SOFTWARESICHTEN: EINE QUELLCODE-ABSTRAKTION ZUM PROGRAMMVERSTEHEN

Hans-Gerd Köhler, Heinrich Rust, Frank Simon

Lehrstuhl Software- und Systemtechnik, Technische Universität Cottbus, Deutschland
E-mail: (hgk | rust | simon)@informatik.tu-cottbus.de

ZUSAMMENFASSUNG

In dieser Arbeit wird eine metrikbasierte und werkzeugunterstützte Technik des Quellcodeverstehens vorgestellt. Sie nutzt die kombinierte Präsentation von Strukturinformationen mit verschiedenen Arten von Metriken innerhalb eines Hypertextdokumentes, um damit verschiedene Systemkomponenten identifizieren, Benutzungs- und Vererbungsbeziehungen erkennen und letztendlich das System verstehen zu können. Für die praktische Validierung dieses Ansatzes wird ein von uns durchgeführtes Experiment und dessen Ergebnisse beschrieben; dabei wurden sowohl herkömmliche Techniken als auch unser Ansatz benutzt, um ein gegebenes Softwaresystem in Subsysteme aufzuteilen. Trotz einiger Einschränkungen, die wir innerhalb des Experimentes machen mußten, wurde unsere grundsätzliche Annahme bestätigt, daß ein grundlegendes Verständnis eines Systems ohne direkte Quellcodebetrachtung möglich ist.

1 Einleitung

Die Verständlichkeit von Software ist eine der wichtigsten Qualitätsfaktoren: In einigen Qualitätsmodellen wird sie direkt gefordert (z.B.. [Grah91], S. 410ff), in einigen anderen wird sie als Parameter betrachtet, der seinerseits viele andere Qualitätsfaktoren beeinflußt (z.B.. Wartbarkeit und Audit-Fähigkeit in [Thom90]). Verständlichkeit kann sicherlich dadurch erhöht werden, daß zusätzliche Dokumentation gepflegt wird. In den meisten Fällen besteht allerdings die Dokumentation ausschließlich aus dem Quellcode selbst. In diesen Fällen hängt die Verständlichkeit sehr stark von dessen Qualität ab.
Verständlichkeit ist allerdings kein ausschließlich internes, sondern eher ein externes Produktattribut (vgl. [FePf96]), Kapitel 9): D.h., sie hängt nicht ausschließlich vom Produkt selbst, sondern darüber hinaus auch von weiteren, externen Faktoren wie Softwaretechnikfähigkeiten des Entwicklers, Werkzeugunterstützung, dem Ziel des Verstehens usw. ab. Unser Ansatz nun versucht, die Verständlichkeit eines Systems zu erhöhen, ohne es selbst zu verändern. Dies erreichen wir durch Verbesserung der externen Attribute, von denen die Verständlichkeit ebenfalls abhängt. In unserem Fall ist dies die Werkzeugunterstützung.
Grundsätzlich ist die Aufgabe des Verstehens großer, existierender Softwaresysteme um so einfacher, je mehr strukturelle Eigenschaften explizit ausgedrückt werden können. Objektorientierte Programmiersprachen unterstützen dieses durch zusätzliche Konzepte, die zusätzliche Strukturen beschreiben (z.B. Klasse, Paket). Wir glauben, daß es positiv ist, diese im Vergleich zu funktionalen oder prozeduralen Softwaresystemen zusätzlichen expliziten Strukturen zu verwenden, um Software verständlicher gestalten zu können.
In dieser Arbeit stellen wir einen Ansatz vor, der beim Verstehen umfangreicher objektorientierter Software hilft. Dabei nutzen wir intensiv die Strukturen objektorientierter Softwaresysteme. Diese werden typischerweise als mathematische Relationen angegeben, z.B. die Benutzt-Relation von Methoden eines Systems, die Vererbungs-Relation zwischen Klassen eines Systems oder die Enthaltensein-Hierarchie von Systemkomponenten auf unterschiedlichen Abstraktionsniveaus.

In dieser Arbeit verwenden wir den Begriff „Komponente“ für Teile eines Softwaresystems auf unterschiedlichen Abstraktionsniveaus. Daher sehen wir sowohl das gesamte System als auch seine Subsysteme, dessen Dateien usw., jeweils als Komponente an.
Das Verständnis von Relationen zwischen Komponenten ist bis zu einer bestimmten Größe des Systems gut durch Graphen unterstützbar; ab einer gewissen Größe allerdings wird ein solcher Graph unübersichtlich und damit wertlos. Exakt mit dieser Situation befaßt sich unsere Arbeit: Denn dann ist es sinnvoll, einige Aspekte bzgl. der Struktur nicht explizit aufzuzählen, sondern quantitativ zu reflektieren. So kann z.B. für ein Element einer Systemkomponente die quantitative Angabe, wieviele andere Systemkomponenten diese benutzen, hilfreich für das Verständnis der gegenseitigen Komponentenabhängigkeiten sein. Oder sie kann verwendet werden, um semantisch kohäsive Subsysteme zu bilden, die die Verständlichkeit des Gesamtsystems erhöhen.

2 Thesen

Unsere Thesen beruhen auf der Annahme, daß die Verständlichkeit von großen Softwaresystemen verbessert werden kann, ohne das Softwaresystem selbst verändern zu müssen. Dies kann durch folgende Maßnahmen erreicht werden:

1) Eine bzgl. qualitätsrelevanter Eigenschaften aufbereitete Sicht auf das System ist für das Verständnis großer Systeme geeigneter als die bloße Quellcodepräsentation. Wenn die richtigen Informationen extrahiert und in einer klar strukturierten Form dargestellt sind, kann diese Präsentation den Quellcode sogar für ein grobes Verständnis der Systemstrukturen ersetzen. Werkzeuge wie *JavaDoc* (vgl. [GoJoSt96], Kapitel 18) oder *CC-Rider* (vgl. [West99]) zeigen bereits in diese Richtung.
2) Eine explizite Präsentation der Enthaltensein-Hierarchie von Systemkomponenten ermöglicht eine schnelle Übersicht über das System. Diese These benutzt das Prinzip „*levels of abstraction*“, das in Myer zum ersten Mal beschrieben wurde [Myer76].
3) Meßwerte von Metriken unterschiedlichen Typs unterstützen den Explorationsprozeß eines Systems und erhöhen dadurch die Verständlichkeit eines Systems. Da während der Exploration verschiedene, vom jeweiligen Kontext abhängige Aspekte relevant sein können, sollten diese Metriken frei definierbar und konfigurierbar sein. Die Mächtigkeit von Metriken und deren effektiver Gebrauch wurde z.B. bereits in [KöRuSi98] gezeigt.
4) Um einen einfachen Zugriff auf die relevanten Informationen zu erhalten, sollte die Navigation zwischen den jeweiligen Präsentationen einzelner, in Relation stehender Komponenten so einfach wie möglich sein. Die Navigation selbst (z.B. was ist von wo aus wie erreichbar) sollte ebenfalls konfigurierbar sein.

