die Kantentypen eines *RFGs*. Zu einer Kante gehört eine *SLoc*, ein Quell- und ein Zielknoten und der Kantentyp.

Funktionen, die nie benutzt werden, (*dead functions*) sind in `Dead_Entity` repräsentiert. Da ein Knoten referenziert wird und eine Version für die toten Funktionen existiert,

kann nachvollzogen werden, seit wann das Element unbenutzt ist. Klone sind in `Duplicated_Fragment` zu finden. Zu einem Klon gehört der Bereich im Quelltext des Originals und der Quelltextbereich der Kopie, der Klontyp sowie eine Version. Die Spalten `Commiter_Group_ID` und `Owner_Group_ID` legen fest, wer für den Klon zuständig ist und wer für ihn verantwortlich war.

Zyklische Abhängigkeiten bilden die beiden Tabellen `Cycle` und `Cycle_Cluster` ab. Letztere fasst Kanten unter einer *ID* zusammen und definiert damit den Zyklus. In `Cycle` werden dann dem Zyklus eine Version, Verursacher und Zuständige zugeordnet. Jede Änderung in einem Zyklus wird als neuer Zyklus erfasst.

`Style_Ref` beinhaltet Style-Verstöße, die von Compilern ausgegeben werden (z.B. *GCC*, *GNAT*). Ein gepeicherter Verstoß beinhaltet eine Position (*SLoc*), ein Format, den Schweregrad des Verstoßes (z.B. *warning*), eine Fehlernummer, das Fehlerobjekt, die zugehörige Nachricht und die beteiligten Personen.

Eine Metrik hat einen bestimmten Typ der in `Metric_Type_Ref` abgelegt ist. Die Tabelle `Metric` speichert zu einer Metrik den Knoten, an den sie annotiert war, ihren Typ, eine Version und schließlich den eigentlichen Wert. Da die Knoten über mehrere Versionen hinweg identifizierbar sind, kann der zeitliche Verlauf von Metriken analysiert werden. Eine Metrik für einen Knoten wird für jede Version mit ihrem Wert gespeichert. Verletzt eine Metrik einen Schwellwert, erfasst dies die Tabelle `Metric_Violation` durch die Metrik, eine Version und die zuständigen Personen. Ein Verursacher wird nicht gespeichert, da es in der Praxis schwer feststellbar ist, wer für die Über- oder Unterschreitung eines Schwellwertes gesorgt hat. Der Schwellwert selber befindet sich in `Metric_Threshold`. Er ist für einen bestimmten Metriktyp und durch einen Maximal- und einen Minimalwert definiert. Da sich Schwellwerte im Laufe eines Projektes ändern können, wird zusätzlich noch eine Version gespeichert.

In der Tabelle `Reflexion` wird das Ergebnis der Reflexions-Analyse gespeichert. Der Typ einer Architekturverletzung entspricht einem Kantentyp des *RFGs* (*abscence*, *divergence*, *convergence*). Außerdem besitzt eine Verletzung noch einen Quell- und einen Zielknoten, die den verursachenden Architekturkomponenten entsprechen, eine Version und die Verursacher sowie die Zuständigen.

### 4.3.3 Hinzufügen weiterer Messdaten zur Datenbank

In derselben Weise wie beliebige Attribute an die Knoten und Kanten des *RFGs* annotiert werden können, erlaubt es das Datenbankschema, weitere Daten zu integrieren. So können beispielsweise Ergebnisse von Performanz-Messungen oder Bug-Trackern als weitere Metrik zur Datenbank hinzugefügt werden. Von mir wurden in der Arbeit prototypisch *SVN* Informationen als neue Metrik eingebunden. Da in C Module Dateien repräsentieren, können die Commits pro Datei aufsummiert werden und an die Modul-Entitäten in der Datenbank angehängt werden. Identifiziert werden die Dateien dabei über den `Object_Name`.

## 4.4 Aufbau und Architektur von Bareto

Der Aufbau der entwickelten Portalsoftware *Bareto* (Bauhaus Reporting Tool) ist in Anlehnung an die in Abschnitt 2.3 beschriebene Architektur von Großmann und Koschek (2005) entstanden.

### Datenquellen

Als Datenquelle dient sowohl die in Abschnitt 4.3.1 beschriebene Datenbank mit den *RFG* Informationen sowie exemplarisch das Versionsverwaltungssystem *Subversion*, kurz *SVN*. Die *SVN* Informationen können sowohl durch ein Python-Skript in die Datenbank integriert werden als auch in Echtzeit zur Laufzeit des Portals abgefragt werden. Das setzt allerdings voraus, dass der *SVN*-Server erreichbar ist, also eine Anbindung an ein Netzwerk (Unternehmensnetzwerk) oder das Internet besteht.

Die Datenbanken von verschiedenen Projekten werden in einem eigenen Verzeichnis von *Bareto* abgelegt. Dieses wird beim Start automatisch nach Datenbanken durchsucht und gefundene Datenbanken werden von der Software in die Liste der verfügbaren Projekte aufgenommen.

### Datenextraktion

Die Funktionalität für den Zugriff auf die Datenquellen ist in Bareto in separaten Modulen implementiert. Diese Module sind für Extraktion der Daten aus den Informationsquellen zuständig. Die Module gehören nicht zur Portalsoftware, sondern stellen eine Art Bibliothek für die Datenanfragen dar. Diese Umsetzung hat den Vorteil, dass nicht nur das Portalsystem auf die Funktionalität zugreifen kann, sondern z.B. auch die prototypisch umgesetzte Komponente zum Generieren eines Reports. Die Funktionalität muss somit nicht redundant implementiert werden und bei einer Änderung am Datenbankschema muss diese nur an einer zentralen Stelle nachvollzogen werden.

Die Anfragen an die *RFG*-Datenbank sind Anfragen in *SQL*, die durch die Python-Schnittstelle zu *SQLite* realisiert sind. Die Informationen aus dem *SVN* werden mit Hilfe einer weiteren Python-Bibliothek extrahiert, die die gängigen *SVN*-Funktionen implementiert.

### Integrationskomponente

Die Integrationskomponente stellt die Schnittstelle zwischen der Portalanwendung und den oben beschriebenen Modulen zur Datenextraktion dar. Sie bildet die zentrale Zugriffsschicht für die angeschlossenen Datenquellen und kapselt den Zugriff auf diese gemäß den Anforderungen in Abschnitt 4.1.

Eine weitere Aufgabe der Integrationskomponente ist die Umwandlung der Ergebnisse der Abfrage in ein einheitliches Format, welches die Komponenten der Präsentationsebene für die Darstellung benötigen.

In Bareto werden die Daten, die von den Portlets auf der grafischen Oberfläche dargestellt werden, in zwei verschiedene Formate umgewandelt. Das ergibt sich aus der Tatsache, dass zwei Technologien, *YUI* und *AMCharts*, für die Erstellung der Diagramme verwendet werden. Als Datenbasis für *YUI*-Diagramme müssen *JavaScript*-Objekte vorliegen. Hierfür werden zuerst vom Server Python-Objekte erzeugt, die in das textbasierte Datenaustauschformat *JSON* umgewandelt werden und nach der Übertragung auf der Client-Seite in *JavaScript*-Objekte konvertiert werden.

Die Generierung von *XML*, das die Datengrundlage für *AMCharts* bildet, ist einfacher, da die Ergebnisse nicht erst zu Python-Objekten zusammengesetzt werden müssen, sondern sequenziell die *XML*-Struktur erstellt werden kann. Diese wird auch von *AMCharts* direkt verarbeitet und bedarf keiner weiteren Umformung.

#### Benutzerverwaltung

Die Komponente der Benutzerverwaltung erfolgt bei Bareto über eine separate *SQLite* Datenbank, die in einer eigenständigen Datei gespeichert ist. Sie wird im Folgenden als Konfigurationsdatenbank bezeichnet. Neben Benutzernamen und deren Zugangsdaten enthält sie Informationen zum globalen und zum benutzerspezifischen Layout der grafischen Oberfläche.

Zu einem Benutzer werden der Name und das Passwort gespeichert. Das Passwort ist, wie in Abschnitt 4.1 beschrieben, in verschlüsselter Form in der Datenbank abgelegt. Das nutzerspezifische Layout, das durch die Angaben des Anwenders erstellt wurde, wird pro Projekt gespeichert, sodass jedes Projekt eine unterschiedliche Darstellungen haben kann. Verschiedene Rollen werden durch Gruppen abgebildet. Ein Nutzer kann Mitglied von mehreren Gruppen sein und somit mehrere Rollen einnehmen.

Den eigentlichen Zugriffschutz übernimmt ein Python-Modul. Es überprüft die Zugangsdaten bei der Anmeldung und stellt sicher, dass nur registrierte Anwender Zugang zum Portal haben. Das rollenbezogene Einschränken der Informationen und Funktionen der grafischen Oberfläche ist zwar in der Datenbank umgesetzt und in dem Berechtigungsmodul implementiert, wird jedoch in dem Prototyp noch nicht verwendet.

### Präsentation

Die Präsentationskomponente von Bareto ist für die Umsetzung der grafischen Oberfläche zuständig. Sie ist die komplexeste Komponente der Software, da verschiedene Techniken

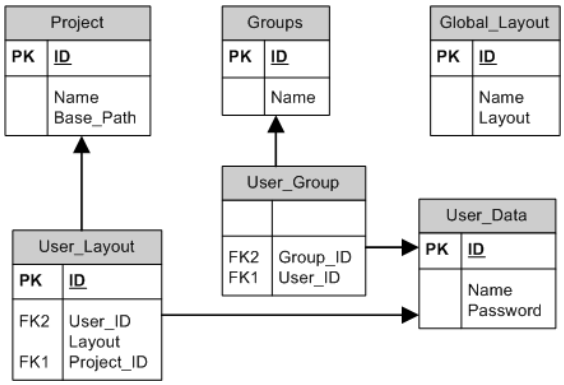


Abbildung 4.2: Konfigurationsdatenbank

und Bibliotheken beim Zusammensetzen der Benutzungsschnittstelle zum Einsatz kommen. Im Folgenden soll deshalb der Aufbau der Präsentationskomponente zur besseren Verständlichkeit nur grob beschrieben werden.

Abbildung 4.3 zeigt den schematischen Aufbau der grafischen Oberfläche von Bareto. In der Kopfzeile wird links das Logo der Axivion GmbH dargestellt. Auf der rechten Seite befinden sich Schaltflächen, über die die Hauptfunktionen, wie das Abmelden zugänglich sind. Zusätzlich gibt es eine Statusanzeige, in der beispielsweise angezeigt wird, mit welchem Namen man gerade angemeldet ist. Die Navigationskomponente beinhaltet ein Menü, über das die verschiedenen Bereiche von Bareto zugänglich sind. Diese werden im Hauptfenster dargestellt.

