Institut für Softwaretechnik
Fachbereich 4 : Informatik

Softwareclustering im Reverse Engineering

Entwurf und Implementation einer Clusteranalyse-Umgebung

Diplomarbeit
zur Erlangung des akademischen Grades eines Diplom-Informatikers
im Studiengang Informatik

Autor
Thomas Bernd
thomasbernd@gmx.net

Betreut von
Prof. Dr. Jürgen Ebert
Dr. Volker Riediger

November 2006

Erklärung

Hiermit erkläre ich, wie in §10 Abschnitt 6.2 der Diplomprüfungsordnung für Studierende der Informatik an der Universität Koblenz-Landau gefordert, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Thomas Bernd

Koblenz, den 30.11.2006

Kurzfassung

Im Rahmen dieser Arbeit wird eine Clusteranalyse-Umgebung durch Kopplung bestehender Lösungen entwickelt.

Um Clusteranalysen auf Softwareelementen durchführen zu können, wird eine Software zur Clusteranalyse an das bestehende Reverse Engineering Tool GUPRO zur Extraktion von Informationen aus Softwareartefakten, wie z.B. Quellcode-Dateien, gekoppelt. Die entstehende Clusteranalyse-Umgebung soll alle Services einer Clusteranalyse auf Softwareelementen unter einer Oberfläche vereinen.

Wichtiger Bestandteil der Arbeit ist sowohl die Aufbereitung der theoretischen Grundlagen der Clusteranalyse mit Blickpunkt auf das Softwareclustering, als auch ein strukturierter Auswahlprozess der für die zu entwickelnde Umgebung verwendeten Software zur Clusteranalyse. Letztendlich soll der Funktionalitätsumfang der Clusteranalyse-Umgebung auf ein im Voraus, in Form einer Anforderungsdefinition, festgelegtes Maß angehoben werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation der Arbeit

Das Forschungsgebiet des Reverse Engineering erfreut sich seit Mitte der 1990er Jahre an stetig steigendem Interesse. Dazu beigetragen haben unter anderem das „Jahr 2000“-Problem und die Umstellung vieler Altsysteme (Legacy Systems) auf den Euro. Doch abgesehen von diesen speziellen Beispielen, sehen sich Softwaretechniker immer häufiger mit der Aufgabe konfrontiert, Legacy Systems um Funktionalitäten zu erweitern, die Performanz zu verbessern oder das Softwaresystem für neue Hardware-Plattformen, Betriebssysteme oder sonstige Technologien (Anbindung an das Web, Übergang zu OO-Architektur) anzupassen.

Um oben aufgeführte Aufgaben der Wartung und des Reengineering von Softwaresystemen bewältigen zu können, ist es unerlässlich, das zu modifizierende Softwaresystem zu verstehen. Da Softwaresysteme im Laufe der Zeit - oftmals von verschiedenen Programmierern - häufig modifiziert werden, findet ein zunehmender Strukturverfall (Entropie) des Quellcodes statt. Die Dokumentation eines Softwaresystems - falls überhaupt vorhanden - stimmt nicht mehr mit der aktuellen Struktur der Software überein. Die an der Erstellung der Software beteiligten Personen sind meist nicht mehr greifbar.

In verschiedenen Veröffentlichungen und Fallstudien wurde belegt, dass im Prozess der Wartung bzw. des Reengineerings eines Softwaresystems mehr als die Hälfte der Zeit zum Verstehen des Systems benötigt wird (siehe [Kos00]).

Clusteranalysen stellen einen Ansatzpunkt dar, um den Prozess des Softwareverstehens durch (halb-)automatische Verfahren zu unterstützen und zu beschleunigen und dem Softwaretechniker eine abstrakte Sicht auf das System zu ermöglichen.

Clusteranalysen verfolgen das Ziel, Objekte in verschiedene Gruppen einzuteilen, sodass Objekte innerhalb einer Gruppe einen hohen Zusammenhang und Objekte aus verschiedenen Gruppen einen geringen Zusammenhang aufweisen. Ein hoher innerer Zusammenhang (Kohäsion) der Softwareelemente einer Komponente und eine geringe Kopplung zwischen den Komponenten sind Qualitätsmerkmale einer Softwarearchitektur, die sich auf Wartbarkeit, Wiederverwendbarkeit, Erweiterbarkeit, Flexibilität und Portabilität auswirken. Das Ziel von Clusteranalysen überschneidet sich also mit dem Ziel der Wiedergewinnung einer abstrakten Sicht auf ein Softwaresystem.

Aus dieser Erkenntnis heraus, hat sich das Softwareclustering als eigenständige Disziplin entwickelt. Softwareclustering befasst sich mit der Einteilung von Softwareelementen in Gruppen. Die bisher viel versprechenden Ergebnisse des Softwareclustering und das Poten-

tial zur Unterstützung von Aufgaben im Software-Lebenszyklus gaben den Ausschlag, sich mit diesem Themengebiet tiefgründig zu befassen.

1.2 Ziel und Ausgangspunkt der Arbeit

Zur (halb-)automatischen Einteilung von Softwareelementen in Gruppen wird eine Clusteranalyse-Software benötigt. Zwar existieren bereits einige Software-Lösungen zur Clusteranalyse, diese weisen jedoch spezifische Mängel bzw. Einschränkungen in der Freiheit des Experimentierens für das Softwareclustering auf.

Um Softwareelemente in Gruppen einteilen zu können, müssen diese in Form von Daten vorliegen. Der Quellcode einer Software selbst eignet sich nicht als direkter Input einer Clusteranalyse, dieser muss erst in einer geeigneten Form aufbereitet werden. Die Extraktion der relevanten Daten aus dem Quellcode und die Bereitstellung derselben auf einem angemessenen Abstraktionsniveau, ist Aufgabe eines Reverse Engineering Tools. Eine Software zur Clusteranalyse setzt auf den durch das Reverse Engineering Tool aufbereiteten Daten auf.

Im Rahmen dieser Arbeit soll eine Umgebung zur Clusteranalyse entwickelt werden. Diese Umgebung soll jedoch nicht von Grund auf neu entwickelt werden, sondern sich im Sinne des Wiederverwendungsgedankens und nach dem Prinzip des COTS (commercial of-the-shelf, siehe [MSPB00]) die Funktionalität existierender Software zunutze machen. Die Clusteranalyse-Umgebung soll dem Benutzer größtmögliche Freiheit zur Konfiguration einer Clusteranalyse gewähren und so vielfältiges Experimentieren ermöglichen.

Als Reverse Engineering Tool zur Extraktion von Informationen aus dem Quellcode wurde in gegenseitiger Übereinstimmung zwischen dem Autor dieser Arbeit und den Betreuern von vorneherein das Tool `GUPRO`[1] festgelegt. Diese Entscheidung wird in Kapitel 6 begründet und das Tool vorgestellt.

1.3 Aufbau der Arbeit

Um dem Leser das theoretische Fundament, auf dem diese Arbeit aufsetzt, zu vermitteln, wird in Kapitel 2 das Themengebiet der Clusteranalyse eingeführt. Dabei werden dem Leser die Verfahren der Clusteranalyse an praktischen und meist visualisierten Beispielen näher gebracht. Abschnitt 2.4 des Kapitels befasst sich speziell mit dem Gebiet des Softwareclustering. Die Besonderheiten und der „state of the art“ dieser Teildisziplin der Clusteranalyse werden hier aufgearbeitet.

Aufbauend auf den theoretischen Grundlagen werden in Kapitel 3 die Anforderungen an die zu erstellende Clusteranalyse-Umgebung definiert. Dieses Kapitel beschreibt die Anforderungen im Fließtext. Eine detaillierte Anforderungsliste findet sich in Anhang A. Die Anforderungen beider Kapitel sind folgendermaßen untergliedert: Zunächst finden sich die

[1] http://www.gupro.de

Anforderungen an die Architektur der Clusteranalyse-Umgebung gefolgt von den Anforderungen an das Reverse Engineering Tool zur Extraktion von Informationen aus Quellcode. Im Anschluss werden die Anforderung an die Clusteranalyse-Funktionalität, die Benutzerschnittstelle und die Programmierschnittstelle der Clusteranalyse-Umgebung definiert. Abschließend finden sich allgemeine Anforderungen.

Kapitel 4 beschreibt den strukturierten Prozess der Auswahl einer geeigneten Software zur Clusteranalyse, die als wichtiger Bestandteil der Clusteranalyse-Umgebung fungieren soll.

Die letztendlich ausgewählte Software `Yale` wird in Kapitel 5 unter softwaretechnischen Gesichtspunkten vorgestellt. Neben den entscheidenden Konzepten von `Yale`, interessiert hier vor allem die Programmierschnittstelle.

Kapitel 6 führt den zweiten wichtigen Baustein der Clusteranalyse-Umgebung, das Reverse Engineering Tool `GUPRO`, ein.

Kapitel 7 beschreibt den Entwurf der Clusteranalyse-Umgebung. Hier wird die Kopplung der zuvor in Kapitel 5 und 6 eingeführten Komponenten vorgestellt und wichtige Entwurfsentscheidungen getroffen.

Die Kapitel 8 und 9 stellen die Erweiterungen der in Kapitel 6 ausgewählten Clusteranalyse-Software `Yale` vor. Sowohl der Entwurf, als auch die Umsetzung der Erweiterungen werden hier beschrieben. Kapitel 8 befasst sich mit den notwendigen Erweiterungen zur Anbindung der Software `GUPRO` an `Yale`, Kapitel 9 mit den Erweiterungen der Clusteranalyse-Funktionalität.

Die erstellte Clusteranalyse-Umgebung wird in Kapitel 10 an einem praktischen Beispiel erprobt. Dieses Kapitel soll die Einsatztauglichkeit der Umgebung zeigen und dient gleichzeitig als ein Tutorial für Anwender der Umgebung.

Im abschließenden Kapitel 11 wird eine Bewertung und Zusammenfassung der vorliegenden Arbeit durchgeführt. In einem Ausblick werden anschließend die Einsatzmöglichkeiten der Clusteranalyse-Umgebung im praktischen Umfeld diskutiert.

Anhang A enthält die detaillierte Anforderungsliste.

In Anhang B werden die im Rahmen dieser Arbeit realisierten Softwarekomponenten zusammengefasst. Außerdem findet sich eine Übersicht der erstellten Operatoren.

2 Clusteranalyse

Das Sammeln und Verarbeiten von Informationen ist seit jeher eine unerlässliche menschliche Aktivität, die zur Weiterentwicklung der menschlichen Spezies beigetragen hat und weiterhin beiträgt. Im Laufe der Evolutionsgeschichte des Homo Sapiens kamen immer mehr Informationen auf, die von Generation zu Generation mündlich weitergegeben wurden. Der Umfang der Informationen nahm irgendwann ein solches Ausmaß an, dass der Mensch danach strebte, die Informationen schriftlich festzuhalten. Nicht ohne Grund zählt die Erfindung des Buchdrucks von Johannes Gutenberg gegen Mitte des 15. Jahrhunderts zu einer der revolutionärsten Entwicklungsschritte der Menschheit, da dieser es ermöglichte, große Mengen von Informationen zu dokumentieren und über regionale Grenzen hinweg anderen Menschen zugänglich zu machen.

Die Erfindung und Einführung von Computern katapultierte die Menschheit aufgrund ihrer immensen Speicherkapazität und Rechenleistung in ein neues Informationszeitalter. In der modernen Informationsgesellschaft schiebt sich die Information als entscheidender Produktionsfaktor zunehmend in den Vordergrund. Sowohl Unternehmen als auch wissenschaftliche (Forschungs-)Einrichtungen versuchen, durch das Ansammeln von riesigen Datenbeständen neue Informationen und Wissen über verschiedene Sachverhalte zu erlangen. Diese Massen an Rohdaten sind jedoch für den Menschen meist nicht mehr zugänglich, sodass eine Forschungsrichtung entstand, die versucht aus diesen Daten aufbereitetes Wissen zu generieren. Diese Forschungsrichtung nennt sich Data Mining und ein wichtiger Teilbereich ist die Clusteranalyse.

2.1 Einführung

Clusteranalysen verfolgen das Ziel, durch systematische Informationsverdichtung in einer Menge von Datenobjekten eine charakteristische Struktur der Objektmenge erkennen zu lassen. Es wird versucht, eine Menge von Objekten so in Gruppen (Cluster) einzuteilen, dass die Objekte innerhalb eines Cluster möglichst ähnlich zueinander sind, wohingegen Objekte aus verschiedenen Clustern möglichst unterschiedlich voneinander sind.

Der Titel des Buches [KR90] bringt es mit „Finding groups in data“ auf den Punkt und grenzt damit die Clusteranalyse direkt von der verwandten Diskriminanzanalyse (siehe [Cac73]) ab. Beide Ansätze verfolgen das Ziel der Klassifizierung von Objekten, bei der Diskriminanzanalyse geht es jedoch darum, Objekte schon (a priori) vorgegebenen Gruppen zuzuweisen, während bei der Clusteranalyse die Gruppen im Laufe des Verfahrens erst gesucht bzw. auf Basis der Objekteigenschaften gebildet werden.