Für die Durchführung eines Experimentes haben wir diese Thesen in die folgenden, konkreteren Thesen überführt. Der Schwerpunkt liegt dabei auf der Exploration objektorientierter Systeme:

1') Für ein grobes Softwaresystem-Verständnis genügen die in Abbildung 1 abgebildeten Informationen. Diese sind aus dem Quellcode automatisch extrahierbar und benutzen die zusätzlichen Strukturkonzepte objektorientierter Software.

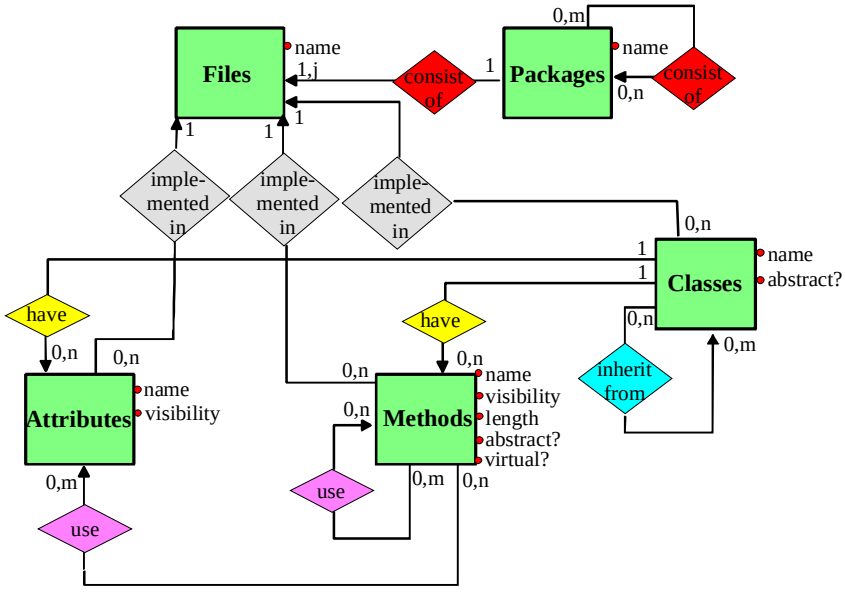

Abbildung 1: Extrahierte Informationen aus dem Quellcode

2') Für die Enthaltensein-Hierarchie des Systems sind die in Abbildung 2 dargestellten Abstraktionsniveaus hilfreich. Diese Niveaus lehnen sich eng an Systeme an, die in den Programmiersprachen C++ oder Java geschrieben sind und im weiteren im Zentrum unseres Interesses stehen.

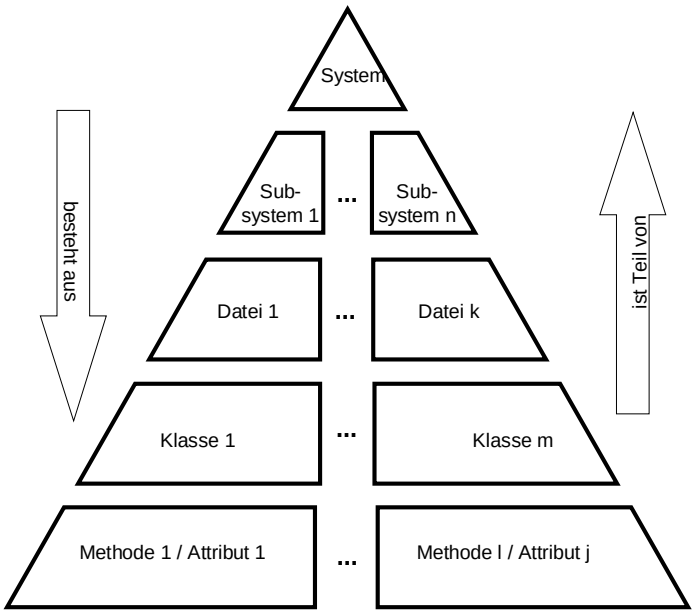

Abbildung 2: Hierarchische Enthaltenseinstruktur

3') Zwei Arten von Metriken sind besonders hilfreich:
Größenmetriken: Sie beschreiben einzelne Komponenten wie z.B. Anzahl von Dateien innerhalb eines Subsystems, Anzahl von öffentlichen Methoden innerhalb einer Klasse oder andere zählbare Teile einer Komponente. Sie helfen, einen Eindruck über die Größe der jeweiligen Komponente zu erhalten.
Kopplungsmetriken: Sie beschreiben die Relationen zwischen den Komponenten wie z.B. die gegenseitige Benutzung von Systemkomponenten (eine Komponente A benutzt eine andere Komponente B, wenn A in ihren Subkomponenten wenigstens eine Methode besitzt, die wenigstens eine Methode aus B bzw. deren Subkomponenten benutzt), Anzahl von anderen Subsystemen, die von einem Subsystem benutzt werden (efferente Subsystem-Kopplung), Anzahl von externen Methoden, die ein Attribut benutzen usw.

4') Für eine einfache Navigation zwischen den Komponentenbeschreibungen haben wir die Dokumente als Hyperdokument organisiert. Jede Systemkomponente ist dort in einem eigenen Dokument beschrieben, das allerdings Verbindungen zu anderen Dokumenten enthält, die in einer Enthaltensein-Relation, Benutzt-Relation oder Vererbungsrelation zu der jeweilig betrachteten Komponente stehen.