Diese Struktur bleibt auf allen Seiten erhalten. Sie ist in einer Vorlage, einem sogenannten *Template*, definiert. Das *Template* besteht aus einer HTML Datei und enthält sowohl Anweisungen für die *Template Engine Genshi*, als auch *JavaScript* Funktionen. Damit das *Template* angewendet wird, müssen alle Inhaltsseiten über eine Referenz darauf verfügen. Beim Laden einer Seite fügt *Genshi* den Inhalt der Seite an der dafür vorgesehenen Stelle im *Template* ein. *Genshi* ist dabei in der Lage, einzelne Bereiche der Inhaltsseite für die Ersetzung zu selektieren, sodass die Seite nicht als ganzes an nur einer Stelle im *Template* platziert werden muss.

Die im *Template* enthaltenen *JavaScript* Funktionen sind zuständig für die Initialisierung des Layouts und der darin enthaltenen Komponenten wie das Menü oder die Schaltflächen in der Kopfzeile. Mit Layout ist die Aufteilung in Kopfzeile, Navigation und Hauptfenster gemeint. Allgemeine Formatierungen wie Schriftart,-größe und Hintergrundfarben sind mit Hilfe eines *Stylesheets* (*CSS*) definiert.

Im Hauptfenster erfolgt die Darstellung der Informationen entweder auf vordefinierten, oder auf personalisierten Seiten. In beiden Fällen kommen Portlets zum Einsatz. Sowohl das Layout der vordefinierten Seiten als auch das der personalisierten Seiten ist in der weiter oben bereits erwähnten Konfigurationsdatenbank gespeichert. In der Datenbank

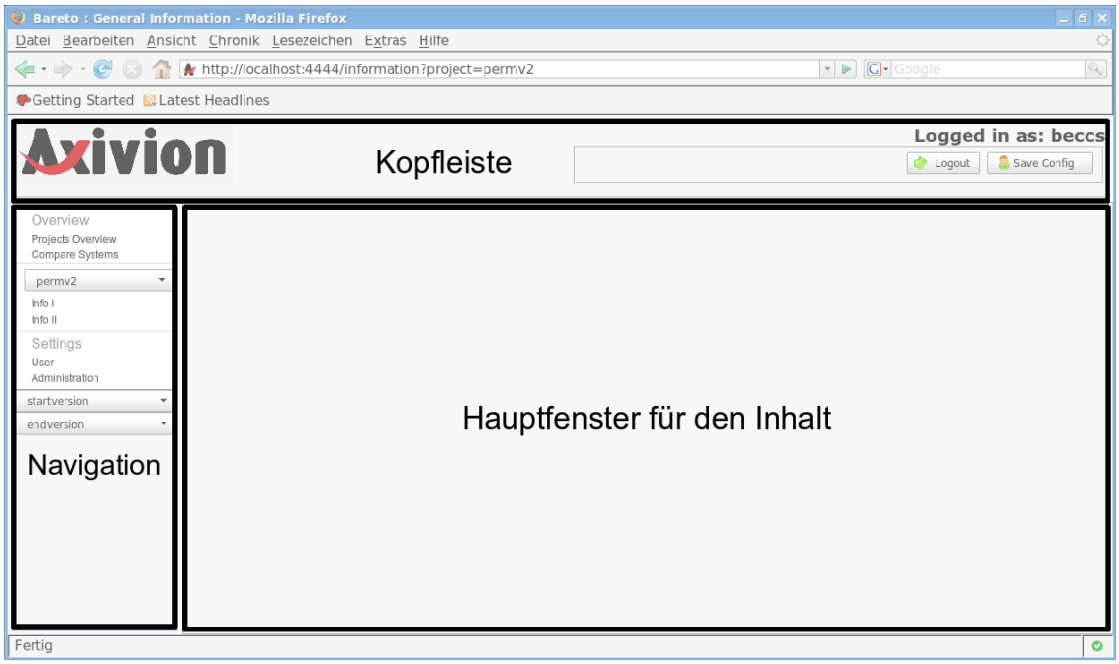


Abbildung 4.3: Aufbau der grafischen Oberfläche

existiert für vordefinierte Seiten nur ein einziges Layout und für personalisierte Seiten ein Layout pro Benutzer. Die vordefinierten Seiten werden also für alle Nutzer identisch dargestellt und besitzen somit ein globales Layout.

**Vordefinierte Seiten:** Bei den vordefinierten Seiten handelt es sich um Seiten mit festgelegtem Aussehen und Inhalt. Diese können nicht vom Benutzer durch Interaktion verändert werden. Die Portlets haben eine feste Position innerhalb der Seite und sind nicht in Größe oder Anordnung veränderbar.

**Personalisierte Seiten:** Die Darstellung und der Inhalt der personalisierten Seiten kann an die persönlichen Präferenzen des Anwenders angepasst werden. Die Anordnung der Portlets kann interaktiv zur Laufzeit durch direkte Manipulation verändert und gespeichert werden

### Portlets

Der Implementierung der Portlets liegt ein modularer Entwurf zugrunde. Für jede Darstellungsart (Tabelle, Liniendiagramm, Baum) existiert ein eigenes Python-Modul mit einer dazugehörigen HTML Datei. Beim Start von Bareto wird das Verzeichnis mit den Modulen durchlaufen und gefundene Module werden automatisch importiert. Dadurch kann die Software sehr einfach um neue Portlets erweitert werden.

Die HTML Dateien beschreiben den Rahmen der Portlets und enthalten eine *JavaScript* Funktion die für die Abfrage des dynamischen Inhalts vom Server zuständig ist. Die Portlets legen nur fest, welche Visualisierungsform für die Informationen zum Einsatz kommt und wohin sie ihre Anfrage richten müssen, aber nicht, welche Daten konkret vom Server geholt werden sollen. Das wiederum ist in der Konfigurationsdatenbank festgelegt. Sie enthält für jedes Portlet die Einzelheiten zur Datenabfrage, wie z.B. die Anzahl der Zeilen für ein bestimmtes Projekt. Dadurch können für ein Portlet beliebige Anfragen erstellt werden, sofern die Visualisierungsform für die Ergebnisse geeignet ist.

Abbildung 4.4 zeigt die Darstellung eines Portlets. Ein Portlet besitzt eine Titelleiste und einen Bereich in dem es Informationen visualisieren kann. Die Portlets sind Komponenten der *YUI* Bibliothek, mussten jedoch für meine Verwendungszwecke von mir erweitert und angepasst werden.


Abbildung 4.4: Aufbau eines Portlets

Für den personalisierbaren Bereich wurden die Portlets um mehrere Funktionen erweitert. Zum einen können sie durch *Drag and Drop* („Ziehen und Fallenlassen) auf der Seite positioniert werden, zum anderen ist es möglich sie zu minimieren. Dadurch kann der Benutzer die Informationen strukturieren, indem er die für ihn wichtigen Informationen weiter oben anordnet oder unwichtige Informationen durch Minimieren ausblendet.

Die benutzerdefinierten Portlets unterscheiden sich von den normalen Portlets durch ein kleines Pfeilsymbol am linken Rand der Titelleiste, wie in Abbildung 4.5 zu sehen ist.

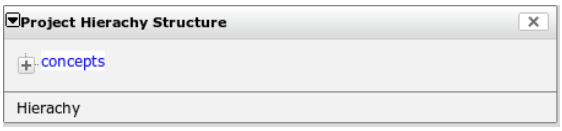


Abbildung 4.5: Benutzerdefiniertes Portlet

Dieser Pfeil dient als Schaltfläche zum Minimieren und zeigt außerdem noch den Zustand des Portlets an. Bei einem minimierten Portlet ändert er die Richtung (siehe Abbildung 4.6). Durch erneutes Anklicken kann das Portlet wieder geöffnet werden.

Wird das Portlet im minimierten Zustand an eine andere Position verschoben, bleibt es geschlossen.


Abbildung 4.6: Minimiertes Portlet

## 4.5 Ablauf einer Anfrage

Nachdem alle Komponenten von Bareto erläutert worden sind, wird im folgenden Abschnitt beschrieben, welche Schritte ausgeführt werden, wenn der Client beim Server eine Seite anfragt.

Der gesamte Ablauf der Kommunikation ist in Abbildung 4.7 zu sehen. Der Client fragt eine bestimmt Seite an. Diese Anfrage wird vom Server bearbeitet, indem er nach dem Layout für diese Seite in der Konfigurationsdatenbank sucht. Handelt es sich um eine personalisierte Seite, entnimmt der Server aus der Konfigurationsdatenbank das für den angemeldeten Benutzer hinterlegte Layout. Handelt es sich um eine vordefinierte Seite, entnimmt er das zugehörige globale Layout.

Als nächstes fügt *Genshi* das *Template* und den Inhalt zusammen und nimmt die nötigen Ersetzungen vor. Anschließend wird die Seite zusammen mit der Konfiguration zum Client übermittelt. Die Konfiguration wird einer *JavaScript* Funktion übergeben, die anhand dieser die benötigten Portlets anfragt. Jedes Portlet wiederum stellt eine Anfrage an den Server, welcher den dynamischen Inhalt der Portlets generiert.