Die Klassifizierung von ähnlichen Objekten in Gruppen spielt in verschiedenen Wissenschaften seit langem eine große Rolle. Schon Mitte des 18. Jahrhunderts wurde in der Biologie damit begonnen, Tiere, Pflanzen und Mineralien aufgrund ihrer Eigenschaften in verschiedene Gruppen einzuteilen. Weitere typische Anwendungen finden sich in der Astronomie, wo z.B. Sterne auf Basis ihrer Lichtintensität und Oberflächentemperatur in verschiedene Kategorien eingeteilt wurden.

Aktuelle Anwendungsgebiete von Clusteranalysen finden sich in zahlreichen Bereichen. Einige dieser Anwendungsgebiete sollen hier, ohne Anspruch auf Vollständigkeit, aufgeführt werden, um einen Eindruck von der konkreten Funktion von Clusteranalysen zu erhalten.

- In den Sozialwissenschaften werden Personen nach ihren Verhaltensweisen, Einstellungen, Leistungen, optischen Merkmalen etc. in Gruppen eingeteilt, um z.B. psychologische Profile, Gruppen leistungsschwacher Schüler, die Bildung extremistischer Gruppen etc. ausfindig zu machen.
- In biologischen und medizinischen Wissenschaften werden mithilfe von Clusteranalysen Pflanzen, Tiere, Krankheiten und Symptome gruppiert. So wird z.B. auf Basis der Gensequenzen eines Lebewesens versucht, dieses einer bestimmten Rasse zuzuordnen bzw. von anderen Rassen abzugrenzen. Weiterhin werden in der Humanmedizin Symptomklassen gebildet, um die Diagnose und Therapie von Krankheitsbildern zu verbessern.
- Marketing erfährt durch die Anwendung der Clusteranalyse seit Mitte der 1990er Jahre einen immensen Schub im Bereich des Customer Relationship Managements. Dort wird unter anderem versucht, Kunden aufgrund ihres Kaufverhaltens, ihrer Reaktion auf Preisänderungen und sonstiger Verhaltensweisen in Kundengruppen einzuteilen oder Marktsegmentierung betrieben, um möglichst homogene Absatzmärkte zu bestimmen.
- Weiterhin findet Clusteranalyse Anwendung in der Geographie (Gruppierung von Regionen), in der Archäologie (Gruppierung von Fossilienfunden) und zahlreichen anderen Bereichen. Diese Liste soll sich aber auf die hier aufgeführten Anwendungsbereiche beschränken.

Ursprünglich wurden Objekte auf Basis des subjektiven Wissens und der Urteilsfähigkeit des jeweiligen Anwenders in Gruppen eingeteilt. Dies ist bei einer begrenzten Anzahl von Merkmalsausprägungen der zu gruppierenden Objekte auch manuell noch möglich, z.B. die Einteilung von Personen auf Basis ihrer Größe und ihres Gewichtes in verschiedene Gruppen. Man stelle sich die zwei Merkmalsausprägungen (Größe und Gewicht) der einzelnen Objekte (Personen) als X- und Y-Achse eines Koordinatensystems vor und zeichnet die Personen entsprechend ihrer Werte bzgl. Größe und Gewicht als Punkte in dieses Koordinatensystem ein (siehe Abbildung 2.1). Es ist eine Leichtigkeit, aus den gegebenen Objekten zwei Cluster zu bilden, nämlich die der Kleinkinder (Cluster A) und die der Erwachsenen (Cluster B). Fügt man zu den Merkmalsausprägungen der Objekte jedoch noch die Augenfarbe, die

Haarfarbe, das Geschlecht, die Blutgruppe und das Herkunftsland hinzu, befinden wir uns in einer Dimension, die weder zeichnerisch festgehalten, noch vom Menschen nachvollzogen werden kann. Deswegen wird unter Zuhilfenahme von Computer und entsprechenden Programmen auf eine automatische Klassifikation gesetzt.

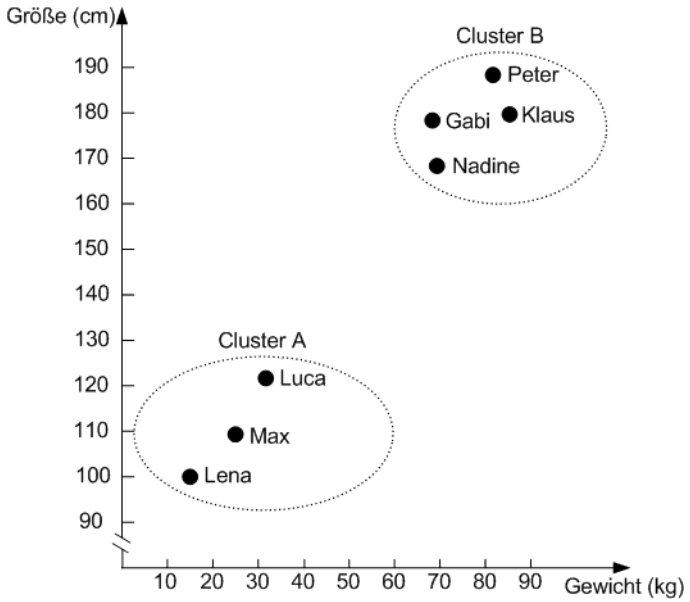


Abbildung 2.1: Größe und Gewicht von sieben Personen in einem Koordinatensystem

Den in Abbildung 2.1 dargestellten Objekten (Personen) wurden bestimmte Merkmale - im Folgenden als Variablen bezeichnet - zugeordnet. Notiert man jedes Objekt als Vektor aus Attribut-Werte-Paaren und schreibt diese Vektoren untereinander, ergibt sich eine nxm-Datenmatrix $X = (x_{ij})$, auch Datentabelle genannt, mit n Objekten und m Variablen - ($i \epsilon \{1, ..., n\}$, $j \epsilon \{1, ..., m\}$). Es handelt sich um eine Objekt-Attribut-Struktur. Der Wert x_{ij} in der i-ten Zeile und j-ten Spalte gibt den Wert der j-ten Variablen des i-ten Objektes an. In einer Datentabelle wird meistens zusätzlich ein ID-Attribut angegeben, das die Objekte eindeutig identifiziert. Tabelle 2.1 zeigt die zu Abbildung 2.1 gehörende Datentabelle.

ID	Variable 1	Variable 2
Name	**Gewicht (kg)**	**Größe (cm)**
Gabi	69	179
Klaus	86	180
Lena	17	100
Luca	31	122
Max	24	111
Nadine	69	170
Peter	83	190

Tabelle 2.1: Datentabelle aus Personen mit den Variablen Gewicht und Größe

Um verschiedene homogene Gruppen mit möglichst ähnlichen Objekten erstellen zu können, müssen die Objekte irgendwie verglichen werden, d.h. es muss festgestellt werden, welche Objekte sich aufgrund ihrer betrachteten Variablenwerte sehr ähnlich sind und welche sich stark voneinander unterscheiden. Es ist also eine Funktion notwendig, die aus den Variablenwerten zweier Objekte einen Wert berechnet, der etwas darüber aussagt, wie ähnlich sich die beiden Objekte sind. Solche Funktionen nennt man Distanz- bzw. Ähnlichkeitsfunktionen, sie werden in Kapitel 2.2 behandelt.

Auf Basis dieser berechneten Ähnlichkeitswerte kann dann eine Einteilung in verschiedene Cluster stattfinden. Dafür gibt es eine Vielzahl an Clusterverfahren, die in Kapitel 2.3 behandelt werden.

Vorgreifend soll an dieser Stelle bereits erwähnt werden, dass die Konfiguration einer Clusteranalyse, d.h. eine Auswahl und Kombination von Verfahren aus der Vielzahl der im Folgenden vorgestellten Distanz- bzw. Ähnlichkeitsfunktionen und Clusterverfahren, stark vom jeweiligen Anwendungsbereich und Untersuchungsziel abhängt. Die Tatsache, dass sich für verschiedene Anwendungsbereiche verschiedene Variationen von Distanz- / Ähnlichkeitsfunktionen und Clusterverfahren eignen, führt zur angesprochenen Vielfalt dieser Verfahren. Der Auswahl der zur Erreichung des Untersuchungsziels optimalen Verfahren sollte große Aufmerksamkeit gewidmet werden.

Ebenso bedeutend ist die Auswahl der Variablen, die die zu gruppierenden Objekte beschreiben. Würde man im Beispiel aus Tabelle 2.1 statt Größe und Gewicht andere Variablen, wie z.B. Augenfarbe, Geschlecht und Blutgruppe heranziehen, würde sich höchst wahrscheinlich eine ganz andere Gruppierung ergeben.

Auch eine angemessene Repräsentation (Visualisierung) der Ergebnisse der Clusterverfahren ist von großer Bedeutung. Clusterverfahren können komplexe Strukturen innerhalb der Daten aufdecken, diese können jedoch nur erkannt werden, wenn sie in einer für den Menschen gut verständlichen Notation vorliegen.

Zusammenfassend kann festgehalten werden, dass der Anwender einer Clusteranalyse in vier Entscheidungsbereichen Einfluß auf eine Clusteranalyse nehmen kann:

- Das Erstellen eines Modells, d.h. die Auswahl der Variablen, die die Objekte beschreiben und die Auswahl der zu gruppierenden Objekte selbst.
- Die Auswahl einer Distanz- bzw. Ähnlichkeitsfunktion.
- Die Auswahl eines Clusterverfahrens.
- Die Auswahl einer geeigneten Visualisierungsmethode.

2.2 Distanz- / Ähnlichkeitsfunktionen

Bei Distanz- / Ähnlichkeitsfunktionen handelt es sich wie bereits erwähnt um Funktionen, die aus den Variablenwerten zweier Objekte einen Wert berechnen, der etwas über die Distanz bzw. Ähnlichkeit der beiden Objekte aussagt.

Formal lässt sich eine solche Funktion für eine Menge $O = \{o_1, o_2, \ldots, o_n\}$ von Objekten definieren als

$$s : O \text{ x } O \rightarrow \mathbb{R}$$

womit also zwei Objekten $o_i, o_j \epsilon O$ eine reelle Zahl $s = s(o_i, o_j)$ oder kurz s_{ij} zugeordnet wird ([SL77]). Die Funktion s wird Ähnlichkeitsfunktion genannt und ihr Ergebnis ist die Ähnlichkeit zwischen den Objekten o_i und o_j. Für eine Ähnlichkeitsfunktion s gelten folgende Axiome:

$$0 \leq s_{ij} \leq 1 \tag{2.1}$$
$$s_{ii} = 1 \tag{2.2}$$
$$s_{ij} = s_{ji} \tag{2.3}$$

Führt man die Ähnlichkeitsfunktion für jedes Objektpaar der Datenmatrix aus, entsteht eine nxn Ähnlichkeitsmatrix $S = (s_{ij})$, wobei sich die n Objekte in Zeile und Spalte gegenüberstehen (Objekt-Objekt-Struktur).

Das Gegenteil der Ähnlichkeit zweier Objekte ist ihre Distanz zueinander, die sich über die Distanzfunktion

$d : O \text{ x } O \rightarrow \mathbb{R}$

berechnet, die zwei Objekten $o_i, o_j \epsilon O$ eine reelle Zahl $d = d(o_i, o_j)$ oder kurz d_{ij} zuordnet. Für eine Distanzfunktion d gelten folgende Axiome:

$$d_{ij} >= 0 \tag{2.4}$$
$$d_{ii} = 0 \tag{2.5}$$
$$d_{ij} = d_{ji} \tag{2.6}$$

Führt man die Distanzfunktion für jedes Objektpaar der Datenmatrix aus, so entsteht eine nxn Distanzmatrix $D = (d_{ij})$, wobei sich die n Objekte in Zeile und Spalte gegenüberstehen (Objekt-Objekt-Struktur).

Die entstehende Distanz- oder Ähnlichkeitsmatrix ist aufgrund der Axiome 2.3 bzw. 2.6 symmetrisch und aufgrund der Axiome 2.2 bzw. 2.5 sind die Diagonalelemente gleich.

Da Distanzen und Ähnlichkeiten nur zwei Seiten derselben Medaille sind - je größer die Distanz desto geringer die Ähnlichkeit - existieren (mehrere) Transformationsfunktionen, von denen hier repräsentativ jeweils eine für jede Richtung aufgeführt wird:

$$s_{ij} = 1 - d_{ij}/a, \quad mit\ a = Max\{d_{ij}\} \tag{2.7}$$

und

$$d_{ij} = 1 - sij \tag{2.8}$$

Im Folgenden werden konkrete Distanz- bzw. Ähnlichkeitsfunktionen vorgestellt, die für zwei Objekte eine Distanz bzw. Ähnlichkeit berechnen. Um Objekte vergleichbar zu machen, muss man die Typen der Variablen betrachten, die zur Beschreibung dieser Objekte verwendet werden, da sich die Funktionen zur Berechnung einer Distanz / Ähnlichkeit je nach Variablentyp unterscheiden. Eine erste grobe Kategorisierung von Variablentypen kann man in qualitative und quantitative Variablen vornehmen. Qualitative Variablentypen lassen sich weiterhin unterscheiden in binäre, nominale oder ordinale Variablen, während quantitative Variablen entweder intervall- oder rational-skaliert sein können (siehe Abbildung 2.2).