3 Das Werkzeug

Unsere Thesen haben wir sowohl theoretisch entwickelt als auch praktisch durch die Implementierung und den Einsatz eines Prototypen mit dem Namen *CrocoBrowse* demonstriert. *CrocoBrowse* realisiert dabei die Anforderungen, die im vorigen Abschnitt beschrieben sind. Eine typische Symboltabelle eines objektorientierten Compiler-Systems enthält alle Informationen, die für unseren Ansatz relevant sind. Unser Werkzeug benutzt die Symboltabelle des CASE-Werkzeugs *SNiFF+* (vgl. [Pfei97]). Aus dieser Symboltabelle werden die Informationen extrahiert, mittels des

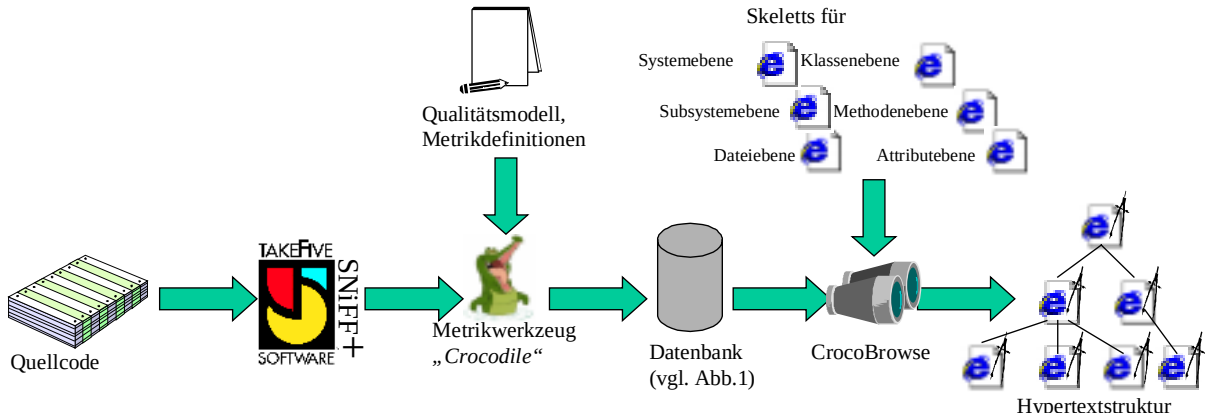

Abbildung 3: Datenfluß vom Quellcode zur HTML-Struktur

Metrik-Werkzeugs *Crocodile*, das an der Technischen Universität Cottbus entwickelt wurde (vgl. [LeSi98]), mit frei definierbaren Metriken vermessen und in einer SQL-Datenbank abgelegt. Diese Datenbank wird anschließend verwendet, um einige HTML-Skeletts mit quantitativen und strukturellen Daten anzufüllen. Jedes einzelne Skelett beschreibt die Struktur und den Inhalt der Dokumentation eines Komponententyps (vgl. Abbildung 2). Eine SQL-ähnliche Sprache innerhalb der Skeletts ermöglicht das Referenzieren von korrespondierenden Meßwerten und strukturellen Informationen. Die Ausgabe unseres Werkzeugs besteht aus mehreren HTML-Dokumenten innerhalb einer hierarchischen Verzeichnisstruktur, die der Enthaltensein-Relation entspricht.

4 Experiment

Nachdem wir ein Werkzeug entwickelt hatten, das die Extraktion und die Präsentation in einer das Verstehen objektorientierter Software unterstützenden Art und Weise ermöglicht, haben wir ein kleines Experiment entworfen, um die Mächtigkeit und eventuelle Unzulänglichkeiten unseres Ansatzes praktisch zu untersuchen. Ein Problem, das sich uns dabei gestellt hat, war die notwendige Operationalisierung von „Verstehen von einem Softwaresystem". Wir haben uns entschieden, unser Tool im Kontext einer typischen Reengineering-Tätigkeit zu untersuchen: Das Aufteilen von gegebenen Klassen eines Systems auf geeignete Subsysteme (vgl. z.B. [AbPeSo98]). Diese Tätigkeit kann als Teilaufgabe des Erstellens einer Systemarchitektur angesehen werden (vgl. [Myer76], Kapitel 5). Unsere Annahme ist, daß die Ergebnisse einer solchen Klassengruppierung bzgl. gesetzter Qualitätsziele um so besser ist, je besser das Verständnis des Softwaresystems ist.
Das Experiment wurde in Form einer Studentenaufgabe im Rahmen einer Vorlesung über Software-Systemtechnik im Wintersemester 1998/1999 durchgeführt. Die Aufgabenstellung bestand im einzelnen aus folgenden Punkten:

- Die Studenten wurden mit zwei objektorientierten C++-Projekten vergleichbarer Größe (60-80 Klassen) konfrontiert: Ein MPEG3-Player namens FREEAMP (vgl. [WoElTh98]) und ein IRC-Client namens cIRCus (vgl. [Wijk98]). Beide Systeme wurden zuvor in der Weise modifiziert, daß alle Dateien und damit auch Klassen in ein und demselben Verzeichnis lagen.
- Die Studentengruppe wurde in zwei Teilgruppen G_1 und G_2 aufgeteilt. Beide Gruppen hatten die Aufgabe, für beide Systeme eine geeignete Subsystemstruktur zu finden, in dem die vorgefundenen Systemkomponenten entsprechend aufgeteilt werden. Für diesen Zweck sollte G_1 das Werkzeug *CrocoBrowse* für das cIRCus-Projekt verwenden und einen beliebigen anderen Ansatz ihrer Wahl für das FREEAMP-Projekt (z.B. die verbreiteten Entwicklungsumgebungen von Microsoft oder Borland, vgl. [Krug96], [Cilw94]). G_2 sollte *CrocoBrowse* für das FEEAMP-Projekt und einen anderen Ansatz für das cIRCus-Projekt verwenden.
- Die Kriterien, nach denen die Subsystembildung erfolgen sollte, wurden explizit innerhalb des Experimentes erläutert: Zwei Klassen gehören zusammen, wenn sie voneinander abhängig sind. Dabei ist eine Klasse von einer anderen abhängig, wenn sie etwas von der anderen verwendet (z.B. eine Methode). Diese Benutztbeziehung kann verursacht sein durch Vererbung, Aggregation oder Assoziation. Das Ergebnis der Klassengruppierung sollte die Beziehungen innerhalb eines Subsystems maximieren und die Beziehungen zwischen den Subsystemen minimieren. Dieses Entwurfsziel wird als *Modul-Unabhängigkeit* bezeichnet (vgl. „*module independence*" in [Myer76]).
- Für eine einfachere spätere Vergleichbarkeit der Ergebnisse haben wir die Zeit für die Klassengruppierung je Projekt auf 120 Minuten beschränkt.
- Die Studenten wurden aufgefordert, zusätzlich zu den Klassengruppierungen Protokolle anzufertigen, die den Prozeß der Gruppierung erläutern. Gefordert waren darin Zeitangaben für einzelne Tätigkeiten und Begründungen für jede vorgenommene Gruppierung.