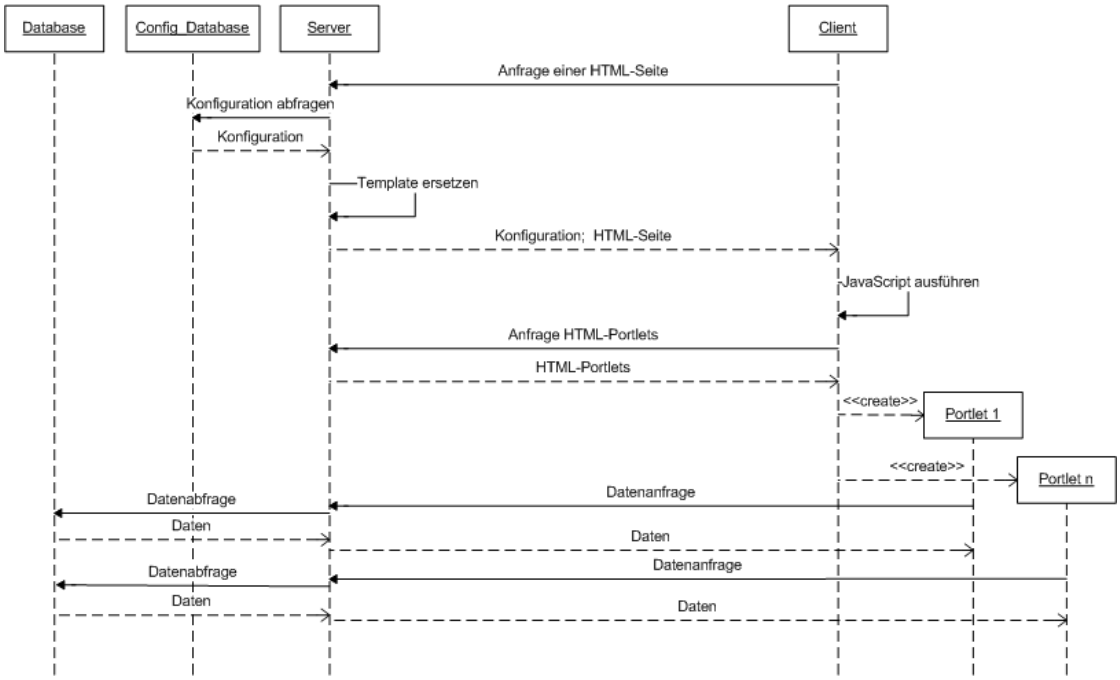


Abbildung 4.7: Ablauf einer Kommunikation

# 5 Auswertung

Im Folgenden soll die in Kapitel 4 beschriebene Software evaluiert werden. Da im Rahmen der Diplomarbeit eine Nutzerbefragung nicht umsetzbar war, soll in Abschnitt 5.2 anhand der Beschreibung eines konkreten Anwendungsfalles überprüft werden ob das entwickelte Werkzeug die beschriebenen Anforderungen erfüllt. Außerdem sollen Performanzmessung der Datenbankanfragen in Abschnitt 5.3 die Skalierbarkeit der Software sicherstellen. Bevor allerdings auf die Auswertung eingegangen wird, sollen zuerst noch die Besonderheiten erläutert werden, die beim Testen von Webanwendungen zu beachten sind.

### Besonderheiten bei der Untersuchung von Webanwendungen

Da sich Webanwendungen von konventioneller Software unterscheiden, ergeben sich auch beim Testen einige Besonderheiten und Schwierigkeiten. Eine Webanwendung besteht oft aus verschiedenen Komponenten, die von verschiedenen Herstellern stammen. Dazu gehören beispielsweise Datenbanken, Applikationsserver und Programmbibliotheken. Sowohl die falsche Konfiguration der Komponenten als auch die nicht vorhandene Kompatibilität zwischen ihnen kann Fehler verursachen. Oft weisen auch einzelne Komponenten Mängel auf, was die Fehlersuche erheblich erschwert.

Die Vielzahl potentieller Endgeräte und die Bandbreite der Browser erschweren das Testen erheblich. Die Hypertext-Struktur, die den Webanwendungen zugrunde liegt, ist oft sehr verzweigt und komplex. Eine vollständige Testabdeckung der Struktur ist sehr aufwendig und kaum umsetzbar.

## 5.1 Statistik

Sowohl die Auswertung der Anwendungsfälle als auch die Performanzmessungen der Datenbank wurden auf einem PC mit der in Tabelle 5.1 Soft- und Hardwarekonfiguration durchgeführt.

Als Testgrundlage wurden die Datenbanken von den zwei Systemen *SDCC* und *concepts* verwendet. Die *RFGs* von verschiedenen Versionen, sowie weitere Daten zur Klon-,

| Eingesetzte Hardware/Software | |
|---|---|
| **Rechner** | PC mit Intel Core Duo, 2.0 GHz, 1 GB Arbeitsspeicher |
| **Betriebssystem** | Gentoo Linux |
| **Datenbank** | SQLite 3.5.6 |
| **Webbrowser** | Mozilla Firefox 2.0.0.11 |

Tabelle 5.1: Testumgebung

Architektur- und Style-Analyse wurden von der Axivion GmbH in die Datenbankstruktur überführt und mir zur Verfügung gestellt.

Die Tabelle 5.2 zeigt die Referenzwerte der beiden Systeme. *Concepts* umfasst ca. 5.000 Zeilen Programmcode und bei *SDCC* sind es 64.000 Zeilen. In der Datenbank von *concepts* sind 448.256 Datensätze zu zwanzig Versionen gespeichert. Die Datenbank für *SDCC* enthält 13.422.676 Datensätze und insgesamt 83 überführte *RFG* Versionen. Die Größe der Dateien auf dem Datenträger entspricht 26 MB für *concepts* und 862 MB für *SDCC*.

| | SDCC | concepts |
|---|---|---|
| **Größe des Systems (*LOC*)** | $\tilde{6}4$ 000 | $\tilde{5}$ 000 |
| **Anzahl der Versionen** | 83 | 20 |
| **Anzahl der Datensätze** | 13 422 676 | 448 256 |
| **Größe der DB-Datei** | 862 MB | 26 MB |

Tabelle 5.2: Untersuchte Systeme

## 5.2 Anwendungsszenario

Zur Überprüfung der Anforderungen soll die beispielhafte Nutzung von Bareto anhand der Anwendungsfälle beschrieben werden. Hierfür werden die beiden Anwendungsfälle aus Abschnitt 4.1.1 zu einem kombiniert. Bei dem Test wurden Datenbanken von den zwei Projekten *concepts* und *sdcc* eingesetzt (siehe Abschnitt 5.1). Folgendes Szenario wird dargestellt:

Ein Projektleiter möchte sich über den aktuellen Stand seiner Projekte informieren und startet die Portalsoftware. Er analysiert die Kennzahlen und stellt fest, dass es bei einem Projekt neue Klone gibt. Aufgrund dieser Information beauftragt er die Entwickler, die an diesem Projekt beteiligt sind, sich darum zu kümmern. Einer der Entwickler nimmt sich dieser Aufgabe an und analysiert die Detailinformationen, um die Zuständigen für die Klone herauszufinden.

Die Abbildung 5.1 zeigt den Anmeldebildschirm von Bareto. Diese Seite wird nach dem Start angezeigt, da ein Zugang zum Inhalt nur für registrierte Anwender möglich ist. Nach

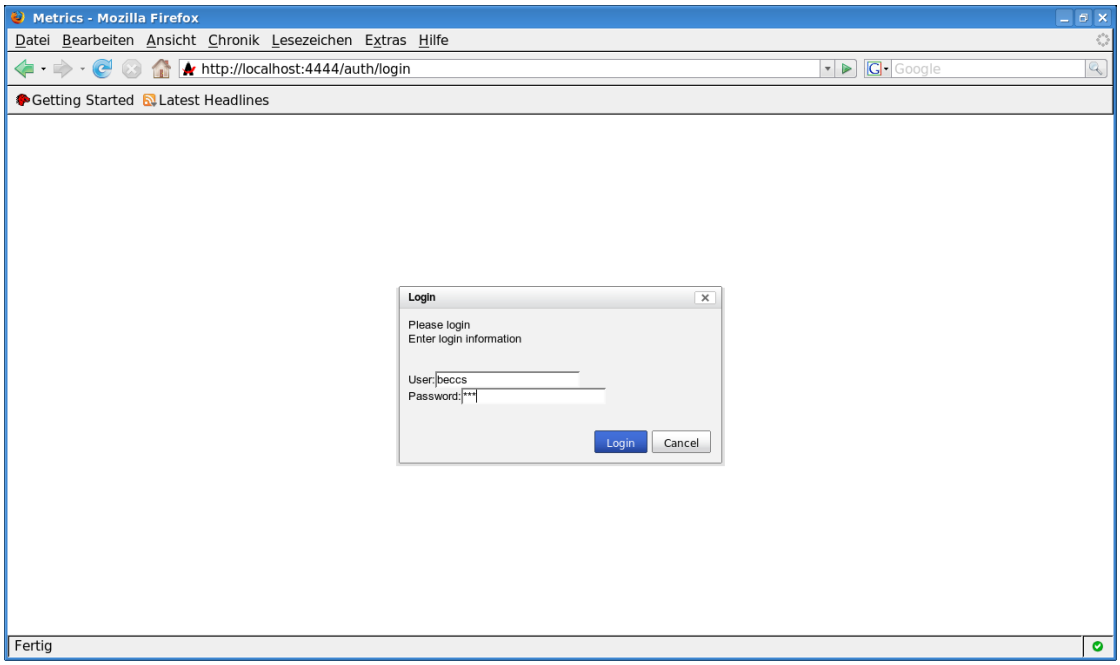


Abbildung 5.1: Anmeldebildschirm

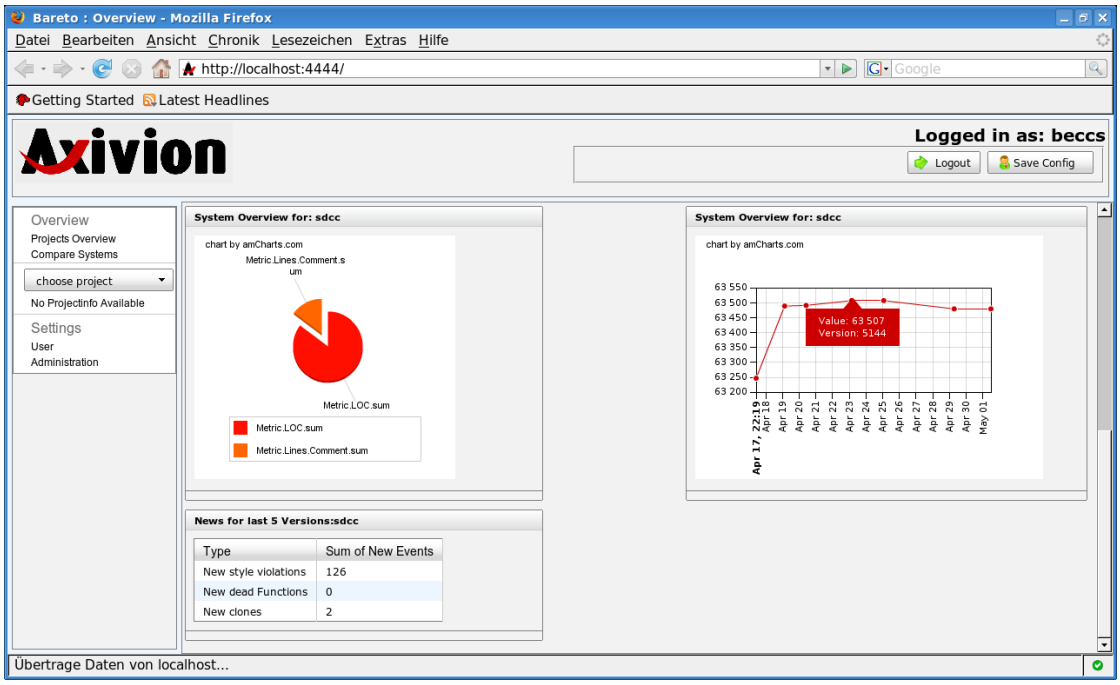


Abbildung 5.2: Übersicht über alle Projekt

der Anmeldung öffnet sich die Übersichtsseite. Auf dieser befinden sich Informationen über alle Projekte, für die eine Datenbank vorhanden ist. In Abbildung 5.2 sieht man drei Portlets, die jeweils Daten für das Projekt *sdcc* enthalten. Die Portlets für das zweite Projekt *concepts* sind hier nicht sichtbar. Sie befinden sich weiter oben auf der Seite und sind über den rechten Scrollbalken zugänglich.