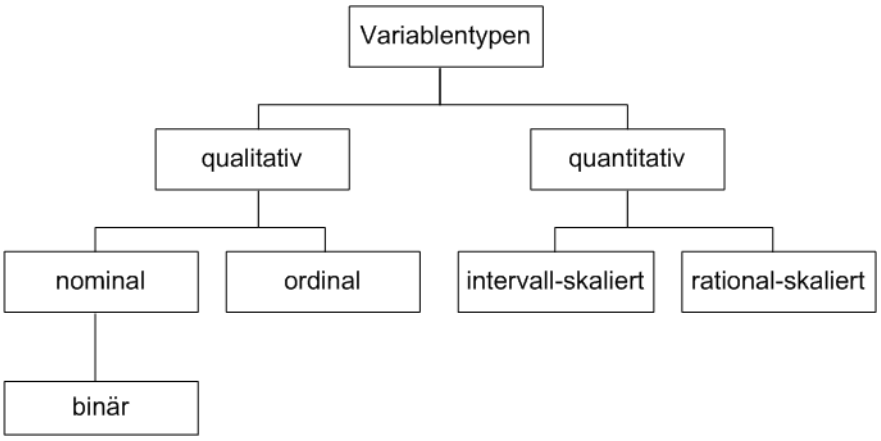


Abbildung 2.2: Variablentypen

Die unterschiedliche Behandlung der einzelnen Variablentypen zur Berechnung einer Distanz bzw. Ähnlichkeit wird in diesem Abschnitt aufgezeigt. In jedem Unterabschnitt wird davon ausgegangen, dass ein Objekt ausschließlich durch Variablen des jeweiligen Typs, den der Unterabschnitt behandelt, beschrieben wird. Zwei Objekte $o_i, o_j\ \epsilon\ O$ mit m Variablen werden als Vektor aus Attribut-Werte-Paaren mit

$$x_i = (x_{i1}, \ldots, x_{im})$$
$$x_j = (x_{j1}, \ldots, x_{jm})$$

beschrieben. In Kapitel 2.2.4 wird der - nicht selten auftretende - Fall behandelt, dass zwei Objekte verglichen werden, die über Variablen verschiedener Typen beschrieben werden. Die folgenden Ausführungen sind angelehnt an [SL77], [KR90] und [Bac96].

2.2.1 Quantitative Variablen

Intervall-skalierte Merkmale werden als Zahl dargestellt, wobei die Differenz zwischen den Werten exakt bestimmt werden kann. Für eine Intervallskala existiert kein natürlicher Nullpunkt[1].

Bei intervall-skalierten Variablen werden die Objekte über m kontinuierliche Messwerte beschrieben, wobei es sich dabei um positive oder negative reelle Zahlen handelt. Ein Objekt lässt sich also als Punkt im m-dimensionalen reellen Zahlenraum $\mathbb{R}^m$ darstellen. Der Preisunterschied (und damit die Distanz) zweier Objekte die 10 bzw. 60 Euro kosten ist gleich dem Preisunterschied zweier Objekte die 150 bzw. 200 Euro kosten. Diese lineare Skalierung gilt für die gesamte Zahlenskala.

Im Gegensatz dazu handelt es sich bei rational-skalierten Variablen stets um positive numerische Messwerte, da die Rationalskala einen natürlichen Nullpunkt hat. Entscheidend ist hier das Verhältnis zwischen zwei Werten. Dieser - eher selten vorzufindende - Variablentyp wird z.B. verwendet um die Abnahme der Intensität der Radioaktivität eines Materials im Zeitverlauf anzugeben: Die Abnahme der Radioaktivität innerhalb eines Jahres von bspw. 100 auf 70 ist gleich der Abnahme von 10 auf 7 (exponentielle Abnahme im Zeitverlauf). Um auf rational-skalierten Variablen dieselben Distanz- bzw. Ähnlichkeitsfunktionen wie auf intervall-skalierten Variablen anwenden zu können, wird in der Praxis oftmals eine *logarithmische Transformation* ($y = log(x)$) eines Wertes x vorgenommen und der Wert y als intervall-skalierte Variable behandelt. Im Folgenden beziehen sich also alle Ausführen in diesem Kapitel auf intervall-skalierte Variablen.

Wie bereits erwähnt, lässt sich ein Objekt mit m intervall-skalierten Variablen als Punkt in einem m-dimensionalen Koordinatensystem einzeichnen. Daher liegt es nahe, die Distanz zweier Objekte als die geometrische Distanz der sie repräsentierenden Punkte im Koordinatensystem zu berechnen. Diese Distanz wird die *Euklidische Distanz* genannt und ist die bekannteste unter den Distanzfunktionen. Die *Euklidische Distanz* zweier Objekte o_i, $o_j \in O$ mit jeweils m Variablen lässt sich berechnen als

$$d_{ij} = \sqrt{\sum_{k=1}^{m}(x_{ik} - x_{jk})^2} \tag{2.9}$$

Abbildung 2.3 zeigt die Herleitung der Formel der Euklidischen Distanz mit m = 2 nach dem Satz des Phytagoras.

Eine weitere häufig verwendete Distanzfunktion, die *Manhattan Distanz*, lässt sich ebenfalls aus Abbildung 2.3 herleiten:

$$d_{ij} = \sum_{k=1}^{m}|x_{ik} - x_{jk}| \tag{2.10}$$

[1] Der willkürlich definierte Nullpunkt der Celsius-Temperaturskala ist kein natürlicher Nullpunkt, während der Nullpunkt der Kelvin-Temperaturskala ein natürlicher Nullpunkt ist, da er einen absoluten Nullpunkt darstellt.

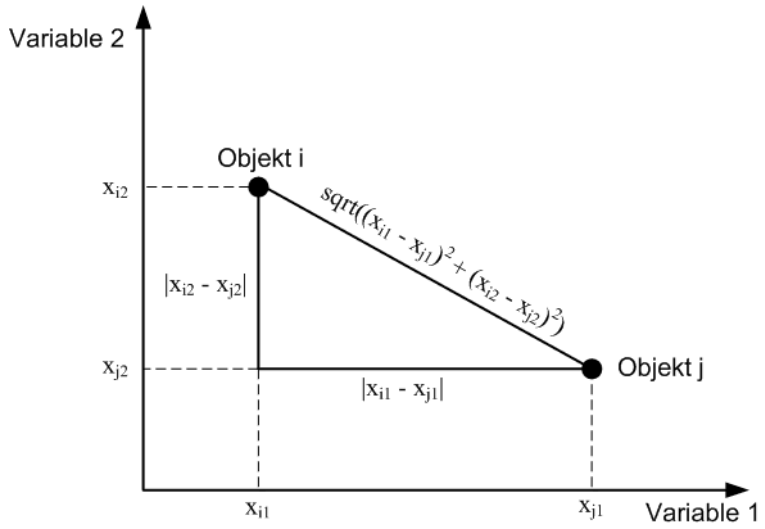


Abbildung 2.3: Distanz zweier Punkte im 2-dimensionalen Koordinatensystem

Diese Distanzfunktion wird auch als *City Block Distanz* bezeichnet, eine Anspielung auf die gitterartige Auslegung des Straßennetzes in einigen Städten der USA, wo man zum Erreichen von Objekt j von Objekt i aus den Weg $|x_{i2} - x_{j2}|$ + $|x_{i1} - x_{j1}|$ nehmen müsste.

Weitere bekannte Distanzfunktionen für intervall-skalierte Variablen sind:

- *Quadratische Euklidische Distanz*

$$d_{ij} = \sum_{k=1}^{m} (x_{ik} - x_{jk})^2 \tag{2.11}$$

- *Chebyshev Distanz*

$$d_{ij} = Max_{k=1}^{m} |x_{ik} - x_{jk}| \tag{2.12}$$

Ein Problem bei Distanzfunktionen stellt die Tatsache dar, dass das Ergebnis ihrer Berechnungen stark von den verwendeten Einheiten abhängt. Nehmen wir als Beispiel die Objekte Klaus und Gabi aus der Datentabelle 2.1 mit den zugehörigen Datenvektoren Klaus = (86, 180) und Gabi = (69, 179) wobei Gewicht in kg und Größe in cm angegeben wurde. Unter Verwendung der *Euklidischen Distanz* ergibt sich ein Distanzwert $d_{Klaus,Gabi} = 17,03$. Gibt man das Gewicht nun in Pfund (pound, 1p = 0,4536 kg) an, ergibt sich nach Umrechnung jeweils ein Datenvektor von Klaus = (190, 180) und Gabi = (152, 179) und somit eine Distanz von $d_{Klaus,Gabi} = 38,01$. Die Distanz hat sich also mehr als verdoppelt, obwohl die Eigenschaften der Objekte noch dieselben sind. Der Grund dafür ist, dass die Variable Gewicht stärker in die Berechnung mit eingeht, da es sich bei der zweiten Berechnung, aufgrund der Umrechnung der Einheit, um größere Zahlenwerte handelt und damit auch die Distanz steigt. Um diesem Problem entgegenzutreten versucht man die Daten vor der Distanzberechnung so zu normieren, dass Werte entstehen, die unabhängig

von einer Einheit sind („unitless“). Dafür wird zunächst für jede Variable $k\epsilon\{1,\ldots,m\}$ das arithmetische Mittel $mean_k = \frac{1}{n}\sum_{i=1}^{n}(x_{ik})$ und anschließend die Standardabweichung $sdev_k = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_{ik})}$ berechnet. Für jedes Objekt o_i wird nun der entsprechende Eintrag k seines Datenvektors über die Formel

$$z_{ik} = \frac{x_{ik} - mean_k}{sdev_k} \tag{2.13}$$

normiert und die Einträge x_{ik} der Datenmatrix durch z_{ik} ersetzt.

2.2.2 Nominale Variablen

Variablen werden als nominal bezeichnet, wenn ihre Werte verschiedene Kategorien annehmen können, diese Kategorien jedoch keiner Ordnung unterliegen. D.h. werden zwei Objekte bzgl. einer nominalen Variable verglichen, kann man lediglich die Aussage treffen, dass sich die beiden Objekte bzgl. der Variablenbelegung unterscheiden oder nicht.

Die binären Variablen stellen einen Spezialfall der nominalen Variablen dar, wobei die Anzahl der möglichen Kategorien (Fälle) auf zwei begrenzt ist. Zunächst sollen in diesem Abschnitt die binären Variablen behandelt werden.

Für zwei Objekte $o_i, o_j \,\epsilon\, O$ mit jeweils m binären Variablen ergeben sich die in Tabelle 2.2 aufgezeigten Auftretungshäufigkeiten der - pro binärer Variable - vier möglichen Kombinationen der Variablenwerte.

$o_i \rightarrow$ $o_j \downarrow$	1	0	
1	a	c	a+c
0	b	d	b+d
	a+b	c+d	m=a+b+c+d

Tabelle 2.2: Kombinationsmöglichkeiten binärer Variablen und deren Auftretungshäufigkeit

Der Wert a der Tabelle 2.2 gibt an, wie viele 1-1-Übereinstimmungen es gibt, Wert b wie viele 1-0-Übereinstimmungen usw.

Zur Berechnung von Distanzen bzw. Ähnlichkeiten zweier Objekte existieren nun zahlreiche Funktionen, die alle auf den Häufigkeitswerten der Tabelle basieren. Bevor einige dieser Funktionen vorgestellt werden, wird an dieser Stelle noch eine wichtige Unterscheidung vorgenommen, die zu einer Zweiteilung der binären Variablen führt.

Auf der einen Seite gibt es die so genannten symmetrischen Variablen. Als Beispiel eignet sich das Geschlecht des Menschen, mit „männlich“ und „weiblich“ als mögliche Zustände. Hier stellt sich die Frage, welche der beiden Zustände man mit 1 kodiert und welchen mit 0,

da keiner der beiden Zustände eine höhere Bedeutung hat bzw. gegenüber dem anderen hervorzuheben ist. Übertragen auf die Bewertung der Distanz zweier Objekte bedeutet das, dass eine 1-1-Übereinstimmung (a) und eine 0-0-Übereinstimmung (d) gleich zu gewichten sind und lediglich die Fälle b und c als Unterschied zwischen den Objekten zu werten sind. Dies wird in den Funktionen zur Berechnung der Distanz bzw. Ähnlichkeiten für symmetrische Variablen berücksichtigt. Tabelle 2.3 zeigt die bekanntesten Distanz- und Ähnlichkeitsfunktionen für symmetrische Variablen.

Name	$\mathbf{s_{ij}}$	$\mathbf{d_{ij}}$
Simple matching coefficient	$\frac{a+d}{a+b+c+d}$	$\frac{b+c}{a+b+c+d}$
Rogers und Tanimoto	$\frac{a+d}{(a+d)+2(b+c)}$	$\frac{2(b+c)}{(a+d)+2(b+c)}$
Sokal und Sneath	$\frac{2(a+d)}{2(a+d)+(b+c)}$	$\frac{b+c}{2(a+d)+(b+c)}$

Tabelle 2.3: Ähnlichkeits- / Distanzfunktionen für symmetrische binäre Variablen (entnommen aus [KR90])

Charakteristisch für symmetrische Variablen ist, dass man die Belegung von 1 und 0 vertauschen könnte und das Ergebnis dabei gleich bleibt. Der am häufigst verwendete *Simple matching coefficient* gibt den prozentualen Anteil der Übereinstimmungen ($a+d$) an der Gesamtmenge der symmetrischen Variablen als Ähnlichkeitswert (s_{ij}) oder den prozentualen Anteil der Nicht-Übereinstimmung ($b + c$) als Distanzwert ((s_{ij})) an. *Rogers und Tanimoto* gewichten die Nicht-Übereinstimmungen doppelt, während *Sokal und Sneath* die Übereinstimmungen doppelt gewichten.