An dem Experiment nahmen 10 Studenten teil. Unsere Hoffnung war, herauszufinden, welche Arten von Information von den Studenten für die Aufgabe verwendet wurden und was eventuelle Stärken bzw. Schwächen der jeweils gewählten Vorgehensweise waren.

Für die Validierung unserer Thesen haben wir die Ergebnisse des Experimentes in drei Teilen evaluiert, die jeweils Inhalt der nächsten Abschnitte sind:
- Allgemeine Beobachtungen aus den Ergebnissen.
- Untersuchung bzgl. der Konsistenz der vorgeschlagenen Klassengruppierungen.
- Untersuchung von Qualitätsunterschieden zwischen den Ergebnissen der beiden unterschiedlichen Vorgehensweisen.

4.1 Allgemeine Beobachtungen aus dem Experiment

Aus den Protokollen der Studenten haben wir verschiedene Teiltätigkeiten identifiziert, die die Studenten während des Verstehens und des Gruppierens des Systems ausgeführt haben. Diese Tätigkeiten erfolgten in verschiedenen Kombinationen und Intensitäten. Wir haben bei der Untersuchung der Ergebnisse des Experimentes folgende Aspekte berücksichtigt:
- Wieviele Studenten berücksichtigten für die Klassengruppierung
  - Benutztbeziehungen,
  - Innere Klassen (nested classes),
  - Include-Struktur der Dateien,
  - Vererbung,
  - Namensähnlichkeit,
  - Dateizugehörigkeit ?
- Wieviele Studenten haben während der Klassengruppierung tote Klassen gefunden (Klassen, die niemals benutzt werden) ?
- Wieviele Klassen wurden beim Klassengruppieren vergessen?

Die Ergebnisse der Untersuchung dieser Aspekte werden für jede Gruppe einzeln beschrieben:

4.1.1 Beobachtungen der CrocoBrowse-Benutzung

Wie erwartet haben die CrocoBrowse-Benutzer alle diejenigen Aspekte für die Klassengruppierung berücksichtigt, die vom Werkzeug explizit präsentiert wurden. Dies waren vor allen Dingen die Vererbungsbeziehungen und die Benutztbeziehungen. Aufgrund der hierarchischen Struktur der Dokumente und der Links dazwischen wurden fast alle Klassen berücksichtigt. Hinsichtlich des vorgegebenen Qualitätsmodells scheint das Benutzen von CrocoBrowse folglich sehr effektiv zu sein, da es dem Benutzer ermöglicht, alle Kriterien innerhalb eines sehr beschränkten Zeitrahmens zu berücksichtigen.
Wie ebenfalls erwartet wurde, haben die CrocoBrowse-Benutzer nicht explizit präsentierte Aspekte auch nicht berücksichtigt. So hat z.B. kein CrocoBrowse-Benutzer innere Klassen oder die Include-Struktur der Dateien verwandt, da ihnen diese Informationen nicht zugänglich waren.
Eine Interpretation dieses Ergebnisses könnte sein, daß das Quellcodeverstehen sehr einfach durch eine individuelle CrocoBrowse-Konfiguration gesteuert werden kann, indem exakt diejenigen Daten präsentiert werden, die vom Entwickler berücksichtigt werden sollen.

4.1.2 Beobachtungen der CASE-Werkzeug Benutzung

Ein wesentlicher Nachteil von CASE-Werkzeugen wie z.B. Microsoft Visual C++ (cf. [Krug96]) oder Borland C++ (cf. [Cilw94]) ist deren Unfähigkeit, nicht kompilierbaren Code mittels Cross-Referenzer oder Vererbungs-Browser zu betrachten. Die Projekte, die wir für das Experiment gewählt haben, waren nicht direkt kompilierbar. Die mit dem jeweiligen Projekt mit ausgelieferten Makefiles mußten angepaßt werden, Umgebungsvariablen gesetzt werden, Pfade zu Bibliotheken angegeben werden usw. Diese Arbeit war nur schwer in den vorgegebenen 120 Minuten zu bewältigen. Das führte dazu, daß die Studenten, die diese CASE-Werkzeuge verwendet haben, sie eher als bekannte Editoren verwendet haben. Um z.B. auf diese Art und Weise die Vererbungsstruktur zu bestimmen, müssen alle Header-Files geöffnet werden. Die Bestimmung der Benutztbeziehungen ist noch aufwendiger, so daß einige Studenten diese Teilaufgabe wegließen. Einige Studenten reduzierten diese Teilaufgabe auf die Identifizierung klassischer Aggregation bzw. Assoziation, die häufig innerhalb der Header-Files als Objektattribute identifiziert werden können.

CASE-Werkzeuge wie SNiFF+, die auch mit unvollständigem und nicht kompilierbaren Code arbeiten können, zeigten hier deutliche Vorteile: Die Vererbungsstrukturen wurden zügig in Form eines Graphen gedruckt, und auch die Benutztbeziehungen konnten entsprechend aufbereitet werden. Im Gegensatz zu CrocoBrowse vereinfachte die direkte Quellcodebetrachtung das Erkennen von inneren Klassen und der Include-Struktur, da diese Angaben einfach aus dem Quellcode herauszulesen sind.