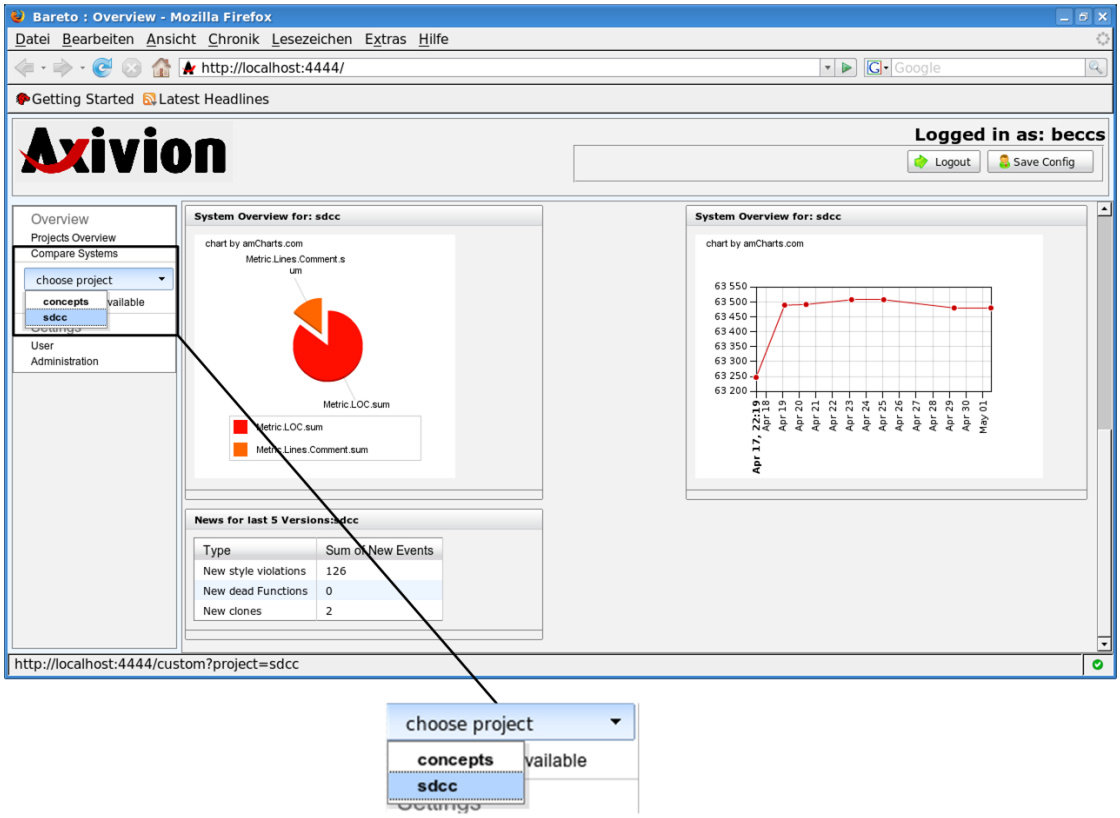


Abbildung 5.3: Auswählen eines Projektes

Das Portlet links oben stellt die Gesamtanzahl der Zeilen und die Gesamtanzahl der kommentierten Zeilen für das Projekt *sdcc* in einem Tortendiagramm dar. Rechts daneben befindet sich ein Liniendiagramm, dass den Verlauf der Größe des Systems anzeigt (auch über die Anzahl der Zeilen). Für die beiden Diagramme kann der Nutzer sich interaktiv weitere Informationen anzeigen lassen, indem er den Mauszeiger über die Elemente des Diagramms führt. In Abbildung 5.2 ist die Information für einen Punkt im Liniendiagramm vergrößert dargestellt. Das Portlet unter dem Tortendiagramm listet in einer Tabelle die Anzahl der Probleme auf, die für das Projekt hinzugekommen sind. In diesem Fall sind das Style-Verstöße und Klone. Mit dieser Information beauftragt der Projektmanager nun einen der Entwickler von *sdcc*, den neu hinzugekommenen Klonen auf den Grund zu gehen.

Nachdem sich der Entwickler erfolgreich am Portal angemeldet hat, möchte er Details für *sdcc* herausfinden. Dafür wählt er über das Navigationsmenü am linken Rand das Projekt aus, wie es die Abbildung 5.3 zeigt.

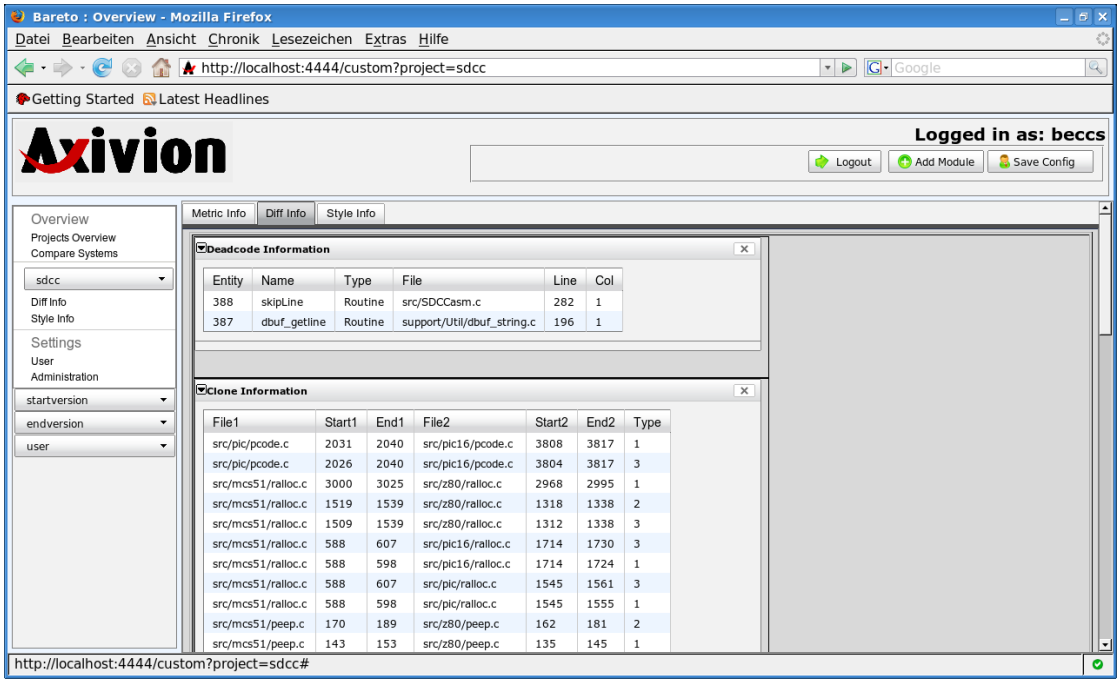


Abbildung 5.4: Detailinformationen zu toten Funktionen und Klonen

Als nächstes gelangt der Entwickler zu den Detailinformationen für das ausgewählte Projekt. In Abbildung 5.4 ist eine personalisierte Seite (siehe Abschnitt 4.4) mit Informationen über tote Funktionen und Klone dargestellt. Die personalisierten Seiten erkennt man an der Seitenaufteilung durch die Reiter (*Metric Info*, *Diff Info* und *Style Info*) und durch das Pfeilsymbol in der Titelleiste der Portlets. Außerdem ist die das Navigationsmenü auf der linken Seite durch drei weitere Auswahlschaltflächen erweitert.

Da der Entwickler nur an den Kloninformationen interessiert ist, möchte er die obere Tabelle ausblenden. Hierfür klickt er mit der Maus auf das Pfeilsymbol, wodurch sich das Portlet minimiert, wie in der Abbildung 5.5 zu sehen ist. Die Tabelle, die die Klone auflistet, enthält allerdings die Klone aller *RFG*-Versionen, die in der Datenbank gespeichert sind und sind somit zu umfangreich. Deshalb möchte der Entwickler die Versionen einschränken.

Das geschieht über die drei Auswahlmenüs *startversion*, *endversion* und *user*. Abbildung 5.6 zeigt die Auswahl einer Startversion. Das Filtern für einen bestimmten Nutzer findet auf die gleiche Weise statt. Nach dem Filtern findet sofort eine Aktualisierung der Tabellen statt. In diesem Fall möchte der Entwickler die Klone zu den neuesten sechs