Ganz anders zu bewerten sind asymmetrische Variablen, bei denen zwischen einer 1-1-Übereinstimmung („positive Übereinstimmung“) und einer 0-0-Übreinstimmung („negative Übereinstimmung“) zu unterscheiden ist. Man stelle sich in der Medizin die Untersuchung einer Gruppe von Personen vor, denen mehrere binäre Variablen zugeordnet wurden, die jeweils indizieren, ob die Person an einer bestimmten (seltenen) Krankheit leidet oder nicht. Würde man zur Berechnung der Ähnlichkeiten eine Funktion aus Tabelle 2.3 heranziehen, würde sich eine sehr homogene Gruppe mit hohen Ähnlichkeiten ergeben, da viele negative Übereinstimmungen auftreten. In der Biologie könnte man an bestimmten Tierartengruppierungen interessiert sein und ein binäres Attribut „Säugetier“ mit ja = 1 / nein = 0 belegen. Während nun eine positive Übereinstimmung durchaus etwas über eine Gemeinsamkeit der beiden Tiere aussagt, kann dies bei einer negativen Übereinstimmung nicht garantiert werden. Tabelle 2.4 zeigt die bekanntesten Funktionen zur Berechnung von Distanzen- / Ähnlichkeiten für asymmetrische binäre Variablen. Dabei sollte der „seltene“ bzw. zutreffende Fall (z.B. Krankheit vorhanden bzw. Säugetier) mit 1 kodiert werden, da die Funktionen daraufhin abgestimmt sind.

Name	$\mathbf{s_{ij}}$	$\mathbf{d_{ij}}$
Jaccard coefficient	$\frac{a}{a+b+c}$	$\frac{b+c}{a+b+c}$
Dice und Sorensen	$\frac{2a}{2a+b+c}$	$\frac{b+c}{2a+b+c}$
Sokal und Sneath	$\frac{a}{a+2(b+c)}$	$\frac{2(b+c)}{a+2(b+c)}$

Tabelle 2.4: Ähnlichkeits- / Distanzfunktionen für asymmetrische binäre Variablen (entnommen aus [KR90])

Aus der Tabelle wird ersichtlich, dass die Funktionen negative Übereinstimmungen (d) nicht mehr berücksichtigen. Andere - hier nicht aufgelistete - Funktionen gewichten a stärker als d.

An dieser Stelle sei kurz angemerkt, dass binäre Variablen zur Berechnung von Distanzen / Ähnlichkeiten gelegentlich wie intervall-skalierte Variablen, z.B. durch Verwendung der *Euklidischen Distanz*, behandelt werden. Dies führt bei symmetrischen Variablen zu relativ vernünftigen Ergebnissen[2], bei asymmetrischen Variablen ist die Aussagekraft des Ergebnis jedoch anzuzweifeln[3].

Zur Distanz- bzw. Ähnlichkeitsberechnung von nominalen Variablen, die r mögliche Zustände mit $r > 2$ annehmen können, existieren verschiedene Vorgehensweisen. Eine Möglichkeit besteht darin, die Variable in r (asymmetrische) binäre Variablen zu zerlegen, wobei jede binäre Variable eine Kategorie der nominalen Variable repräsentiert, und die jeweils zutreffende Kategorie auf 1 gesetzt wird. Dies kann zu einem immensen Overhead führen, z.B. bei einer nominalen Variablen „Herkunftsland“, für die r sehr groß werden kann.

Am häufigsten wird zur Berechnung der Distanz bzw. Ähnlichkeit nominaler Variablen der *Simple Matching Coefficient* für nominale Variablen verwendet, der sich wie folgt berechnet:

$$s_{ij} = \frac{a}{m} \tag{2.14}$$

$$d_{ij} = \frac{m-a}{m} \tag{2.15}$$

Dabei steht a für die Anzahl der Variablen in denen o_i und o_j sich im gleichen Zustand befinden (Übereinstimmung), und m für die Gesamtanzahl nominaler Variablen. Diese Funktion ist angelehnt an den *Simple matching coefficient* aus Tabelle 2.3 und ebenso wie bei dieser Funktion kann auch hier bei Bedarf eine unterschiedliche Gewichtung der Übereinstimmungen (a) oder Nicht-Übereinstimmungen ($m-a$) vorgenommen werden. Zusätzlich existieren

[2] Sowohl für positive als auch negative Übereinstimmung ergibt sich eine Distanz von 0, also eine hohe Ähnlichkeit. Die Fälle b und c fließen jedoch nur mit geringen Distanzwerten in die Rechnung mit ein.

[3] Negative und positive Übereinstimmungen werden gleich behandelt.

Ansätze, die Übereinstimmungen in Abhängigkeit der Anzahl r möglicher Zustände einer Variable stärker oder schwächer gewichten.

2.2.3 Ordinale Variablen

Ordinale Variablen unterscheiden sich von nominalen Variablen dadurch, dass die r Kategorien der Variable geordnet sind, also Aussagen wie $x_i < x_j$ und $x_i > x_j$ möglich sind. Beispiel sind Schulnoten von 1 bis 6, wobei zwei Objekte mit einer Belegung von 1 und 5 eine höhere Distanz aufweisen sollten als zwei Objekte mit einer Belegung von 2 und 3. Ausschlaggebend sind hier die Anzahl der Kategorien die zwischen zwei Werten liegen, die verglichen werden.

Auch hier besteht die Möglichkeit, eine ordinale Variable mit r Kategorien auf r-1 binäre Variablen abzubilden. Nimmt das betreffende Objekt die s-te Position der ordinalen Variable ein, setzt man die ersten $s-1$ binären Variablen auf 1 und die restlichen $r-s$ auf 0. Auf die binären Variablen wendet man die Funktionen aus Tabelle 2.3 an.

Weitaus häufiger werden ordinale Variablen jedoch wie quantitative Variablen behandelt und die Formeln 2.9 bzw. 2.10 der *Euklidischen* bzw. *Manhattan Distanz* verwendet. Dies liegt insofern nahe, dass je weiter zwei Kategorien voneinander entfernt liegen desto höher sollte die Distanz der Objekte sein. Es gilt jedoch zu beachten, dass verschiedene ordinale Variablen eines Objektes eine unterschiedliche Anzahl r an Kategorien aufweisen können und diese Variablen, um ungewollte unterschiedliche Gewichtung zu vermeiden, normiert werden sollten. Über die Formel

$$z_{if} = \frac{k_{if} - 1}{r_f - 1} \tag{2.16}$$

wird jede Kategorie k der ordinalen Variable f des Objektes i gleichmäßig auf einen Wert innerhalb des Intervall $[0, 1]$ abgebildet.

2.2.4 Gemischte Variablen

Bisher wurde vorausgesetzt, dass die Menge der Variablen eines Objektes aus einem einheitlichen Variablentyp besteht. In der Praxis wird man jedoch häufig auf den Fall gemischtskalierter Variablen treffen, d.h. hier taucht zur Beschreibung von Objekten eine Kombination der vorgestellten Variablentypen auf. Im Folgenden werden einige Vorgehensweisen aufgezeigt, um mit dieser Problematik umzugehen.

Ein Ansatz, der sich nur in den seltensten Fällen eignet, schlägt vor, die vorliegende Datenmatrix in ihre Variablentypen aufzusplitten und für jede Teilmatrix eine gesonderte Distanz- / Ähnlichkeitsberechnung und anschließende Clusteranalyse durchzuführen. Entstehen als Ergebnis ähnliche Clusterstrukturen, kann man sich mit dem Resultat zufrieden geben, ansonsten ist es nahezu unmöglich, die Ergebnisse zu kombinieren.

Ein zweiter Vorschlag verfolgt den Ansatz der Skalentransformation. Wie für einige Variablentypen bereits kurz erläutert (nominale, ordinale), besteht die Möglichkeit, diese auf binäre Variablen abzubilden. Intervall-skalierte Variablen können bspw. in ordinale Variablen überführt werden[4], indem man das Maximum und Minimum einer intervall-skalierten Variable bestimmt und das entstehende Intervall ($[Minimum, Maximum]$) in r Kategorien aufspaltet. Dies führt jedoch zu Informationsverlust! Das Vorgehen, alle Variablentypen auf das niedrigste Skalenniveau der vorkommenden Variablen zu transformieren, nennt man *Niveau-Regression*. Ebenso ist eine *Niveau-Progression*, also eine Transformation der niedrigen Skalen auf das höchste Skalenniveau, möglich.

Einige Autoren schlagen vor, alle Variablentypen einfach wie intervall-skalierte Variablen zu behandeln. Dies wurde für den Fall der binären Variablen bereits beschrieben und festgestellt, dass dieses Vorgehen bei symmetrischen Variablen zu akzeptablen Ergebnissen führt, asymmetrische Variablen allerdings wie symmetrische behandelt werden. Auch bei ordinalen Variablen ist dieses Vorgehen durchaus sinnvoll, da Kategorien, die sich weiter entfernt voneinander befinden zu einer höheren Distanz führen. Genau dies ist das Problem bei den nominalen Variablen. Hier kommt es zu einer Fehlinterpretation, da hier nicht nur die Fälle der Übereinstimmung oder Nicht-Übereinstimmung betrachtet werden, sondern sich bei Nicht-Übereinstimmung durch die Berechnung zweier kodierten Kategorien, die sich weiter entfernt voneinander befinden, eine höhere Distanz ergibt als bei näher zusammenliegenden Kategorien. Trotz der Unzulänglichkeiten findet dieser Ansatz breite Anwendung.

Aufgrund der Unzulänglichkeiten der bisher vorgeschlagenen Ansätze, wird in [SL77] vorgeschlagen, die Distanz jeder Variable für sich zu berechnen und die Ergebnisse in einer Formel als gewichteter Mittelwert der Einzeldistanzen zusammenzutragen. Somit ergibt sich für die Objekte o_i, o_j eine Gesamtdistanz von

$$d_{ij} = \frac{m_I}{m} d_{ij}^I + \frac{m_N}{m} d_{ij}^N + \frac{m_O}{m} d_{ij}^O \tag{2.17}$$

mit d^I, d^N und d^O gleich der jeweiligen Distanzfunktionen für intervall-skalierte, nominale und ordinale Variablen; m_I, m_N und m_O gleich der jeweiligen Anzahl intervall-skalierter, nominaler und ordinaler Variablen und m gleich der Gesamtanzahl der Variablen.

Ein weiterer Ansatz, der versucht alle Variablentypen, unter Berücksichtigung des jeweiligen Typs, mithilfe einer einheitlichen Formel (mit Fallunterscheidung) abzudecken, findet sich in [KR90].

2.2.5 Weitere Anmerkungen

An dieser Stelle soll der Leser kurz auf zwei Dinge hingewiesen werden, deren Nicht-Beachtung das Ergebnis einer Clusteranalyse stark verzerren kann.

[4] Eine Überführung in binäre und nominale Variablen ist - unter höherem Informationsverlust - ebenso möglich.

In einer Datenmatrix kann es vorkommen, dass eventuell Werte einiger Objekte fehlen, z.B. weil diese nicht gemessen wurden oder gemessen werden konnten. Diese sog. „missing values“ sollten angemessen behandelt werden. Füllt man diese Lücken bspw. mit einer möglichst hohen Zahl als Zeichen des nicht Vorhandenseins, wird diese bei der Distanzberechnung mitberechnet und führt zu starken Verzerrungen. Es wird vorgeschlagen, missing values bei der Berechnung von Distanzen / Ähnlichkeiten nicht zu berücksichtigen oder durch einen Mittelwert der Werte der übrigen Objekte zu ersetzen (letzteres macht nur bei quantitativen Variablen Sinn).

Ebenso gilt es darauf zu achten, dass man Variablen, die keine zur Gruppierung relevanten Informationen beinhalten, vor der Distanz- / Ähnlichkeitsberechnung eliminiert. Solche Variablen nennt man „trash variables“. Möchte man in einer biologischen Untersuchung bspw. Personen aufgrund ihrer körperlichen Merkmale in Gruppen einteilen, spielt das Einkommen der Person keine Rolle. Diese Variable muss außen vor gelassen werden, da sie das Ergebnis in unvorhersehbarer Weise beeinflusst und den Beitrag der sinnvollen Variablen unterdrückt! Die Auswahl der geeigneten Variablen zur Beschreibung von Objekten ist eine nicht-triviale Aufgabe und verlangt Erfahrung und Wissen des Anwenders im jeweiligen Anwendungsbereich.