5 Quantitative Vergleiche der Vorgehensweisen

Um die beiden Vorgehensweisen zum Verstehen gegebener Software zu vergleichen (CrocoBrowse vs. andere Werkzeuge) haben wir zwei unterschiedliche Untersuchungen angestellt: Zuerst haben wir die Konsistenz der jeweilig vorgeschlagenen Klassengruppierungen betrachtet. Ziel war zu zeigen, daß die Gruppierungen nicht zufällig gewählt waren sondern nach reproduzierbaren Kriterien vorgenommen wurden. Ein weiterer Aspekt war die Qualität der vorgeschlagenen Klassengruppierungen: Eine solche Untersuchung war möglich, da wir explizit die Ziele und damit ein diesbezügliches Qualitätsverständnis des Gruppierens genannt hatten. Während des Versuchs haben wir also mit einem „*fixen Qualitätsmodell*" gearbeitet (vgl. [FePf96]).

5.1 Konsistenzvergleich

Um einen Eindruck über die vorgeschlagenen Klassengruppierungen zu erhalten, haben wir das Konzept der Distanzen eingeführt, da dieses innerhalb des menschlichen Wahrnehmungsapparates ein mächtiges Instrument darstellt und in vielfältiger Weise visualisiert werden kann. Dabei sollten zwei ähnliche Klassengruppierungen eine kleine Distanz zueinander aufweisen, wohingegen zwei weniger ähnliche eine größere Distanz zueinander haben sollten. Eine Herangehensweise für eine derartige Distanz kann durch das paarweise Vergleichen der Klassen zweier Klassengruppierungen errechnet werden. Dabei haben wir folgende Annahmen gemacht: Zwei Klassen C_1 und C_2 sind *gleich gruppiert* innerhalb zweier Klassengruppierungen S_1 und S_2 wenn C_1 und C_2 innerhalb der Gruppierung S_1 zur gleichen Gruppe gehören und wenn C_1 und C_2 ebenfalls in S_2 zur gleichen Gruppe gehören.
Unter Verwendung dieser Annahmen haben wir folgendes Distanzmaß definiert (vgl. [FaHa84], Kapitel 9): Die Konsistenz-Distanz (*cd*) von zwei gegebenen Klassengruppierungen S_i, S_j für dieselbe Menge von Klassen X ist definiert als:

$$cd(S_i, S_j) := 1 - \frac{\left(|\{(C_m, C_n), m < n : C_m \text{ und } C_n \text{ sind gleich gruppiert in } S_i \text{ und } S_j\}|\right)}{\left(|\{(C_m, C_n), m < n : C_m \text{ und } C_n \text{ sind gleich gruppiert in } S_i \text{ oder } S_j\}|\right)}$$

Dieses Konsistenzmaß ist normalisiert zwischen 0 bis 1, $cd(S_i, S_i) = 0$ und $cd(S_i, S_j) = cd(S_j, S_i)$.
Die Berechnung von CD für jede Klassengruppierung führt zu einer symmetrischen Distanzmatrix, in der die Elemente $a_{i,j}$ die Werte von $cd(S_i, S_j)$ repräsentieren. Tabelle 1 zeigt einige statistische Charakteristika der berechneten Distanzen für die zwei Projekte im Vergleich zu einer zufällig gewählten Klassengruppierung.

Univariate operation/project	FreeAmp-Project	cIRCus-Project	Randomly Clustering
Median	0.704	0.681	0,855
Mean	0.679	0.687	0,877
Standard deviation	0.092	0.124	0.033
Min/Max	0.498 / 0.829	0.410 / 0.893	0.782 / 0.930

Table 1: Some univariate operations on the data of the experiment

Interpretation: Die meisten Gruppierungsvorschläge der Studenten waren konsistenter als die zufälligen (vgl. Median und Durchschnitt). Die höheren Standardabweichungen (und damit die größeren Max- und Min-Werte) reflektieren einige extreme Gruppierungsvorschläge (z.B. hat ein Student bis auf zwei Klassen alle anderen einem einzigen Subsystem zugeordnet).

Für die Visualisierung beider Vorgehensweisen (CrocoBrowse vs. andere Werkzeuge) haben wir *Spring-embedding* und die *Hauptkomponentenanalyse* (*principle component analysis)* verwendet.

5.1.1 Spring-embedding-Visualisierung (SEV)

Das Spring-embedder-Modell ist eine Heuristik, das Graphen mittels gerichteter Kräfte anordnet (vgl. [QuBr79], [Eade84]): Während die Knoten dabei als sich gegenseitig abstoßende Ladungen aufgefaßt werden, ziehen vorkommende Kanten die Knoten mittels Federn (springs) zusammen. In unserem Fall haben alle Knoten Federn zu allen anderen Knoten. Zwei Knoten sind dabei in einem Kräftegleichgewicht, wenn ihre geometrische Distanz im ausgegebenen Graphen gleich der errechneten Distanz von *cd* ist. Zwei Knoten ziehen sich gegenseitig an, wenn die geometrische Distanz höher ist als die Distanz *cd*, und zwei Knoten stoßen sich gegenseitig ab, wenn die geometrische Distanz kleiner ist als die Distanz *cd*.
Ein solcher Spring-embedding-Algorithmus ist innerhalb einer Demo von SUNs JDK bereits implementiert (*GraphLayout*, vgl. [SUN99]). Mit GraphLayout ist es auf einfache Art und Weise möglich, die berechneten Distanzen aller Klassengruppierungen zu visualisieren. Für eine einfachere Identifikation der Klassengruppierung sind die einzelnen Punkte numeriert worden. Abbildung 4 zeigt ein Ergebnis des Spring-embedder-Visualisierers für das cIRCus-Projekt:

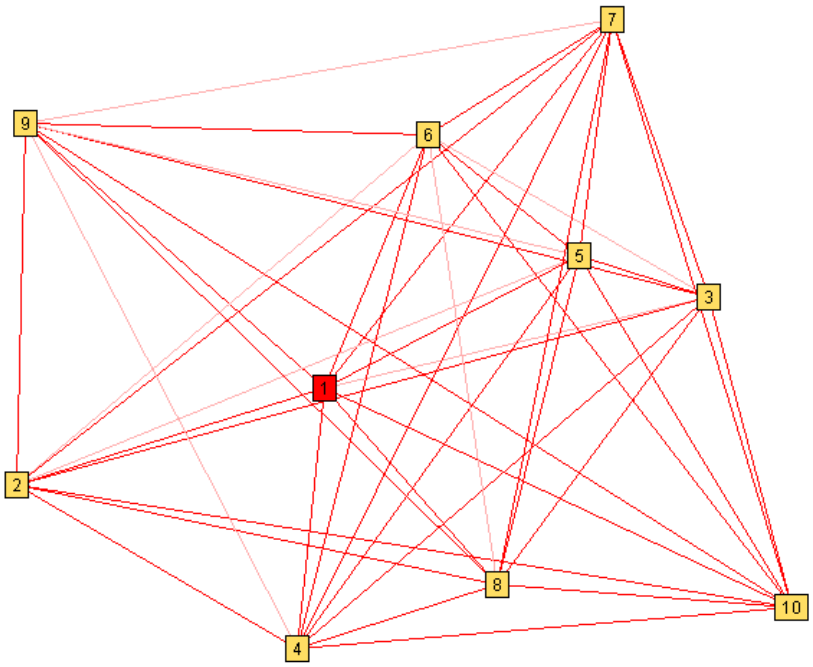

Abbildung 4: Klassengruppierungen des cIRCus-Projekts, SEV

Legende:
Die Knoten 1-4 repräsentieren CrocoBrowse-Benutzer, Knoten 5-10 repräsentieren andere Vorgehensweisen.
Interpretation:
Ein auf dieser Visualisierung basierender Vergleich zwischen beiden Vorgehensweisen ist nur sehr schwer vorzunehmen. Offensichtlich existiert kein großer Konsistenzunterschied zwischen den mit beiden Vorgehensweisen gemachten Klassengruppierungen.

5.1.2 Hauptkomponentenanalyse-Visualisierung, HKAV

Die Hauptkomponentenanalyse ist in vielen Büchern der multivariaten Analyse im Detail beschrieben (vgl. z.B. [Ever93], [FaHa84]). Die grundlegende Technik ist dabei die Transformation von Eingangsdaten $x_1, x_2, ..., x_n$ in eine neue Menge von Variablen $y_1, y_2, ..., y_m$. Die neuen Variablen y_i sind Lineartransformationen der x_i mit den folgenden Eigenschaften:

- $y_1, y_2, ..., y_m$ sind gegenseitig unkorreliert.
- $y_1, y_2, ..., y_m$ repräsentieren die abnehmenden Anteile der Varianz von x_i .

In den Fällen, in denen die ersten errechneten Variablen (die Hauptkomponenten) einen großen Anteil der Varianz der $x_1, x_2, ..., x_n$ beschreiben, können sie als gering-dimensionale Zusammenfassung der Originaldaten verwendet werden. Für den Konsistenzvergleich haben wir drei Dimensionen gewählt. Für die Visualisierung haben wir einen Ansatz gewählt, der bereits von Lund benutzt wurde (vgl. [Lund74]) und im Detail in ([FaHa84], Kapitel 12) erläutert ist. Nach einigen Transformationen der Distanzmatrix und einigen Eigenwertberechnungen haben wir die in Ab-

bildung 5 dargestellte Visualisierung für die Klassengruppierungen des cIRCus-Projektes erhalten. Als Visualisierungsfrontend haben wir *Mathematica* verwendet (vgl. [Wolf99]).

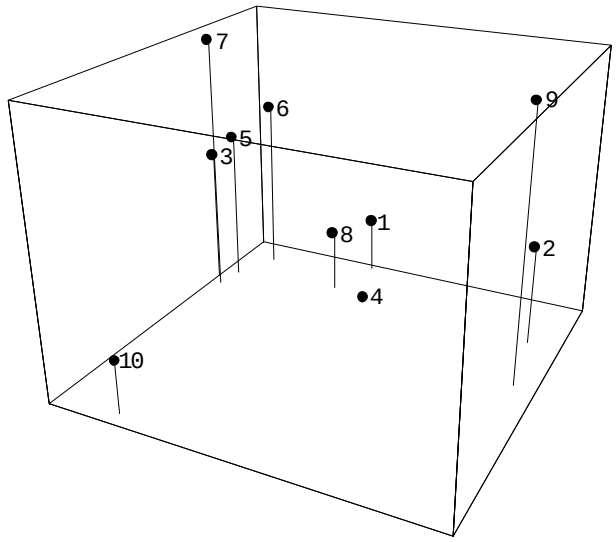

Abbildung 5: Klassengruppierungen vom cIRCus-Projekt, HKAV

Legende:
Die Punkte 1-4 repräsentieren die CrocoBrowse-Benutzer, Punkte 5-10 repräsentieren die anderen Vorgehensweisen.
Statistische Aussagekraft der Visualisierung:
Wir haben die statistische Aussagekraft entsprechend [Much92], Kapitel 3 für jede Dimension berechnet (dabei haben wir nur positive Eigenwerte berücksichtigt, da der einzig negative fast 0 war): X-Achse: 30%, Y-Achse: 26%, Z-Achse: 15% → 71% der Varianz der Originaldaten sind dargestellt.
Interpretation:
Die CrocoBrowse-Klassengruppierungen scheinen ein wenig konsistenter zu sein als die anderen. Der offensichtlichste Außenseiter (2) ordnete bis auf zwei Klassen alle anderen in ein einziges Subsystem. Deswegen besteht keine große Konsistenz zu den anderen Klassengruppierungen. Die größere Inkonsistenz der anderen Werkzeugbenutzer kann mit den unterschiedlichen berücksichtigten Kriterien für die Gruppierung begründet werden. Trotzdem ist der Unterschied zwischen beiden Vorgehensweisen nicht sehr offensichtlich.