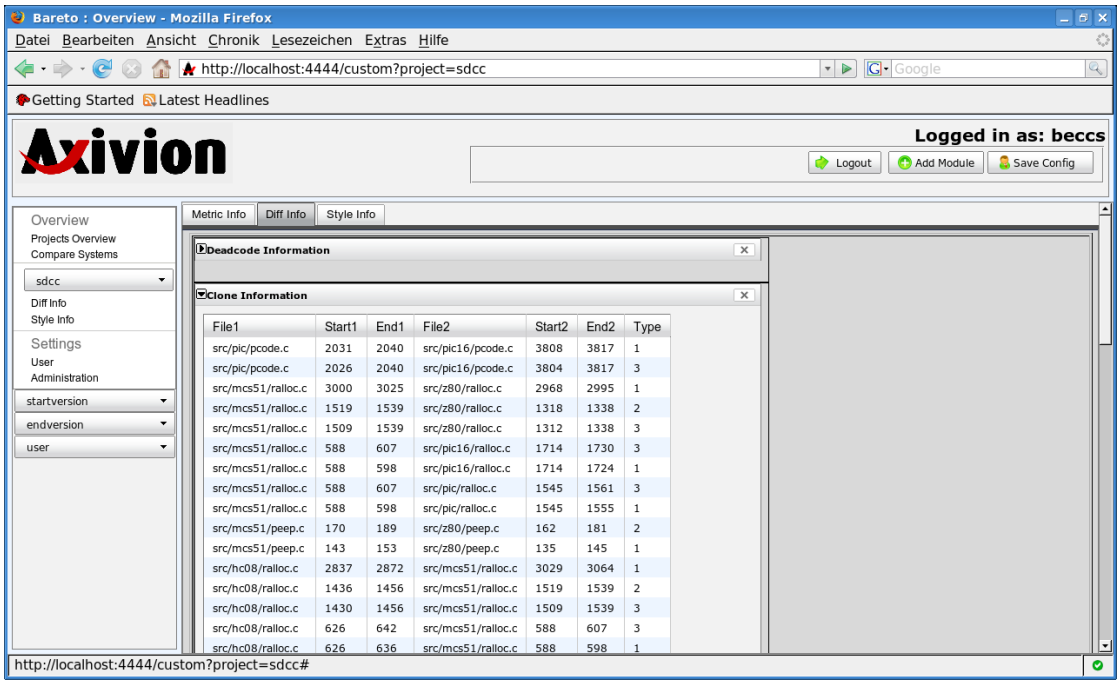


Abbildung 5.5: Minimiertes Portlet für Kloninformationen

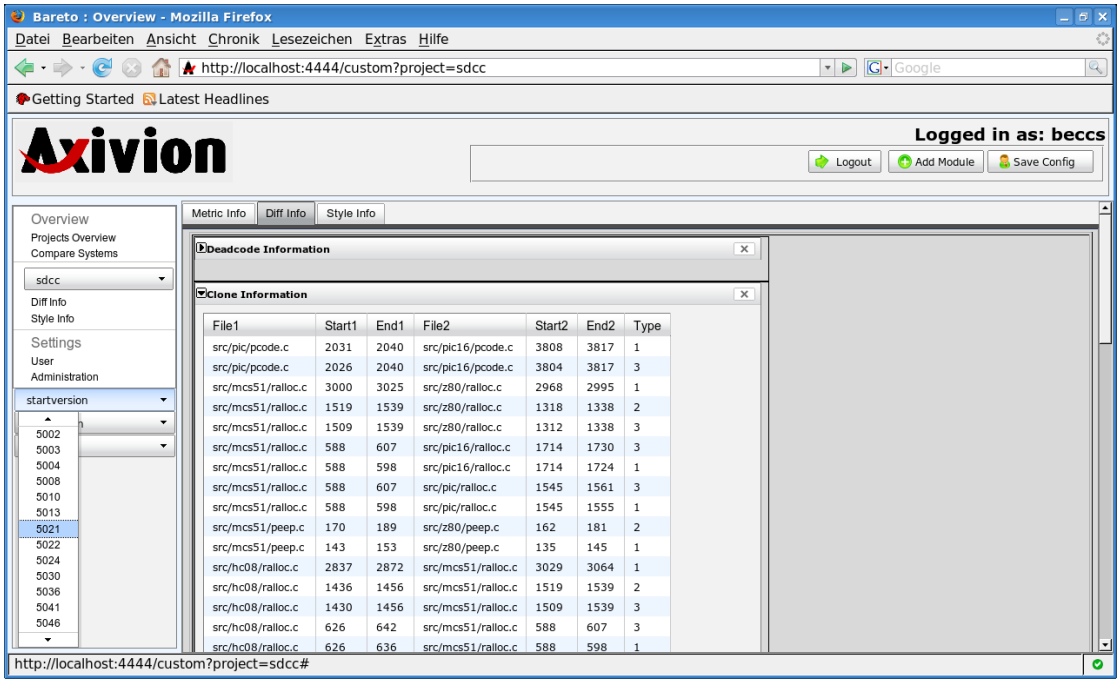


Abbildung 5.6: Einschränken der Versionen

Versionen, für die er zuständig ist, anzeigen lassen. Abbildung 5.7 zeigt, dass mit diesen Suchkriterien für ihn keine Einträge in der Tabelle vorhanden sind.

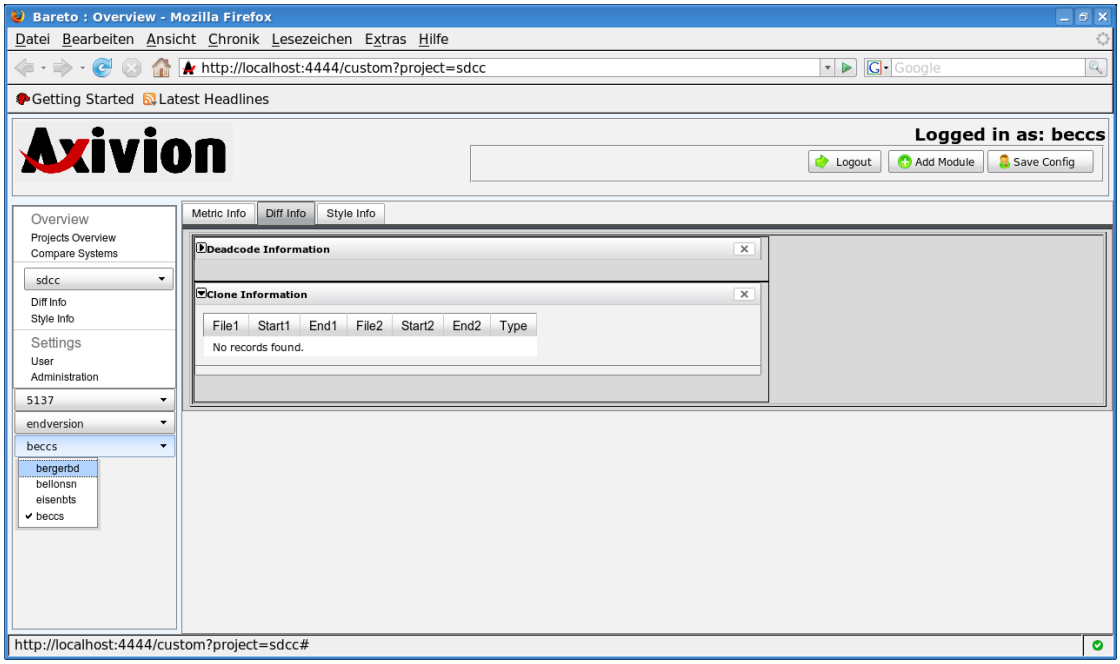


Abbildung 5.7: Nach einem Benutzer filtern

Im nächsten Schritt sucht der Entwickler nach dem Zuständigen und wählt einen anderen Nutzer aus. Für diesen werden die beiden Klone, die der Projektleiter in seiner Übersicht gemeldet bekommen hat, angezeigt (siehe Abbildung 5.8). Der Entwickler kann nun diese Tatsache dem zuständigen Kollegen melden, damit dieser die Fehler beheben kann.

Nach Abschluss des Anwendungsfalles sollen im Folgenden noch weitere Funktionen von Bareto beschrieben werden, um die Benutzung zu verdeutlichen. Neben den bereits vorgestellten Portlets wurden von mir in Bareto außerdem zweigeteilte Portlets umgesetzt. Die ermöglichen die Darstellung von zusammenhängenden Informationen. In Abbildung 5.9 ist ein solches geteiltes Portlet dargestellt. Auf der linken Seite befindet sich ein Baum, der die hierarchische Struktur der Knoten im *RFG* wiedergibt. Auf der rechten Seite befindet sich eine Tabelle, die die Metrikinformationen zu einem ausgewählten Knoten anzeigt. Die Knoten in dem Baum lassen sich durch Klicken auf die Symbole öffnen und durch einen Klick auf ihren Namen auswählen. Die Tabelle wird für jeden neu ausgewählten Knoten aktualisiert.

Eine weitere Möglichkeit, um Metrikwerte anzuzeigen, bieten die weiteren Diagrammarten in Bareto. Abbildung 5.10 zeigt ein Blasendiagramm und eine Tabelle, die Werte für Module eines Systems realisieren.