2.3 Clusterverfahren

Clusterverfahren existieren in unzähligen Ausprägungen. Eine Systematisierung von Clusterverfahren lässt sich nach verschiedenen Kriterien vornehmen. In [SL77] werden unter anderem eine Einteilung hinsichtlich des Gruppierungsresultats und des Gruppierungsprozesses als Kriterien angegeben.

Eine Systematisierung anhand des Gruppierungsresultats unterteilt Clusterverfahren nach der Struktur der Cluster, die sie produzieren. So wird auf der ersten Stufe eine Trennung in hierarchische und nicht-hierarchische Verfahren vollzogen, während weiterhin eine Differenzierung danach stattfindet, ob sich die Cluster überlappen oder elementfremd sind (nichtdisjunkte und disjunkte Cluster).

Eine Systematisierung nach dem Gruppierungsprozess betrachtet die Vorgehensweise der Verfahren, um Objekte in Cluster einzuteilen, wobei man bspw. eine Unterscheidung von iterativen und nicht-iterativen Verfahren vornehmen kann. Hier gibt es neben den gängigsten Clusterverfahren der partitionierenden und hierarchischen Verfahren moderne Ansätze wie dichtebasierte Verfahren.

Im Folgenden wird eine Systematisierung nach dem Gruppierungsprozess zugrunde gelegt und partitionierende (Kapitel 2.3.1), hierarchische (Kapitel 2.3.2) und dichtebasierte (Kapitel 2.3.3) Verfahren vorgestellt. In Kapitel 2.3.4 werden unbekanntere Verfahren kurz angesprochen.

Wie in Kapitel 2.1 bereits erwähnt, hängt die Wahl des zu verwendeten Clusterverfahrens vom jeweiligen Untersuchungsziel und Anwendungskontext ab. Alle Clusterverfahren wei-

sen spezifische Vor- und Nachteile auf, die in folgenden Ausführungen jeweils kurz angesprochen werden. Es ist Aufgabe des Anwenders, das / die für seine Untersuchungen geeignete(n) Clusterverfahren auszuwählen.

Neben den Vor- und Nachteilen der einzelnen Verfahrensgruppen werden jeweils der ihnen zugrunde liegende Algorithmus grob beschrieben und Möglichkeiten aufgezeigt, die Ergebnisse der Verfahren in angemessener Weise zu repräsentieren. Außerdem werden die bekanntesten Verfahren der einzelnen Verfahrensgruppen namentlich genannt.

2.3.1 Partitionierende Verfahren

Der Name für diese Verfahren leitet sich daraus her, dass partitionierende Verfahren die Objekte in k Cluster einteilen, wobei die Cluster zusammen die Anforderungen einer Partition erfüllen:

- Jede Gruppe (Teilmenge der Partition) muss mindestens ein Objekt beinhalten. (nichtleere Gruppen)
- Jedes Objekt muss zu genau einer Gruppe gehören. (disjunkte Gruppen)

Dies setzt zunächst voraus, dass $k \leq n$ gilt und weiterhin eine Anzahl k der zu erstellenden Cluster bekannt ist. Der Wert k wird vom Benutzer angegeben. Jedes Cluster wird durch sein Clusterzentrum (Centroid) beschrieben, dessen initiale „Lage“[5] sowie die Berechnung im Laufe des Algorithmus vom jeweiligen konkreten Verfahren abhängt. In einem iterativen Prozess wird jedes Objekt dem Clusterzentrum zugeordnet, dem es - im Sinne einer Distanz - am nächsten ist. Anschließend werden die Clusterzentren auf Basis der Objekte, die ihnen im vorhergehenden Schritt zugeordnet wurden, neu berechnet. Nun wird wieder für jedes Objekt geprüft, welchem Clusterzentrum es am nächsten ist. Diese Iteration wird solange fortgeführt, bis sich entweder keine Änderung mehr ergibt (z.B. die Lage der Clusterzentren ändert sich nicht) oder bis ein Abbruchkriterium erreicht wurde. Ein Abbruchkriterium kann z.B. eine maximale Anzahl von Iterationen sein, die durchgeführt werden sollen.

Die Objekte können also im Laufe des Verfahrens verschiedenen Clustern angehören, wobei in einem iterativen Prozess versucht wird, eine optimale Einteilung der Objekte in k Cluster vorzunehmen. Der beschriebene Algorithmus ist allen partitionierenden Clusterverfahren gemein. Die Realisierung der Art der Clusterzentren, der initialen Lage der Clusterzentren, der Berechnung der Distanz zwischen Objekten und Clusterzentren, der Neuberechnung der Clusterzentren und der Abbruchbedingung sind jedoch abhängig vom jeweiligen konkreten Verfahren, teilweise sogar von der konkreten Implementierung eines Verfahrens. So bestehen z.B. zur Festlegung der initialen Lage der k-Clusterzentren unter anderem folgende Möglichkeiten:

- Zufällige Auswahl von k Clusterzentren.

[5] Punkt im m-dimensionalen Raum mit m gleich der Anzahl der Variablen der Objekte.

- Die ersten k Elemente der Datenmatrix werden als Clusterzentren verwendet.
- Der Benutzer spezifiziert k Clusterzentren.
- Die k Objekte mit der größten Distanz zueinander bilden die initialen Clusterzentren.
- ...

Als bekanntestes unter den partitionierenden Clusterverfahren soll der *K-Means* Algorithmus vorgestellt werden. Abbildung 2.4 zeigt den Algorithmus als UML-Aktivitätsdiagramm.

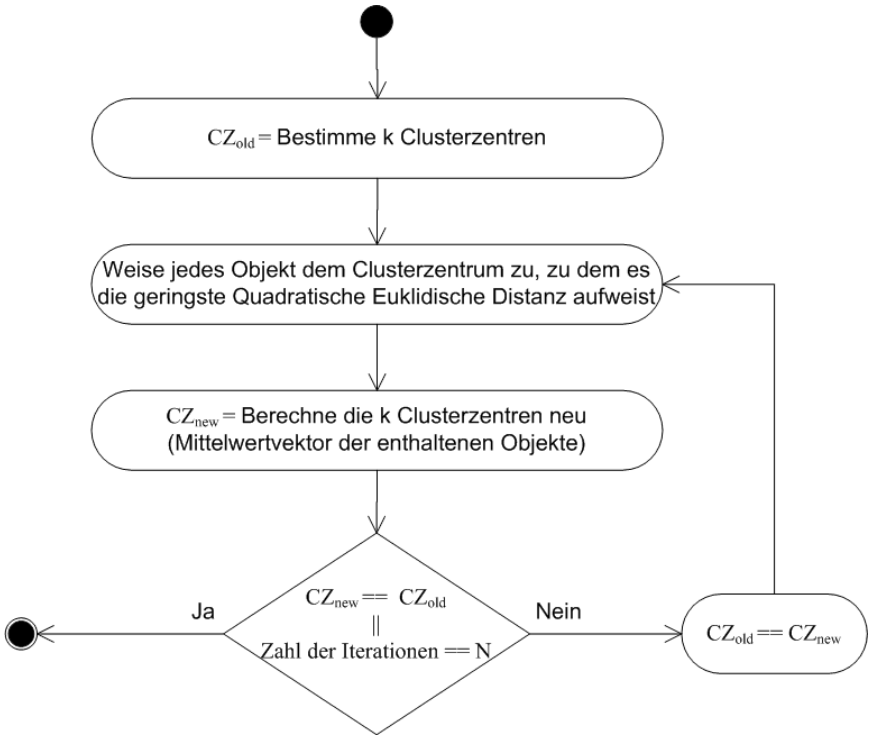


Abbildung 2.4: K-Means-Algorithmus als UML-Aktivitätsdiagramm

Die Clusterzentren werden als Punkte im m-dimensionalen Raum gewählt, repräsentieren also nicht zwangsweise ein Objekt der Datenmatrix. Die Bestimmung der initialen Lage ist durch den *K-Means*-Algorithmus nicht festgelegt. Die Clusterzentren werden als Mittelwertvektor der Dimension m, als arithmetisches Mittel der jeweiligen Variable der im Cluster enthaltenen Objekte, neu berechnet. Zur Berechnung der Distanzen zwischen den Objekten und Clusterzentren wird die *Quadratische Euklidische Distanz* herangezogen (siehe Formel 2.11, einige *K-Means*-Implementationen erlauben nur quantitative Variablen). Die Iteration wird abgebrochen, wenn sich entweder kein Clusterzentrum verändert hat oder eine maximale Anzahl an Iterationen, die vorher vom Benutzer festgelegt wurde, erreicht wurde.

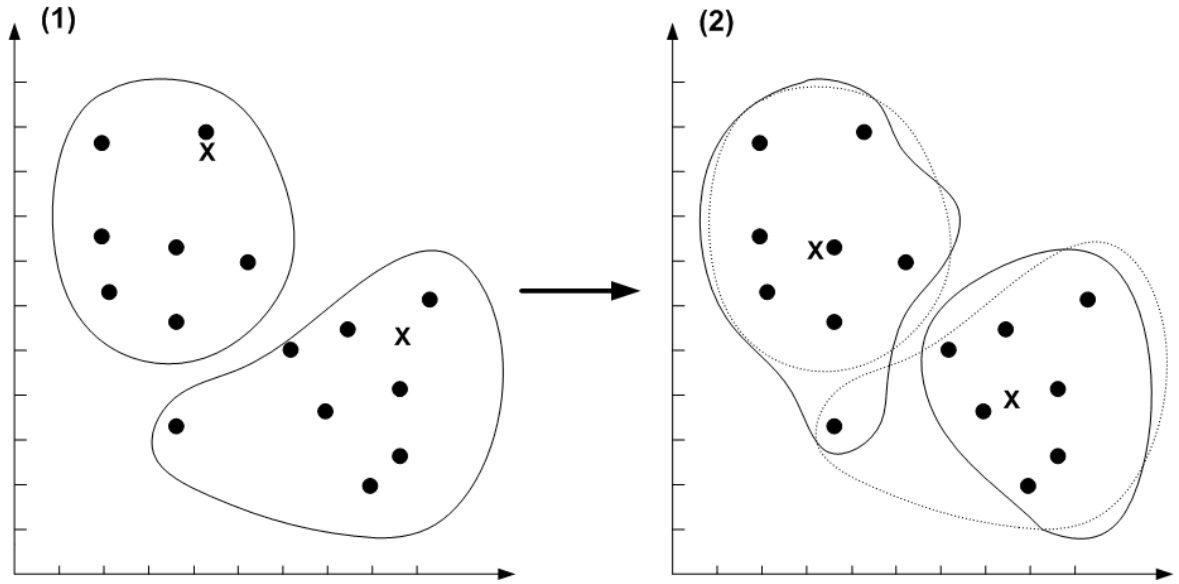


Abbildung 2.5: Erster Iterationsschritt des K-Means-Algorithmus (Beispiel)

Abbildung 2.5 zeigt ein Beispiel der ersten zwei Iterationsschritte des Algorithmus für $m = 2$ und $k = 2$. In Iterationschritt (1) wurden zufällig zwei Clusterzentren (dargestellt als **x**) bestimmt und jedes Objekt dem nächsten Clusterzentrum zugeordnet. In Iterationsschritt (2) wurden die Clusterzentren auf Basis ihrer zugeordneten Objekte als Mittelwertvektor neu berechnet, und jedes Objekt wiederum dem Clusterzentrum zugeordnet, dem es am nächsten ist. Die dünne, gestrichelte Linie im rechten Koordinatensystem zeigt die Clusterausdehnungen aus Iterationsschritt (1). Hier wurde also ein Objekt einem anderen Cluster zugeordnet.

Als Ergebnis partitionierender Verfahren erhält man eine Aufteilung von n Objekten in k Clustern, wobei jedem Objekt eindeutig ein Cluster zugewiesen wurde. Um die Güte der entstehenden Gruppierung besser beurteilen zu können, werden oftmals zusätzlich der durchschnittliche Abstand der Objekte zu ihrem Clusterzentrum, die Mittelwerte der einzelnen Variablen der Objekte eines Cluster (im Falle von *K-Means* also das Clusterzentrum) und die Varianz innerhalb der Cluster angegeben.

Weitere bekannte partitionierende Verfahren sind *K-Medoids* und *PAM* ([KR90], S.38ff).

K-Medoids unterscheidet sich von *K-Means* insofern, dass k Objekte der Datenmatrix als Clusterzentren, Medoid genannt, fungieren, wobei das zentralste Objekt eines Clusters das Clusterzentrum bildet. Dies minimiert den Einfluss von Ausreißerobjekten (siehe folgender Abschnitt). *PAM* (Partitioning Around Medoids) optimiert zusätzlich die Neuberechnung der Medoiden in jedem Iterationsschritt.

Ein Nachteil aller partitionierenden Verfahren ist, dass die Anzahl der zu generierenden Cluster k im Voraus angegeben werden muss, was eine gewisse Kenntnis der Daten voraussetzt. Da nicht alle Werte von k zu einer „natürlichen“ Gruppierung führen, sollten mehrere

Durchläufe mit unterschiedlichen Werten für k durchgeführt werden[6] oder zuvor durch eine hierarchische Clusteranalyse (siehe Kapitel 2.3.2) eine geeignete Anzahl von Clustern bestimmt werden. Ein weiterer Nachteil besteht darin, das so genannte Ausreißerobjekte oder Daten, deren Objekte eine starke räumliche Verteilung aufweisen, die Berechnung zu stark beeinflussen. Außerdem ist das Ergebnis und auch die Effizienz des Algorithmus stark abhängig von der Wahl der initialen Clusterzentren. Sind diese schon gut gewählt, sind nur wenige Iterationen notwendig, um eine gute Verteilung der n Objekte in k Cluster zu berechnen. Insgesamt liefern partitionierende Verfahren jedoch gute Ergebnisse, da sie die Gruppierungen der Objekte in k Cluster iterativ optimieren.