5.2 Qualitätsvergleich

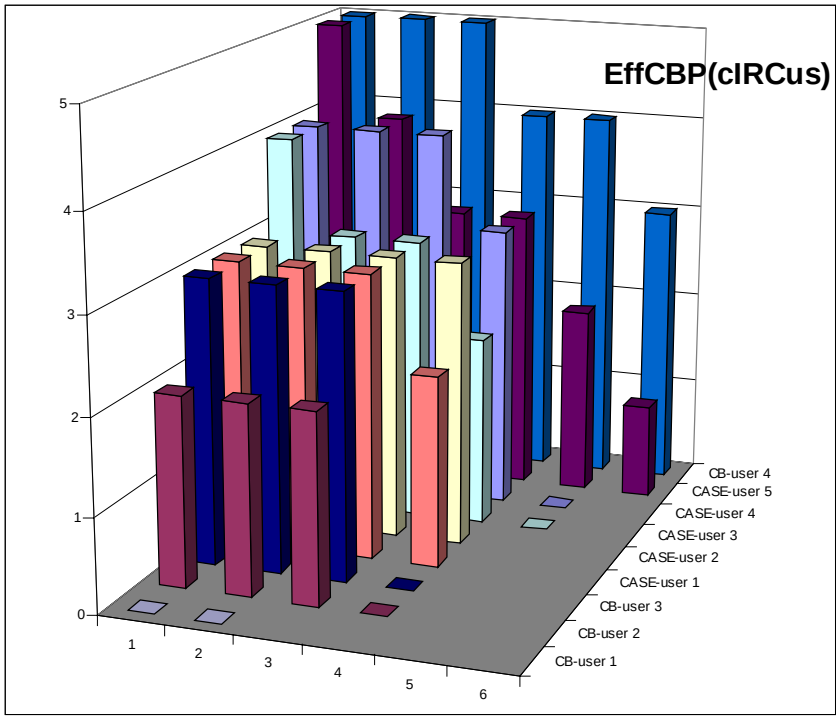

Abbildung 6: Qualitätsbewertung für Klassengruppierungen des cIRCus-Projekts

Das Ziel der Gruppierung war klar definiert: Klassen, die zusammengehören, sollten einem gemeinsamen Subsystem zugeordnet werden. Dabei machten wir explizit, was wir unter dem „Zusammengehören" zweier Klassen verstehen, nämlich das gegenseitige Benutzen von etwas aus einer anderen Klasse. Ein entsprechend dieser Vorgaben abgeleitetes Qualitätskriterium kann z.B. die Anzahl von extramodularen „Zusammengehörigkeits-Relationen" sein. Wir haben die dazugehörige Metrik *efferente Kopplung zwischen Paketen* genannt (EffCBP). Diese Metrik berechnet für jedes vorgeschlagene Subsystem die Anzahl von anderen Subsystemen, von denen irgendetwas (z.B. eine Methode) benutzt wird. In der Abbildung 6 ist der Benutzer auf der Z-Achse dargestellt, die jeweils vorgeschlagenen Subsysteme auf der X-Achse, und auf der Y-Achse ist für jedes Paket der Wert der EffCBP-Metrik dargestellt. Die Benutzer sind sortiert nach der Quadratsumme der EffCBP-Werte ihrer Subsysteme.

Interpretation

Der jeweilige EffCBP-Wert muß im Zusammenhang mit der Anzahl der Klassen innerhalb der vorgeschlagenen Klassengruppierung gesehen werden. Anderenfalls wäre entsprechend unserer Zielvorgabe die beste Qualität durch das Zuordnen aller Klassen in ein Subsystem erreicht. Diese Klassenanzahl-Verteilung ist im nächsten Diagramm durch die Metrik *Anzahl von Klassen innerhalb eines Paketes* (NoclassesP) dargestellt (vgl. Abbildung 7). Die Benutzerbezeichner entsprechen dabei denen der oberen Abbildung. Ein Qualitätsvergleich zwischen den Klassengruppierungen kann nur durch die kombinierte Betrachtung beider Diagramme geschehen:

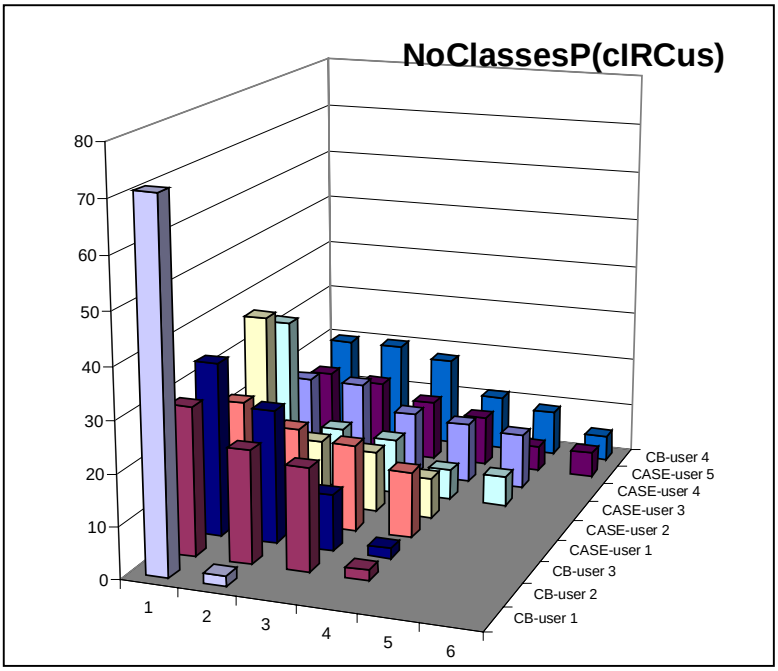

Abbildung 7: Klassenverteilung der Klassengruppierungen des cIRCus-Projekts

- CB-Benutzer 1 hat keine hochwertige Gruppierung vorgeschlagen: Die EffCBP-Metrik zeigt zwar gute Werte, allerdings zeigen die korrespondierenden NoClassesP-Werte, daß fast alle Klassen einem Subsystem zugeordnet wurden.
- Die Klassengruppierung von CB-Benutzer 2 ist besser als die von CB-Benutzer 3: Trotz gleicher Anzahl von Gruppierungen hat ersterer ausgewogenere NoClassesP-Werte und bessere EffCBP-Werte.
- CB-Benutzer 4 hat keine hochwertige Gruppierung vorgeschlagen: Er erstellte 6 stark voneinander abhängige Subsysteme mit dem Effekt, daß kein Subsystem einfach wiederzuverwenden ist, da es jeweils von vielen anderen Subsystemen abhängt.
- Ein genereller qualitativer Vergleich der beiden Vorgehensweisen ist aufgrund der geringen Teilnehmerzahl dieses Experimentes schwierig.