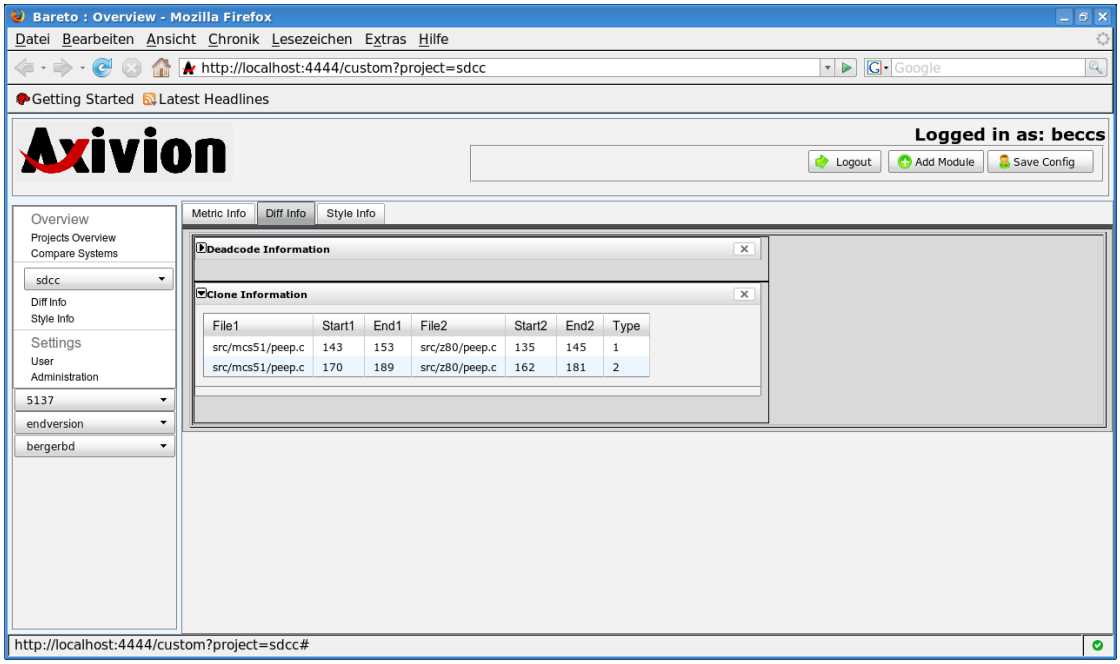


Abbildung 5.8: Anzeige der Klone für einen Versionsbereich und einen Nutzer

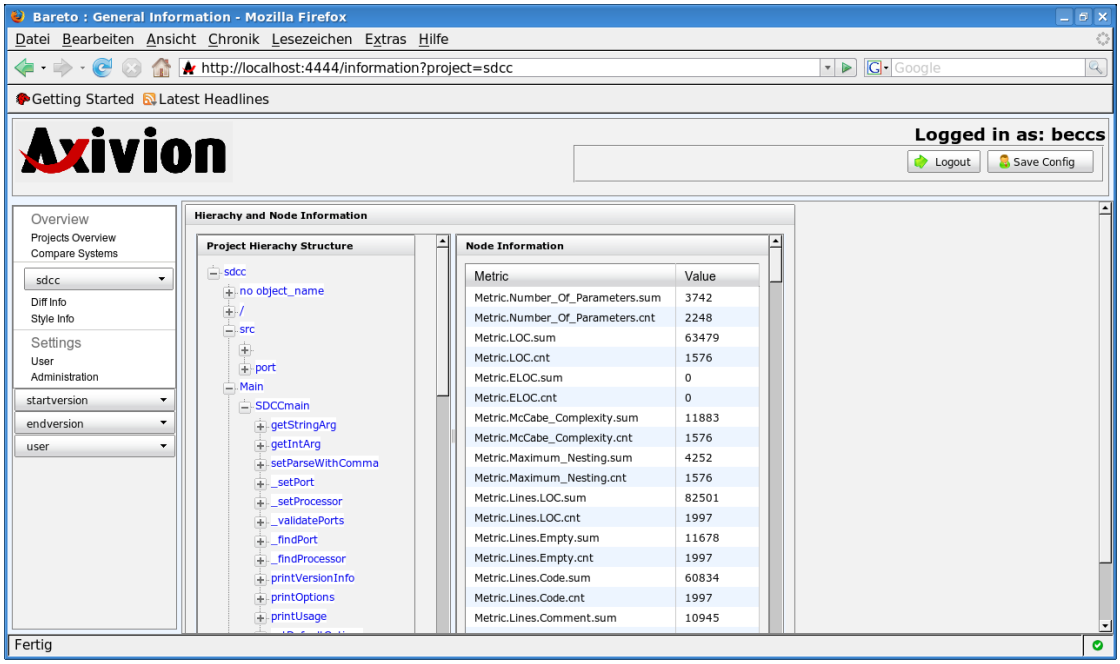


Abbildung 5.9: Ein zweigeteiltes Portlet

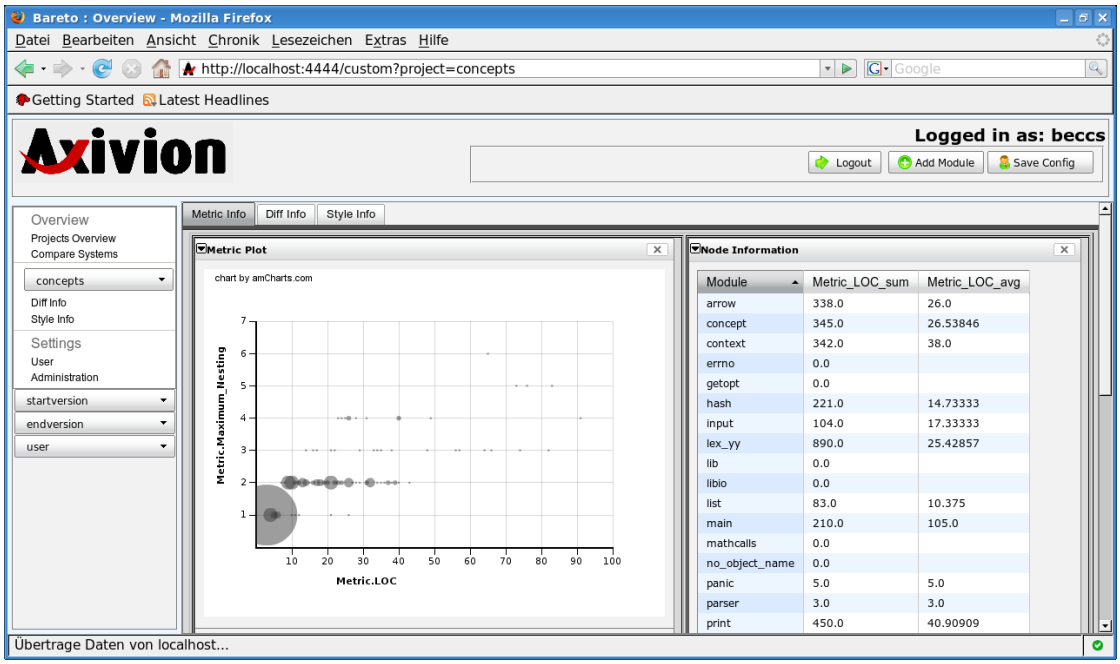


Abbildung 5.10: Detailinformationen für Metriken

Der Anwender hat bei personalisierten Seiten die Möglichkeit, die Portlets frei auf der Arbeitsfläche anzuordnen. Das Portlet, das er verschieben möchte, kann er mit Hilfe der Maus auswählen und an eine neue Position ziehen. Die Position auf der Arbeitsfläche muss dabei nicht frei sein. Ist an dieser Stelle bereits ein anderes Portlet, wird das vorhandene Portlet an eine neue Position verschoben.

Abbildung 5.11 stellt das Verschieben des linken Portlets aus Abbildung 5.10 dar. Das Portlet wird beim Umsortieren durch ein kleines graues Rechteck abgebildet, da ein zu großes Portlet andernfalls den gesamten Blick auf die Arbeitsfläche verdecken würde. In diesem Fall wird das Portlet von der linken auf die rechte Seite geschoben. Dadurch tauschen die beiden Portlets ihre Position. Das Ergebnis der interaktiven Umsortierung zeigt Abbildung 5.12

Abschließend soll noch kurz auf die Funktion des Hinzufügens von Portlets eingegangen werden. Die Oberfläche hierfür ist in Abbildung 5.13 abgebildet. Sie zeigt, wie ein Portlet mit Metrikinformationen hinzugefügt werden kann.