2.3.2 Hierarchische Verfahren

Bei hierarchischen Verfahren muss, im Gegensatz zu partitionierenden Verfahren, die Anzahl k der gewünschten Cluster nicht angegeben werden. Vielmehr erstellen hierarchische Verfahren in einem Durchlauf Ergebnisse für alle möglichen Werte von k für $1 <= k <= n$ mit n gleich der Anzahl der Objekte. Dabei befindet sich immer mindestens ein Objekt in jedem Cluster. Es wird sequentiell eine Hierarchie von Clustern erstellt, wobei Cluster auf verschiedenen Hierarchieebenen entweder elementfremd sind oder das Cluster der niedrigeren Ebene im Cluster der höheren Ebene enthalten ist. Die entstehende Clusterhierarchie kann als voller binärer Baum[7] dargestellt werden. Abbildung 2.6 zeigt einen solchen beispielhaften Baum mit $n = 5$.

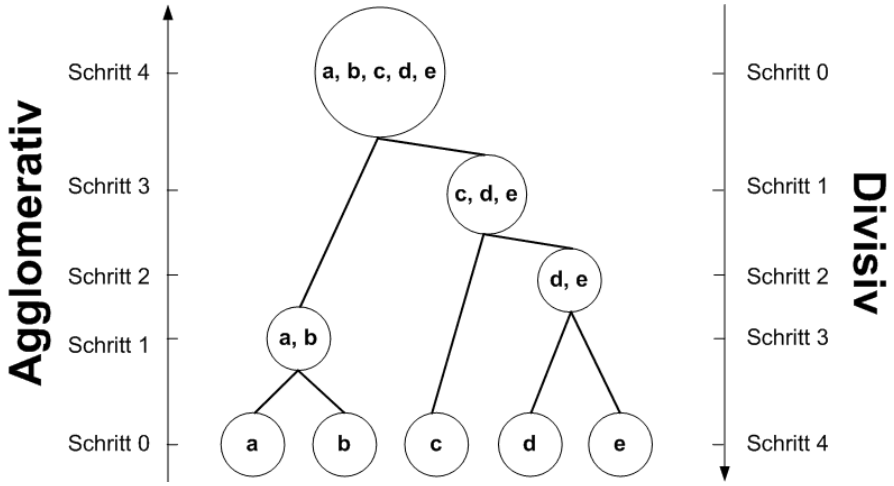


Abbildung 2.6: Hierarchische Clusterstruktur für $n = 5$

Je nach Verfahren wird diese Hierarchie entweder durch schrittweise Vereinigung (Fusion) von je zwei Clustern aufgebaut (agglomerativ), wobei sich hier zu Beginn jedes Objekt

[6] Dies kann von der jeweiligen Software eventuell automatisiert und die optimale Anzahl von Clustern über bestimmte Kriterien, wie z.B. der innere Zusammenhalt der Objekte eines Cluster oder der durchschnittliche Abstand der Objekte zu ihren Clusterzentren in Abhängigkeit von k, festgestellt werden.

[7] Jeder Knoten des Baumes hat entweder kein Kind oder genau zwei Kinder

in einem eigenen Cluster befindet ($k = n$), oder durch schrittweise Teilung eines Cluster in jeweils zwei Cluster (divisiv), ausgehend von einem Cluster ($k = 1$), das alle Objekte beinhaltet. Dies wird jeweils solange ausgeführt, bis eine Abbruchbedingung eintritt. Die Abbruchbedingung lautet beim agglomerativen Verfahren, dass alle Objekte sich in einem Cluster befinden ($k = 1$) und beim divisiven Verfahren, dass jedes Objekt sich in einem eigenen Cluster befindet ($k = n$).

Dabei werden im Falle des agglomerativen Verfahrens, wo sich zu Beginn jeweils nur ein Objekt in jedem Cluster befindet, die Cluster fusioniert, die die geringste Distanz zueinander aufweisen (in Abbildung 2.6 die Cluster {a} und {b} in Schritt 1). Dies verlangt die Berechnung der kompletten Distanzmatrix im Voraus. Da die Distanzmatrix jedoch nur etwas über die Distanz einzelner Objekte aussagt, im Verlauf des Algorithmus aber auch die Distanzen von Clustern mit mehreren Objekten verglichen werden müssen, werden Kriterien benötigt, die dies ermöglichen. Werden z.B. in einem ersten Schritt zwei Cluster mit jeweils einem Objekt zu einem neuen Cluster fusioniert, muss die Distanz des neuen Cluster zu allen übrigen Clustern neu berechnet (in Abbildung 2.6 bspw. Distanz von {a, b} zu {c}, {d} und {e}) und die Distanzmatrix dementsprechend aktualisiert werden. Kriterien, die eine Berechnung der Distanz von Clustern ermöglichen, heißen *Linkage-Kriterien* und werden im Folgenden vorgestellt.

Abbildung 2.7 zeigt den Algorithmus des agglomerativen Verfahrens, der im Vergleich zum divisiven Verfahren wesentlich häufiger eingesetzt wird, als UML-Aktivitätsdiagramm.

Die angesprochenen *Linkage-Kriterien* ermöglichen also die Berechnung der Distanz zweier Cluster, von denen mindestens eins mehr als ein Objekt enthält. In der folgenden Auflistung werden die bekanntesten *Linkage-Kriterien* mit ihren jeweiligen Eigenschaften vorgestellt, wobei jeweils eine Formel zur Berechnung der Distanz d_{AB} zwischen einem neu gebildeten Cluster A und einem existierenden Cluster B angegeben wird:

- *Single-Linkage*
  Die Distanz ergibt sich aus der geringsten Distanz zweier Objekte der Cluster A und B, weshalb dieses Verfahren auch *Nearest Neighbor* genannt wird. Formal geschrieben:

  $$d_{AB} = Min(d_{ij}) \tag{2.18}$$

  mit $i \epsilon A$ und $j \epsilon B$. Das Kriterium birgt die inhärente Eigenschaft, dass es zu „Aneinanderreihungseffekten“ (siehe [SL77]) führt, d.h. Cluster bildet, deren Objekte (als Punkte im m-dimensionalen Raum) gewisse Strukturen bilden, wie z.B. eine Linie oder ein Kreis (dies kann durchaus wünschenswert sein). Es werden somit auch Cluster vereint, die über ein so genanntes „Brücken-Objekt“ verfügen, d.h. ein Ausreißerobjekt eines Cluster kann eine Vereinigung bewirken. Dies führt dazu, dass früh große Cluster gebildet werden.

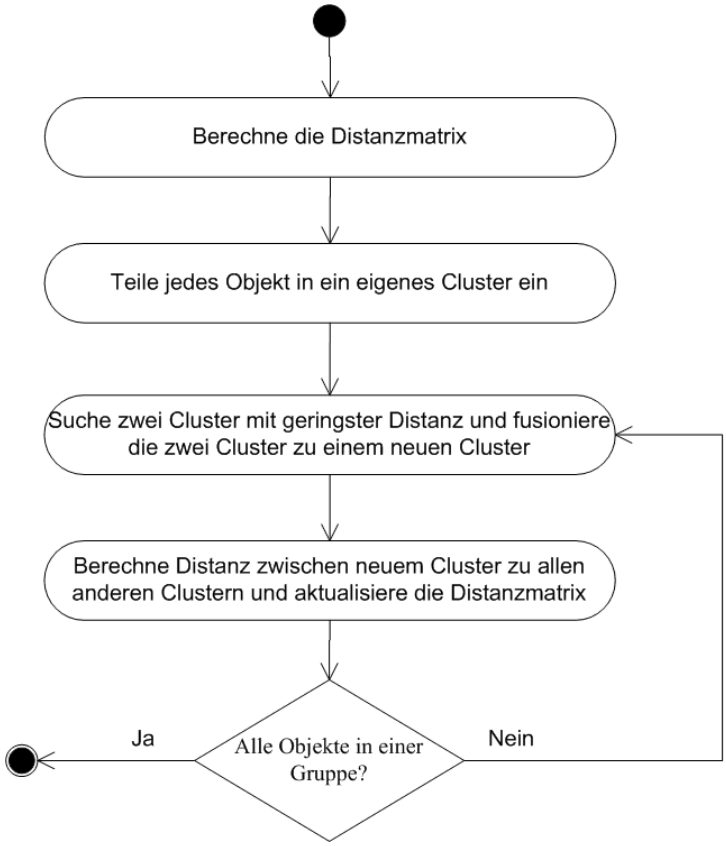


Abbildung 2.7: Algorithmus des agglomerativen Clusterverfahrens als UML-Aktivitätsdiagramm

- *Complete-Linkage*
  Die Distanz ergibt sich aus der größten Distanz zweier Objekte der Cluster A und B, weshalb dieses Verfahren auch *Furthest Neighbor* genannt wird. Formal geschrieben:

$$d_{AB} = Max(d_{ij}) \tag{2.19}$$

  mit $i \epsilon A$ und $j \epsilon B$. Das Kriterium birgt die inhärente Eigenschaft, dass früh viele kleine Cluster gebildet werden, da von einer strengen Vorstellung der Homogenität innerhalb der Cluster ausgegangen wird[8].

- *Average-Linkage*
  Die Distanz ergibt sich als Mittelwert aller Distanzen zwischen jedem Objekt aus Cluster A und jedem Objekt aus Cluster B, weshalb dieses Verfahren auch als *Mittelwert-Verfahren* bezeichnet wird. Formal geschrieben:

[8] Die Distanz der am weitesten entfernten Objekte muss gering sein, damit die Gesamtdistanz der zwei Cluster ebenfalls gering ist.

$$d_{AB} = \frac{1}{Na * Nb} \sum_{i=1}^{Na} \sum_{j=1}^{Nb} d_{ij} \tag{2.20}$$

mit $i \epsilon A$, $j \epsilon B$, $dist_{ij}$ der Distanz zwischen i und j und Na gleich der Anzahl der Objekte in Cluster A (analog Nb). Dieses Verfahren bildet einen Kompromiss zwischen den Extremen *Single-Linkage* und *Complete-Linkage* und führt zu guten Ergebnissen. Es sollte besonders dann angewandt werden, wenn über die Eignung der beiden extremen Ansätze für die vorliegenden Daten keine Aussage gemacht werden kann. Es existieren einige Variationen dieses Verfahrens, die in [Gor99] nachgelesen werden können.

Die aufgelisteten Verfahren stellen die bekanntesten *Linkage-Kriterien* dar. Es existieren noch eine Vielzahl zusätzlicher Kriterien (siehe [SL77], S.75ff), von denen einige, wie z.B. das *Wards Verfahren*, nicht die Distanz als Kriterium heranziehen, sondern das Heterogenitätsmaß innerhalb der Cluster und dessen Zuwachs durch Fusion mit anderen Clustern.

Zur Repräsentation der Ergebnisse hierarchischer Clusterverfahren, und damit des entstehenden vollen binären Baumes, hat sich das Dendrogramm durchgesetzt. Abbildung 2.8 zeigt ein Dendrogramm einer beispielhaften hierarchischen Clusteranalyse mit sechs Objekten mit hier nicht näher spezifizierten Variablen. Ein Dendrogramm lässt die schrittweise Teilung bzw. Vereinigung und die resultierende hierarchische Anordnung der Cluster erkennen. Dabei zeigt die X-Achse, bei horizontaler Ausrichtung, die relative Distanz (Abstandsrelation) der Cluster zueinander und damit auch die Reihenfolge der Vereinigung. Steigt diese relative Distanz an einigen Stellen des Dendrogramm sprunghaft an, weist dies auf eine natürliche Gruppierung der Objekte hin und lässt die Anzahl k der Cluster erkennen. In Abbildung 2.8 ergibt sich für $k = 2$ und $k = 4$ eine gute Gruppierung. Ohne Dendrogramm (oder ähnliche Visualisierungstechniken) wären die Ergebnisse hierarchischer Clusteranalysen nur schwer zu interpretieren und damit wenig aussagekräftig.

Beispiele konkreter Implementationen agglomerativer und divisiver hierarchischer Verfahren sind AGNES (AGglomerative NESting, [KR90], S.199) und DIANA (DIvisive ANAlysis, [KR90], S.253).