Die Untersuchung des FreeAmp-Projektes führte zu ähnlichen Ergebnissen, nämlich daß ein genereller Vergleich zwischen den Klassengruppierungen der beiden Vorgehensweisen sehr schwierig ist. Daher sind die Qualitätsdiagramme dieses Projektes hier nicht dargestellt.

6 Ergebnisse / Zusammenfassung

Das Hauptergebnis unseres Experimentes war, daß innerhalb der gleichen Zeit die Studenten, die CrocoBrowse für das Verstehens eines Projektes verwendet haben, mehr unterschiedliche Informationen berücksichtigen konnten als dies unter Verwendung anderer Vorgehensweisen möglich war. Dieses läßt sich mit der expliziten, kondensierten und mit Metriken angereicherten Darstellung erklären, die das Erkennen dieser Informationen vereinfacht. Die Ähnlichkeit der Ergebnisse beider Vorgehensweisen belegt, daß solche Softwaresichten eine mächtige Alternative zum klassischen Softwarelesen darstellen. Die Vorteile sind u.a. die strukturiertere, homogenere und von Quellcodedetails wie z.B. Einrückung unabhängigere Präsentation.

Obwohl CrocoBrowse nur ein Prototyp mit einigen Unzulänglichkeiten ist (z.B. fehlende Informationen wie die Include-Struktur) scheint es einen vielversprechenden Weg aufzuzeigen. Es ist möglich, vom detaillierten Quellcode zu abstrahieren und sich auf qualitätsrelevante Eigenschaften zu konzentrieren. Das Verstehen wird durch die Art der extrahierten Daten dominiert; die Metriken unterstützen den Benutzer beim „browsen“ durch die Softwaresicht.

Die Möglichkeit der Konfiguration von CrocoBrowse und die damit gegebene Möglichkeit, sich auf verschiedene Aspekte zu konzentrieren, erscheint ebenfalls vielversprechend und bedarf weiterer Untersuchung.

7 Referenzen

[AbPeSo98]Fernando Brito e Abreu, Concalo Pereira, Pedro Sousa: *"Reengineering the modularity of object oriented systems"*, pres. on Workshop "Techniques, tools and formalisms for capturing and assessing the Architectural quality in object oriented software", ECOOP98, Brussel 98

[Cilw94] Paul Cilwa: „*Borland C++ insider*“, Wiley, New York, 1994

[Eade84] Eades, P.: „*A Heuristic for Graph Drawing*“, Congressus Numerantium 41, pp. 149-160, 1984.

[Ever93] Brian S. Everitt: "*Cluster analysis*", Hodder & Stoughton, London, third edition, 1993

[FaHa84] Ludwig Fahrmeier und Alfred Hamerle: "*Multivariate statistische Verfahren*", Walter de Gruyter, Berlin, 1984

[FePf96] Norman E. Fenton, Shari Lawrence Pfleeger: "*Software Metrics, a rigorous & practical approach*", International Thomson Computer Press, London 1996

[GoJoSt96] James Goslin, Bill Joy, Guy Steel: *"The Java Language Specification (Java Series)"*, Addison-Wesley Pub,

[Grah91] I. Graham "*Object-Oriented Methods*", Addison-Wesley, Wokingham, UK, 1991

[HeNy97] Mats Henricson, Erik Nyquist: „*Industrial strength C++*“, Prentice Hall PTR, Upper Saddle River, 1997

[LeSi98] Claus Lewerentz, Frank Simon: *"A product metrics tool integrated into a software development environment"*, in Proceedings of "object-oriented product metrics for software quality assessment workshop" edited by Walcelio Melo, Sandro Morasca, Houari A. Sahraoui, at 12th european conference on object-oriented programming, CRIM Montreal, 1998

[Lund74] T. Lund: "*Multidimensional scaling of political parties*", Scand. J. Psychol. 15, pp 108-118, 1974

[KöRuSi98] Gerd Köhler, Heinrich Rust, Frank Simon: *"An assessment of large object ori-ented software systems"*, in Proceedings of "object-oriented product metrics for soft-ware quality assessment workshop" edited by Walcelio Melo, Sandro Morasca, Houari A. Sahraoui, at 12th european conference on object-oriented programming, CRIM Montreal, 1998

[Krug96] David J. Kruglinski: „*Inside Visual C++*“, Microsoft Press, Unterschleißheim, 1996

[Much92] Hans-Joachim Mucha: "*Clusteranalyse mit Mikrocomputern*", Akademie Verlag GmbH, Berlin 1992

[Myer76] Glenford J. Myers: "*Software Reliability - Principles and Practices*", John Wiley & Sons, New York 1976

[Pfei97] Andreas Pfeiffer: *"SNiFF+: eine einheitliche Arbeitsumgebung für große Softwareproject"*, in: OBJEKTspektrum March/April 97, SIGS Conferences, Bergisch-Gladbach, pp 30-34

[QuBr79] Quinn Jr., N. R.; Breuer, M. A.: „*A Force Directed Component Placement Procedure for Printed Circuit Boards*“, IEEE Trans. on Circuits and Systems, CAS-26(6), pp. 377-388, 1979.

[Sun99] Applet-Homepage of SUN: http://java.sun.com/applets/index.html, 1999

[Thom90] R. Thomsett: "*Management implications of object-oriented development*", ACS Newsletter, October, 1990

[Wijk98] Ivo van der Wijk (ivo@cs.vu.nl): „*cIRCus: An IRC-Client*“, Homepage: http://www.nijenrode.nl/~ivo/circus/

[WoElTh98] Jason Woodward, Mark Elrod, Brett Thomas: „*FreeAmp: An MP3-Player for Unix and Windows*“, Homepage: http://www.freeamp.org/

[Wolf99] Stephen Wolfram: „*The mathematica*“, Cambridge University Press, second edition, Cambridge 1999

[West99] Western Wares: *"CC-Rider: C/C++ Code Visualization"*, White Paper from http://www.westernwares.com