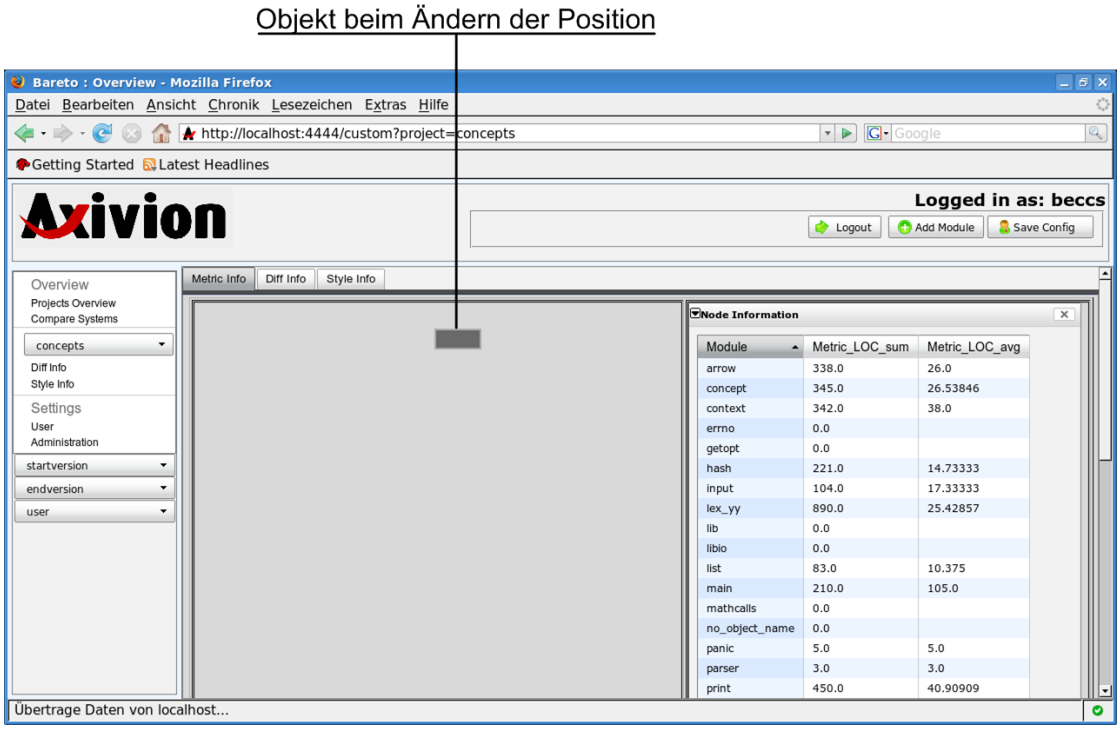


Abbildung 5.11: Verschieben eines Portlets

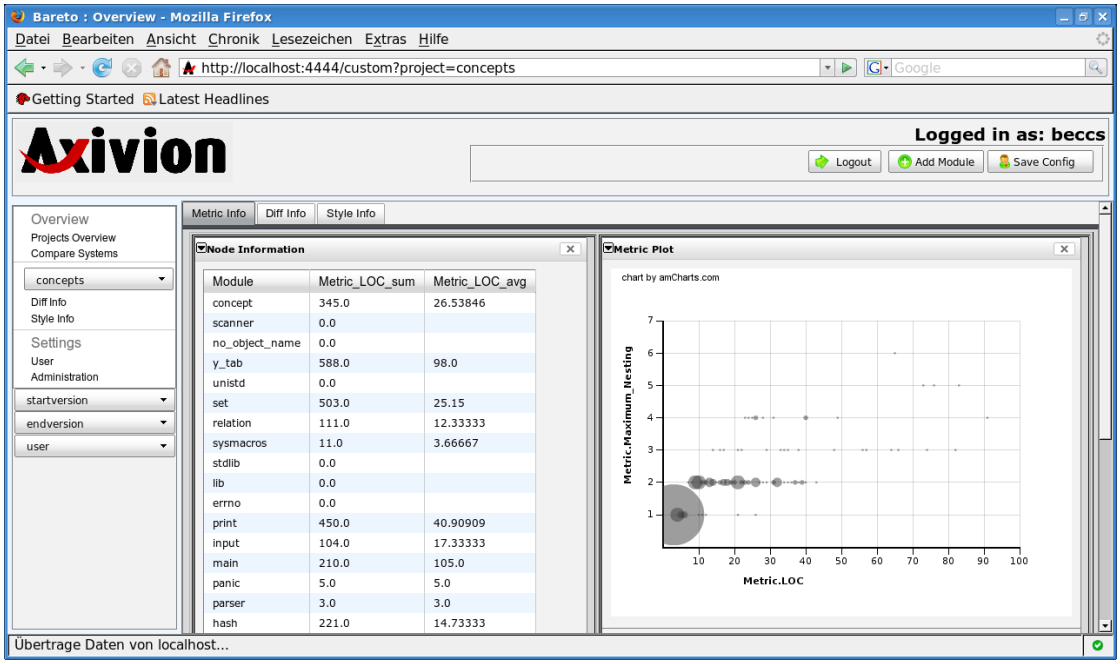


Abbildung 5.12: Veränderte Anordnung nach Verschieben des Portlets

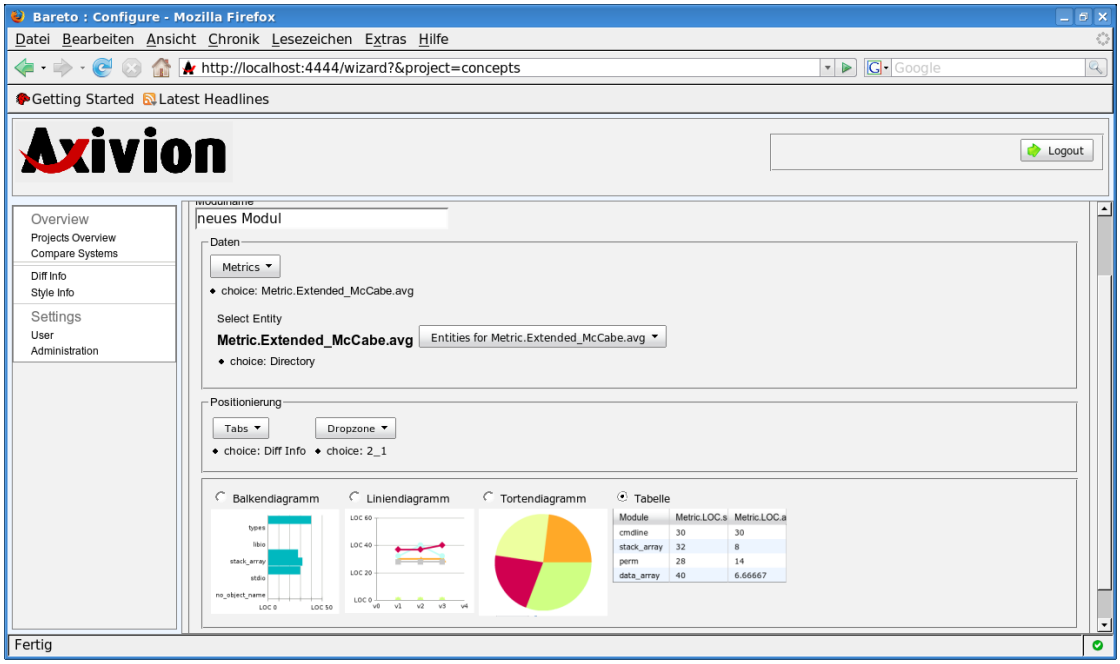


Abbildung 5.13: Hilfeseite zum Hinzufügen von Portlets

## 5.3 Performanz

Die Performanz der Datenbankanfragen spiegelt sich in der Antwortzeit der Portalsoftware wider. Dadurch stellt sie wie bereits in Abschnitt 4.1.2 beschrieben eines der wichtigsten Kriterien im Bezug auf die Akzeptanz der Anwendung beim Nutzer dar. Da die Datenbank großen Einfluss auf die Performanz der Portalanwendung hat, wurden die Testsysteme nach diesem Kriterium vermessen.

Hierfür wurden Beispielanfragen für alle in Bareto umgesetzten Anfragearten formuliert und mit Hilfe eines Werkzeuges zur Zeitmessung ausgeführt. Da lediglich die Datenbank von *SDCC* ausreichende Informationen für die Anfragen zu Klon-, Style- und weiteren Analyseergebnissen enthält, können im Folgenden nur die Anfragen für Metrikwerte über mehrere Versionen verglichen werden. Diese stellen aber auch die wichtigsten Anfragen dar, da die Tabellen mit den Metrikinformationen in beiden Fällen rund 90% der gesamten Anzahl an Datensätzen ausmachen.

Abbildung 5.14 zeigt die Ergebnisse der Messungen für *SDCC*. Es wurden nacheinander die Werte für eine steigende Anzahl von Versionen ermittelt. Wie die Grafik, zeigt bewegen sich die gemessenen Zeiten in einem Bereich von $12ms$ für eine einzige Version, bis zu knapp über einer Sekunde für 80 Versionen.

Die Ergebnisse der Messung von *concepts* bewegen sich, wie zu erwarten in einem noch niedrigeren Bereich von $0ms$ bis $20ms$. Allerdings konnten hier nur die maximale Anzahl von zwanzig Versionen abgefragt werden.

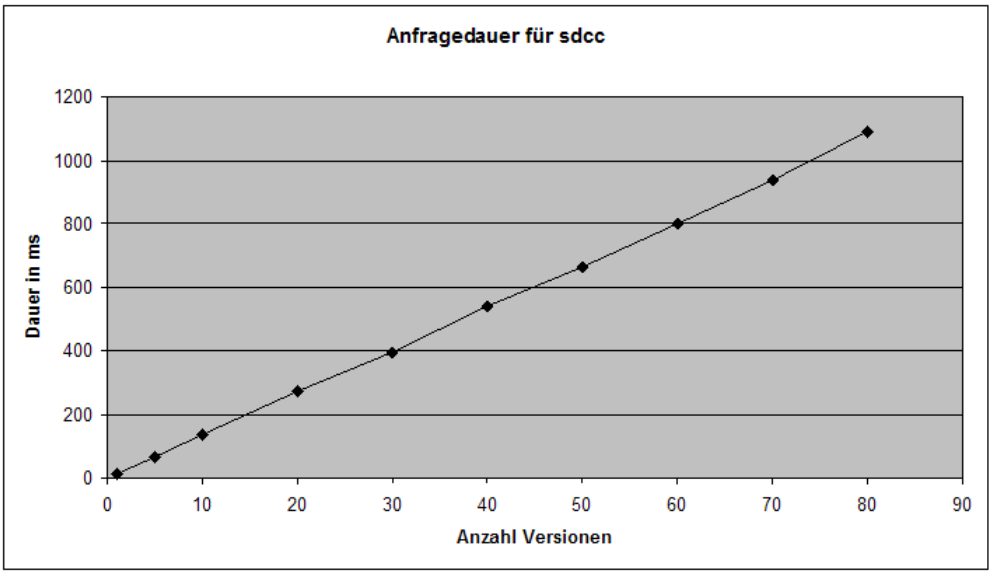


Abbildung 5.14: Ergebnis der Performanzmessung für *SDCC*

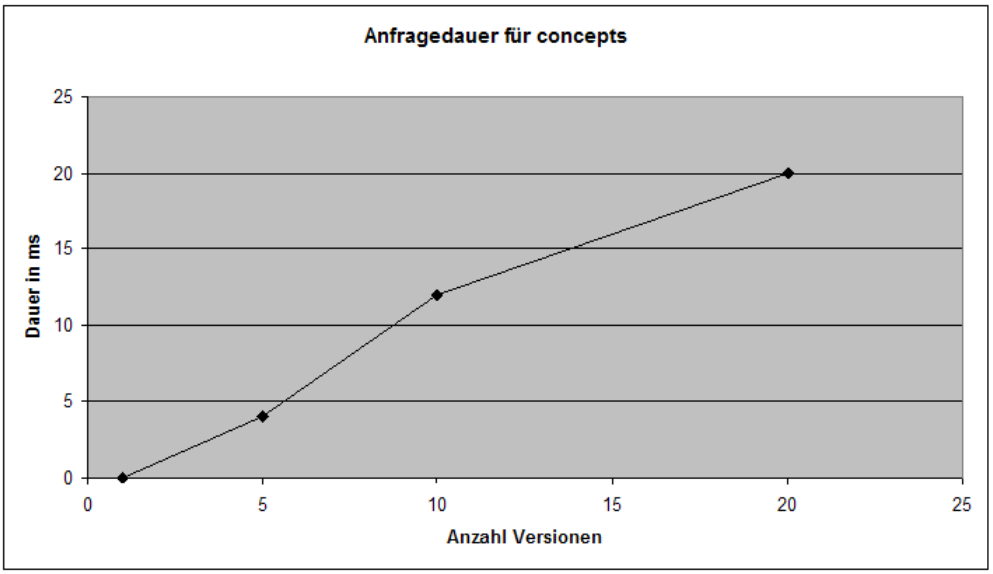


Abbildung 5.15: Ergebnis der Performanzmessung für *concepts*

Die erhobenen Werte zeigen die Erfüllung der Skalierbarkeit und der Performanz der entwickelten Anwendung. Die Anfragen an die Datenbank wurden soweit optimiert, dass selbst für eine Datenbank mit rund 13.500.000 Datensätzen die Antwortzeit bei einer Sekunde lag. Diese Ergebnisse allein geben noch keinen Aufschluss auf die Antwortzeit der Anwendung im Einsatz, da hierfür z.B. weitere Netzwerkeigenschaften mit einbezogen werden müssten. Dennoch zeigen sie, das die Versionierung eines *RFGs* mit Hilfe des Datenbankschemas auch für große Systeme möglich ist.

# 6 Fazit

Dieses Kapitel soll abschließend einen Überblick über die Diplomarbeit geben. Abschnitt 6.1 enthält eine Zusammenfassung der Arbeit sowie eine Beschreibung der aufgetretenen Probleme. Eine Interpretation der Ergebnisse findet sich in Abschnitt 6.2.

## 6.1 Rückblick auf die Arbeit

Das Thema der Arbeit war „Software-Monitoring mit Hilfe eines Dashboards“. Die Idee zu diesem Thema kam ursprünglich von der Axivion GmbH aus Stuttgart. Zu Beginn der Diplomarbeit existierte jedoch keine Möglichkeit, die Daten von mehreren *RFG*-Versionen in Beziehung zueinander zu setzen. Aus diesem Grund wurde von Axivion ein Datenbankschema zur Überführung der Struktur des *RFGs* in eine Datenbank erstellt. Durch Gespräche mit meinem Betreuer und der Axivion GmbH wurden eine ganze Reihe von Anforderungen zusammengetragen. Aus den Anwendungsfällen ergab sich der Bedarf eines Software-Dashboards, dass die Eigenschaften einer Portalsoftware besitzt und sowohl Projektleiter als auch Wartungsprogrammierer als potentielle Nutzergruppen anspricht. Außerdem sollte der entwickelte Ansatz so gestaltet werden, dass die Möglichkeit zur Integration weiterer Analysedaten in der Planung mit bedacht werden sollte.