Ein immenser Vorteil hierarchischer Clusterverfahren ist die Tatsache, dass die Anzahl k der gewünschten Cluster im Voraus nicht festgelegt werden muss. Daher eignen sich diese Verfahren vor allem dann, wenn man keine Vorstellung bzgl. der möglichen bzw. optimalen Anzahl der Cluster hat. In diesem Fall kann man mit den Daten erste Experimente ausführen, um mögliche Gruppierungen im Dendrogramm zu erforschen und anschließend auf andere Clusterverfahren zurückgreifen, die eine Optimierung der Einteilung von n Objekten in k Cluster vollziehen. Weiterhin eignen sich diese Verfahren für Daten, die eine natürliche Hierarchie aufweisen, z.B. finden hierarchische Verfahren breite Anwendung in der Biologie zur Klassifizierung von Tieren und Pflanzen. Diese Vorteile werden jedoch dadurch erkauft, dass erstens keine Optimierung der Einteilung der Objekte in k Cluster stattfindet und zweitens einmal begangene Fehler, z.B. die Vereinigung zweier Cluster, nicht mehr

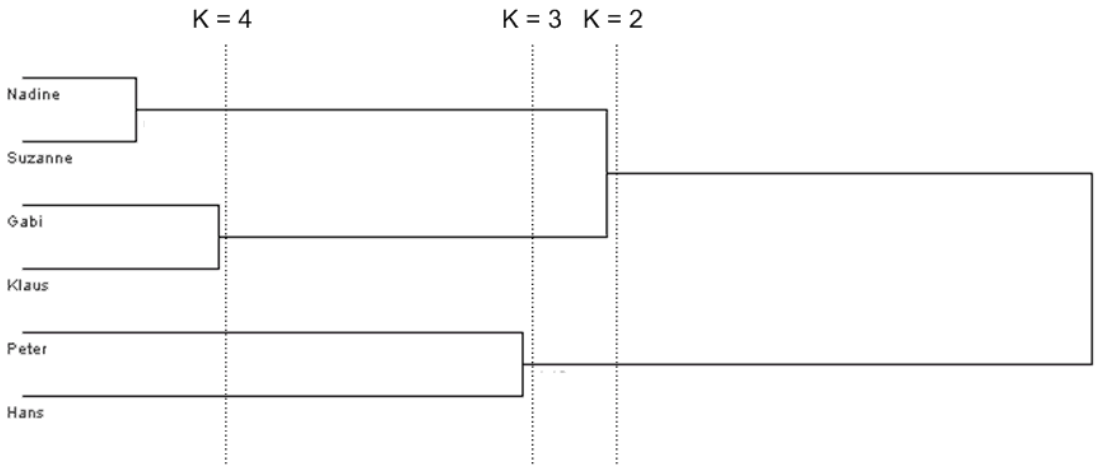


Abbildung 2.8: Dendrogramm einer hierarchischen Clusteranalyse

rückgängig gemacht werden können: „...it can never repair what was done in the previous step.“ ([KR90]). Die Neuberechnung der Distanz bei jeder Vereinigung zweier Cluster zu den übrigen Clustern führt, bei großer Anzahl von Objekten, zu einem hohen Rechenaufwand des Verfahrens.

2.3.3 Dichtebasierte Verfahren

Dichtebasierte Verfahren finden in der Praxis weitaus weniger Anwendung als die bisher vorgestellten Verfahren. Dennoch stoßen dichtebasierte Verfahren auf breites Interesse, da sie für Daten geeignet sind, für die die Anwendung partitionierender oder hierarchischer Verfahren keine optimale Lösung darstellt. Deshalb werden diese Verfahren hier kurz vorgestellt.

Dichtebasierte Verfahren erzeugen Cluster auf Basis der Dichte der Objekte zueinander, d.h. hier wachsen Cluster solange an, bis die Distanz von benachbarten Objekten einen gewissen Schwellwert überschreitet. Am besten verständlich wird das Vorgehen dichtebasierter Verfahren wenn man sich den Algorithmus ansieht. Dazu werden einige Definitionen vorgestellt, die jedem dichte-basierten Verfahren zugrunde liegen.

Zunächst werden zwei Parameter definiert, ein Abstand bzw. eine Distanz ε und eine minimale Anzahl $MinPoints$ an Objekten. Die Menge von Nachbarn (**Nachbarschaft**) $N(i)$ des Objektes $i \, \epsilon \, O$ ist definiert als

$$N(i) = \{j \epsilon O | d_{ij} \leq \varepsilon\} \tag{2.21}$$

mit d_{ij} gleich der Distanz zwischen Objekt i und j. Ein Objekt gilt als **Kernobjekt**, wenn

$$|N(i)| \geq MinPoints \tag{2.22}$$

gilt. Weiterhin gelten folgende Relationen für Objekte i und j:

- Ein Objekt j ist **direkt Dichte-erreichbar** vom Kernobjekt i, falls j in der Nachbarschaft von i ist, also wenn

  $j \epsilon N(i)$ (j ist in Nachbarschaft von i)
  $|N(i)| \geq MinPoints$ (i ist Kernobjekt)

  gilt.

- Ein Objekt j ist **Dichte-erreichbar** von Kernobjekt i, falls es eine Folge von Objekten zwischen i und j gibt, die über die direkte Dichte-Erreichbarkeit gekoppelt sind.

Abbildung 2.9 zeigt die Begrifflichkeiten an einem konkreten Beispiel mit ε gleich der Länge der eingezeichneten Pfeile und $MinPoints = 2$.

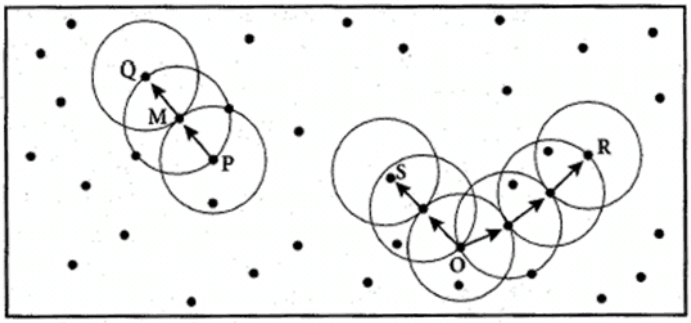


Abbildung 2.9: Dichte-Eigenschaften von Objekten

Im Beispiel aus Abbildung 2.9 haben die Objekte folgende Eigenschaften:

- Objekte M, P, O, R sind **Kernobjekte**. Objekte Q und S sind keine Kernobjekte (da $|N(Q)| < 2$, analog für S).

- Objekt Q ist **direkt Dichte-erreichbar** von M; M von P und P von M. Sonst besteht diese Relation zwischen keinen weiteren Objekten.

- Objekt Q ist **Dichte-erreichbar** von P; R und S von O; O von R (nicht O von S, da S kein Kernobjekt!). Sonst besteht diese Relation zwischen keinen weiteren Objekten.

Das *DBSCAN*-Clusterverfahren (Density Based Spatial Clustering of Applications with Noise, siehe [EKSX96]) ist das bekannteste dichtebasierte Verfahren. Der Algorithmus des Verfahrens ist, aufbauend auf den vorgenommenen Definitionen, leicht nachzuvollziehen und wird in Abbildung 2.10 als UML-Aktivitätsdiagramm dargestellt. Es wird über alle Objekte iteriert und nur die Kernobjekte betrachtet. Ausgehend von diesen werden alle Dichteerreichbaren Objekte gesucht, und aus der entstehenden Menge ein Cluster gebildet. Eine

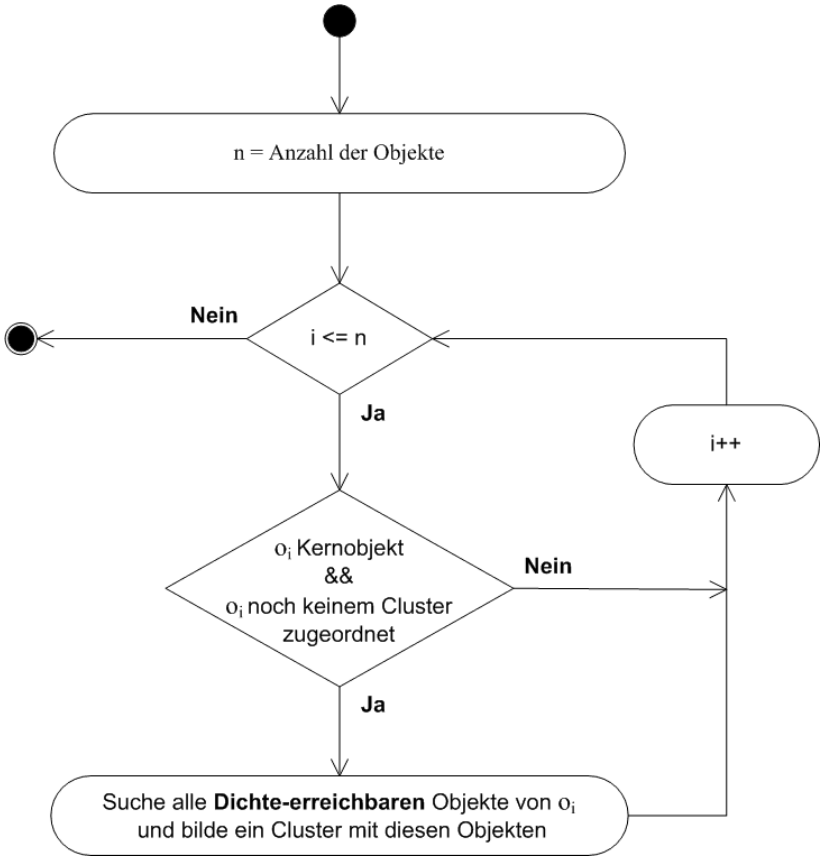


Abbildung 2.10: DBSCAN-Algorithmus als UML-Aktivitätsdiagramm

inhärente Eigenschaft des Algorithmus ist, dass einige Objekte - Ausreißerobjekte oder auch „noise“ genannt - eventuell keinem Cluster zugeordnet werden.

Ein weiteres dichtebasiertes Verfahren ist OPTICS (Ordering Points To Identify the Clustering Structure, siehe [ABKS99]). Die Innovation dieses Algorithmus besteht darin, dass keine wirklichen Cluster erstellt werden, sondern eine Ordnung der Objekte. Diese Ordnung basiert auf zwei Eigenschaften, die jedes Objekt besitzt und die vom Algorithmus berechnet werden, die Kerndistanz und die Erreichbarkeitsdistanz. Durch die Visualisierung der Werte in einem sog. Erreichbarkeitsgraphen werden Clusterstrukturen ersichtlich.

Ein Vorteil dichtebasierter Verfahren ist, dass die Anzahl k der Cluster im Voraus nicht bekannt sein muss, sondern diese auf Basis der räumlichen Verteilung der Objekte selbst bestimmt wird. Dichtebasierte Verfahren eignen sich damit für Daten mit beliebiger räumlicher Verteilung und somit auch für Daten mit Ausreißerobjekten. Zudem weist das Verfahren eine sehr gute Performanz auf, da es sich nicht um ein iteratives Verfahren handelt und die Gruppierung in nur einem Durchlauf stattfindet. Damit eigen sich dichtebasierte Verfahren auch für extrem große Datenmengen. Obwohl der Parameter k automatisch ermittelt wird, müssen mit ε und $MinPoints$ zwei Werte angegeben werden, die das Ergebnis stark

beeinflussen. Vor allem ist es für den Anwender nur schwer nachvollziehbar, wie sich die Änderungen auf das Ergebnis auswirken. Einer weiterer Nachteil ist darin zu sehen, dass bei Daten mit einer sehr hohen Dichte oftmals nur ein Cluster entsteht. Man muss sorgfältig prüfen, ob dichtebasierte Verfahren für die vorliegenden Daten geeignet sind.

2.3.4 Weitere Verfahren

An dieser Stelle sei der Vollständigkeit halber erwähnt, dass noch weitere Clusterverfahren existieren, die sich nach dem Gruppierungsprozess systematisieren lassen. Es handelt sich dabei um moderne Ansätze, die bisher noch keine breite Anwendung erfahren haben. Zu nennen sind die gitterbasierten und die modellbasierten Clusterverfahren. Der interessierte Leser findet in [HK01] eine Einführung in diese Verfahren.

Ebenfalls erwähnenswert sind Fuzzy-Verfahren, die keine eindeutige Zuordnung von Objekten zu Clustern vollziehen (können nicht-disjunkte Cluster erzeugen).

2.4 Softwareclustering

Wie in Kapitel 2 bereits dargelegt, kommt die Clusteranalyse in verschiedensten Anwendungsbereichen zum Einsatz. Die 1981 veröffentlichte Arbeit [BE81] von Belady und Evangelisti prägte den Begriff **Softwareclustering** und zählt zu den ersten Veröffentlichungen, die Verfahren der Clusteranalyse für die Anwendung auf Softwaresysteme vorschlagen.

Im Laufe der Jahre beschäftigten sich weitere Wissenschaftler mit diesem Themenbereich und spätestens seit dem wachsenden Interesse am Reverse Engineering Mitte der 1990er Jahre hat sich das Softwareclustering als eigenständige Disziplin etabliert.

Folgende Ausführungen zeigen zunächst bisherige Forschungsarbeiten im Bereich des Softwareclustering und behandeln anschließend - von der Theorie der „klassischen“ Clusteranalyse abweichende - Besonderheiten im Bereich des Softwareclustering. Abschließend wird auf einige offene Aspekte eingegangen, die Anreize und Vorschläge für weitere Forschungen im Gebiet des Softwareclustering darstellen.