### Aufgetretene Probleme

Da sich die Axivion GmbH zu Beginn meiner Diplomarbeit noch nicht über die Anforderungen an eine Dashboard-Software im Klaren war und sich auch das Schema der Datenbank noch in der Planungsphase befand, änderten sich im Laufe der Diplomarbeit einige Male die Anforderungen. Durch die Änderungen am Datenbankschema musste immer wieder meine Implementierung angepasst werden. Nachdem ein Großteil meiner Software bereits umgesetzt war, wurde ein komplett neues Datenbankschema entwickelt und eingesetzt. Diese Tatsache kostete viel Zeit, da ich die Änderung nachvollziehen musste, um weiterhin die Testdatenbanken von Axivion nutzen zu können.

Ein weiteres Problem war der Umfang an Techniken, die zum Einsatz kamen und der damit verbundene Einarbeitungsaufwand. Außerdem wiesen einige der eingesetzten Bibliotheken erhebliche Mängel auf, da sie sich noch nicht in einem stabilen Zustand befanden.

## 6.2 Interpretation der Ergebnisse

Das Ziel dieser Diplomarbeit war die Entwicklung eines Software-Dashboards zur Aufbereitung und Darstellung der Ergebnisse der Analysewerkzeuge von Bauhaus mit Hilfe eines web-basierten Systems, das die Eigenschaften einer Portalsoftware besitzt.

Das von mir entwickelte System Bareto erfüllt, wie die beispielhafte Nutzung in Abschnitt 5.2 beschreibt, die Anforderungen an eine Portalsoftware. Sie verfügt über Analyse- und Auswertungsfunktionen und bietet die Daten auf verschiedenen Abstraktionsebenen an. Sowohl verdichtete Kennzahlen für einen Projektleiter, als auch detaillierte Informationen für einen Wartungsprogrammierer werden durch verschiedene Visualisierungsformen repräsentiert.

Die grafische Oberfläche ist personalisierbar und kann interaktiv zur Laufzeit vom Anwender beeinflusst werden. Dadurch kann der Nutzer die Portalsoftware an seine persönlichen Präferenzen und seine Arbeitsweise anpassen. Die Architektur von Bareto ist so gestaltet, dass eine einfache Integration von weiteren Datenquellen möglich ist.

Im Hinblick auf die Eigenschaft der Performanz, belegen die Messwerte, dass die Anfragen an die Datenbank die Antwortzeiten der Portalsoftware nicht negativ beeinflussen. Weiterhin ist aus den Daten ersichtlich, dass Bareto skalierbar ist und sich auch für die Überwachung von großen Systemen eignet.

## 6.3 Schwachstellen

Während der Umsetzungen sind mir einige Schwachstellen in meinem Ansatz aufgefallen. Zum einen ist die Kombination der vielen Techniken sehr fehleranfällig und zum anderen kann dadurch die Kompatibilität und Portabilität nicht gewährleistet werden. Durch die Verwendung von *JavaScript* läuft die Software nicht in allen Web-Browsern fehlerfrei ab. Auch die Darstellung einiger Komponenten der *YUI*-Bibliothek funktioniert nicht überall.

## 6.4 Verbesserungen

Ein Ansatz, um die von mir entwickelte Software zu verbessern, wäre die Reduzierung der eingesetzten Techniken auf ein Minimum. Außerdem werden in der Implementierung noch nicht alle Informationen aus den Analysedaten in der Datenbank dargestellt. Die in Abschnitt 4.4 beschriebene Zugriffsschicht, könnte um weitere Anfragen erweitert werden. Diese könnten mit Hilfe der vorhandenen Portlet-Module ohne großen Aufwand dargestellt werden.

# Literaturverzeichnis


[Basili u. a. 1994] Basili, Victor R. ; Caldeira, Gianluigi ; Rombach, H. D.: *Encyclopedia of Software Engineering - 2 Volume Set*. Kap. Goal Question Metric Paradigm, S. 528–523, John Wiley & Sons, 1994

[Boehm 1981] Boehm, Barry W.: *Software Engineering Economics*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1981. – ISBN 0138221227

[Bommer u. a. 2008] Bommer, Christoph ; Spindler, Markus ; Barr, Volkert: *Software-Wartung – Grundlagen, Management und Wartungstechniken*. Dpunkt Verlag, 2008. – ISBN 9783898644822

[Card u. a. 1999] Card, Stuart K. (Hrsg.) ; Mackinlay, Jock D. (Hrsg.) ; Shneiderman, Ben (Hrsg.): *Readings in information visualization: using vision to think*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999. – ISBN 1-55860-533-9

[Dahm 2006] Dahm, Markus: *Grundlagen der Mensch-Computer Interaktion*. München : Pearson Studium, 2006

[Deißenböck und Seifert 2006] Deissenböck, Florian ; Seifert, Tilman: Kontinuierliche Qualitätsüberwachung mit CONQAT. In: (Hochberger und Liskowsky, 2006), S. 118–125. – ISBN 978-3-88579-188-1

[Diehl 2007] Diehl, Stephan: *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007. – ISBN 978-3-540-46504-1

[DIN EN ISO 9241 1996] 9241, DIN EN I.: *DIN EN ISO 9241-110 Grundsätze der Dialoggestaltung*. Geneva, Switzerland : International Organization for Standardization, 1996

[Ducasse u. a. 2005] Ducasse, Stephane ; Girba, Tudor ; Nierstrasz, Oscar: Moose: an agile reengineering environment. In: Wermelinger, Michel (Hrsg.) ; Gall, Harald (Hrsg.): *ESEC/SIGSOFT FSE*, ACM, 2005, S. 99–102. – ISBN 1-59593-014-0

[Däßler 1999] Dässler, Rolf: Informationsvisualisierung - Stand, Kritik und Perspektiven. In: *Methoden/Strategien der Visualisierung in Medien, Wissenschaft und Kunst* (1999)

[Eick u. a. 2001] Eick, Stephen G. ; Graves, Todd L. ; Karr, Alan F. ; Marron, J. S. ; Mockus, Audris: Does Code Decay? Assessing the Evidence from Change Management Data. In: *IEEE Trans. Software Eng.* 27 (2001), Nr. 1, S. 1–12

[Eisenbarth u. a. 1999] Eisenbarth, Thomas ; Koschke, Rainer ; Plödereder, Erhard ; Girard, Jean-François ; Würthner, Martin: Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. In: *Workshop Software-Reengineering*. Bad Honnef : Universität Koblenz-Landau, 1999 (Fachberichte Informatik 10-99)

[Fenton und Pfleeger 1998] Fenton, Norman E. ; Pfleeger, Shari L.: *Software Metrics: A Rigorous and Practical Approach*. Boston, MA, USA : PWS Publishing Co., 1998. – ISBN 0534954251

[Few 2006] Few, Stephen: *Information Dashboard Design*. München : O'Reilly Media, 2006. – ISBN ISBN 978-0-596-10016-2

[Fjeldstad und Hamlen 1997] Fjeldstad, R. K. ; Hamlen, W.T.: Application Program Maintenance Study – Report to our Respondents. In: *Proceedings of the GUIDE 48*. Philadelphia, PA : IEEE Computer Society Press, 1997

[Gamperl 2007] Gamperl, Johannes (Hrsg.): *AJAX – Grundlagen, Frameworks, APIs*. Bonn : Galileo Press, 2007. – ISBN 987-3-89842-857-6

[Giesecke u. a. 2006] Giesecke, Simon ; Rohr, Matthias ; Hasselbring, Wilhelm: Software-Betriebs-Leitstände für Unternehmensanwendungslandschaften. In: (Hochberger und Liskowsky, 2006), S. 110–117. – ISBN 978-3-88579-188-1

[Großmann und Koschek 2005] Grossmann, Martina ; Koschek, Holger: *Unternehmensportale*. Berlin : Springer-Verlag, 2005. – ISBN 978-3-540-22287-3

[Hochberger und Liskowsky 2006] Hochberger, Christian (Hrsg.) ; Liskowsky, Rüdiger (Hrsg.): *Informatik 2006 - Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*. Bd. *94*. GI, 2006. (LNI). – ISBN 978-3-88579-188-1

[IEEE 1061 1992] IEEE: *IEEE 1061:1992 IEEE Standard for a Software Quality Metrics Methodolog*. New York : Institute of Electrical and Electronics Engineers, 1992

[IEEE 1219 1998] IEEE: *IEEE 1219:1998 IEEE Standard for Software Maintenance*. New York : Institute of Electrical and Electronics Engineers, 1998

[ISO/IEC 9075 1992] Standardization, International O. for: *ISO/IEC 9075:1992 Informationstechnik – Datenbanksprachen – SQL*. Geneva, Switzerland : International Organization for Standardization, 1992

[ISO/IEC 9126 2001] Standardization, International O. for: *ISO/IEC 9126:2001 Software Engineering – Product Quality*. Geneva, Switzerland : International Organization for Standardization, 2001

[Maletic u. a. 2002] Maletic, Jonathan I. ; Marcus, Andrian ; Collard, Michael L.: ATask Oriented View of Software Visualization. In: *VISSOFT*, IEEE Computer Society, 2002, S. 32–. – ISBN 0-7695-1662-9

[Nance und Arthur 2002] Nance, Richard E. ; Arthur, James D.: *Managing Software Quality.* Berlin : Springer-Verlag, 2002. – ISBN 1-85233-393-6

[Parnas 1994] Parnas, David L.: Software aging. In: *ICSE '94: Proceedings of the 16th international conference on Software engineering.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1994, S. 279–287. – ISBN 0-8186-5855-X

[Rötschke 2006] Rötschke, Tobias: Metamodellbasierte Generierung von kundenspezifischen Software-Leitständen. In: (Hochberger und Liskowsky, 2006), S. 95–102. – ISBN 978-3-88579-188-1

[Swanson 1976] Swanson, E. B.: The dimensions of maintenance. In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1976, S. 492–497

[Ware 2000] Ware, Colin: *Information Visualization: Perception for Design.* The Morgan Kaufmann Publishers, 2000. – ISBN 1-558-60511-8

[Young und Munro 1998] Young, Peter ; Munro, Malcolm: Visualizing Software in Virtual Reality. In: *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*, IEEE Computer Society, 1998, S. 19–