2.4.1 Bisherige Arbeiten

Belady und Evangelisti gehören zu den Pionieren in der Anwendung von Clusteranalysen zur automatischen Gruppierung von Komponenten eines Softwaresystems (siehe [BE81]). Sie entwickelten eine Clusteranalyse-Software, die speziell auf die Analyse eines Softwaresystems zugeschnitten war. Die Gruppierung der Objekte fand jedoch nicht auf Basis von Quellcode sondern mithilfe der Dokumentation des Softwaresystems statt.

Hutchens und Basili [HB85] verwendeten den Datenzugriff, d.h. den Zugriff einer Prozedur auf eine Variable, als Kriterium zur Gruppierung von Prozeduren. Aus dem entstehenden hierarchischen Clusterbaum konnten mögliche Modularisierungen des Systems abge-

lesen werden. Die Ergebnisse der Clusteranalyse wurden mit Einschätzungen von System-Experten verglichen, was zu zufrieden stellenden Ergebnissen führte. Aus weiteren Experimenten gewannen Hutchens und Basili die Erkenntnis, dass die Reduktion der zugrunde liegenden Datentabelle auf die wesentlichen Objekte (Prozeduren) und Attribute (Variablenzugriffe) zu einer Verbesserung und einfacheren Interpretation der Ergebnisse führte.

Anquetil und Lethbridge [AL99] führten hierarchische Clusteranalysen auf mittelgroßen bis sehr großen Systemen (> 2 Millionen Loc) aus. Sie konzentrierten sich dabei auf die Variation von zwei Aspekten: Wie sollen die zu gruppierenden Objekte mit Attributen beschrieben werden? Welcher agglomerative hierarchische Clusteralgorithmus (also welches Linkage-Kriterium) soll verwendet werden? Die verwendeten Features (Attribute) zur Beschreibung der Objekte wurden in formale und nicht-formale Features unterteilt. Formale Features sind solche, die das Systemverhalten beeinflussen, wie bspw. Aufrufbeziehungen zwischen Methoden, während nicht-formale Features keinen direkten Einfluss auf das Systemverhalten haben. Als nicht-formale Features wurden die Namen von Variablen, Prozeduren, etc. verwendet. Ergebnis der Untersuchungen war, dass die Gruppierung nach nicht-formalen und formalen Features Ergebnisse gleicher Qualität hervorbringen. Hier ist jedoch anzumerken, dass dieser Ansatz stark von einer konsistenten Namensvergabe der Programmelemente von Seiten des Quellcode-Autors abhängig ist. Weiterhin stellten sich das Complete- und Average-Linkage als die geeignetsten Linkage-Kriterien heraus.

Lung et al. stellen in [Lun98] einen Ansatz zur Erkennung von Entwurfsmustern im Quellcode vor. An einer exemplarischen Architektur einer Softwarekomponente wird die Restrukturierung der Komponente unter Anwendung des Facade-Musters ([Gam97]) beschrieben.

Zusammenfassend ist festzuhalten, dass sich die große Mehrheit der Forschung im Bereich des Softwareclustering auf die Wiedergewinnung der Softwarearchitektur konzentriert. Dies liegt insofern nahe, dass mithilfe von Clusteranalysen eine Struktur innerhalb der zugrunde liegenden Daten und damit eine abstraktere Sicht auf diese gefunden werden kann, was mit der Wiedergewinnung der Softwarearchitektur als abstrakte Systemsicht auf den implementierten Quellcode einhergeht. Qualitätsmerkmale einer Softwarearchitektur sind ein hoher innerer Zusammenhalt der Komponenten und eine geringe Kopplung zwischen den Komponenten. Das Ergebnis einer Clusteranalyse wird als gut empfunden, wenn die Elemente innerhalb eines Cluster möglichst ähnlich und die Elemente aus verschiedenen Clustern möglichst unähnlich sind.

Zur Wiedergewinnung der Softwarearchitektur wurden unterschiedlichste Ansätze und Konfigurationen verfolgt, die je nach Zielsystem mehr oder weniger viel versprechende Ergebnisse lieferten. Die zahlreichen ermutigenden Ergebnisse und die Vielfältigkeit der Konfigurationsmöglichkeiten von Clusteranalysen bieten Anreiz für weitere Forschungen im Bereich des Softwareclustering, wobei sich diese nicht unbedingt nur auf die Wiedergewinnung der Softwarearchitektur beschränken müssen (siehe auch Abschnitt 2.4.3).

2.4.2 Besonderheiten des Softwareclustering

Die bisherigen Forschungsarbeiten im Bereich des Softwareclustering führten dazu, dass sich spezielle Charakteristika in diesem Teilgebiet der Clusteranalyse herauskristallisierten.

In diesem Abschnitt sollen lediglich die von der „klassischen Clusteranalyse“ abweichenden Chrakterisitika vorgestellt werden. Dazu gehören:

- die Verwendung einer von der „klassischen Clusteranalyse“ leicht abweichenden Terminologie.
- ein für das Softwareclustering typisches Vorgehen bei der Auswahl und Beschreibung der zu gruppierenden Daten.
- ein Verfahren zur Evaluation der Ergebnisse einer Clusteranalyse auf Softwareelementen.

Terminologie

Die Disziplin des Anwendens der Clusteranalyse auf Softwareelemente wird im allgemeinen als **Softwareclustering** bezeichnet [BE81].

Wiggerts gibt in seiner Arbeit [Wig97] einen Überblick über Techniken und Verfahren im Softwareclustering. Dabei schlägt er die Verwendung der Bezeichnung **Entitäten** für die zu gruppierenden Objekte und **Features** für die Attribute der Objekte vor. Die Bezeichnung Feature fand seit der Veröffentlichung des Artikels breite Anwendung in der Softwareclustering-Community, während für die zu gruppierenden Objekte weiterhin verschiedene Terminologien wie Elemente, Objekte, Entitäten, etc. verwendet werden.

Im Folgenden wird die Bezeichnung Objekte für anonyme bzw. abstrakte Objekte und im konkreten Fall die Bezeichnung Softwareelemente verwendet. Handelt es sich um Softwareelemente, wird der Begriff Feature für die Beschreibung der Eigenschaften der Softwareelemente verwendet.

Auswahl und Beschreibung der Daten

Bei der Auswahl und Beschreibung der Daten geht es darum, den Softwareelementen Features zuzuordnen, auf deren Basis eine - im Sinne des Ziels der Clusteranalyse - geeignete Bestimmung der Ähnlichkeit bzw. Distanz der Softwareelemente durchgeführt werden kann.

Das Ziel des Softwareclustering ist oftmals die Gewinnung der Architektur einer Software zur (Re-)Modularisierung derselben. Qualitätsmerkmale einer Softwarearchitektur sind ein hoher innerer Zusammenhalt der Softwareelemente, der durch eine enge Beziehung zwischen den Softwareelementen zum Ausdruck gebracht wird. Zwei Softwareelemente sollten also eine hohe Ähnlichkeit aufweisen, wenn sie viele gemeinsame Beziehungen zu anderen Softwareelementen haben. Abbildung 2.11 verdeutlicht dies an einem Beispiel.

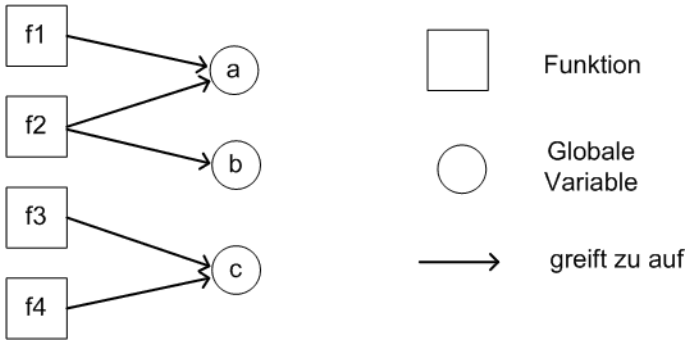


Abbildung 2.11: Beziehungen zwischen Funktionen und globalen Variablen

Die Abbildung zeigt die Beziehung „greift zu auf“ zwischen den Funktionen *f1* bis *f4* und den globalen Variablen *a*, *b* und *c* und legt eine Einteilung der Funktionen *f1* und *f2* bzw. *f3* und *f4* in ein jeweils eigenes Modul nahe. Weitere Beispiele für Softwareelemente sind Dateien, Module, Klassen, Prozeduren und Variablen. Zwischen diesen Softwareelementen können verschiedene Beziehungen existieren, wie z.B. „ruft auf“, „schreibt in“, „inkludiert“ oder „benutzt“.

Die Softwareelemente und deren Beziehungen zueinander möchte man gerne auf eine Datentabelle, die als Input einer Clusteranalyse dient, abbilden. Dafür stellt man die Softwareelemente, z.B. Prozeduren und Variablen, in Zeile und Spalte gegenüber. Liegt eine solche Tabelle vor, spricht man statt von einer Objekt-Attribut-Struktur von einer Objekt-Objekt-Struktur. Es handelt sich hierbei eher um eine rein begriffliche Differenzierung, da man die Spaltenbeschreibungen durchaus auch als Features der Objekte ansehen kann. Trotzdem hat diese Differenzierung Auswirkung auf die Auswahl und Beschreibung der einer Clusteranalyse zugrunde liegenden Daten.

Tabelle 2.5 zeigt eine solche Objekt-Objekt-Struktur. Zwischen den Objekten der Datentabelle muss eine implizite Beziehung - wie z.B. „ruft auf“ oder „schreibt in“ - bestehen. Ein „x“ in einer Zelle der Tabelle gibt an, dass die implizierte Beziehung - hier „greift zu auf“ - besteht.

Prozedur ↓	Variable „counter“	Variable „data“
paint		
submit	x	
calculateResult	x	x
modifyResult	x	x
getData		x

Tabelle 2.5: Objekt-Objekt-Datentabelle

Solche Objekt-Objekt-Tabellen sind im Softwareclustering, im Gegensatz zur Clusteranalyse in anderen Anwendungsbereichen, häufig vorzufinden. Bei den vorliegenden Features handelt es sich immer um (symmetrisch oder asymmetrisch) binäre Features.

Evaluation der Ergebnisse

Um die Qualität eines Ergebnisses einer Clusteranalyse zu bewerten, existieren kaum anerkannte Verfahren. Im Softwareclustering haben sich als Maße für die Qualität (Güte) eines Cluster-Ergebnisses **Precision** (Präzision) und **Recall** (Wiedererkennung) etabliert. Diese Maße sind vor allem als Gütekriterien zur Beurteilung der Qualität von Suchergebnissen im Information-Retrieval bekannt.

Soll durch eine Clusteranalyse von Softwareelementen bspw. eine geeignete Strukturierung bzw. Modularisierung eines Softwaresystems bestimmt werden, wird eine durch Experten - z.B. die Softwarearchitekten bzw. Programmierer des Systems - erstellte Strukturierung als Vergleich herangezogen. Die von der Clusteranalyse erstellte Architektur wird als Test-System, die von den Experten erstellte Architektur als Experten-System bezeichnet.

Zur Berechnung der Werte Precision und Recall werden **intra pairs** definiert. Dies sind Paare von verschiedenen Softwareelementen im selben Cluster. Betrachtet man in Abbildung 2.12 bspw. das Cluster B_2 des Experten-Systems, bilden { (M4, M7), (M4, M8), (M7, M8) } die Menge der intra pairs. Die Anzahl $|B_2|$ der intra pairs des Cluster B_2 ist somit drei.

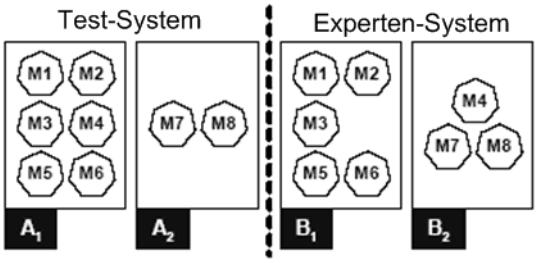


Abbildung 2.12: Ein Test-System und ein Experten-System

Vergleicht man nun ein Test-System A mit einem Experten-System B, dann ist die Precision P definiert als der Prozentsatz von intra pairs in A, die ebenfalls intra pairs in B sind, oder anders ausgedrückt:

$$P = \frac{|A \cap B|}{|A|}$$

Recall R ist definiert als der Prozentsatz von intra pairs in B, die ebenfalls intra pairs in A sind, oder anders ausgedrückt:

$$R = \frac{|A \cap B|}{|B|}$$

Wünschenswert sind hohe Werte für beide Maße. Bei einem Ergebnis einer hierarchischen Clusteranalyse ergibt sich zu Beginn, wenn jedes Softwareelement sich in einem eigenen

Cluster befindet, eine Precision von 100% und ein Recall von 0%. Mit steigender Zahl der Iterationsschritte sinkt die Precision und steigt der Recall. Befinden sich alle Elemente in einem Cluster, ergibt sich ein Recall von 100%.

Für die Gruppierung in Abbildung 2.12 ergibt sich eine Precision von 84,6% und ein Recall von 68,7%.

Dieses Verfahren eignet sich natürlich nur, um die Konfiguration einer Clusteranalyse für bestimmte Gruppierungszwecke zu testen. Möchte man ein Softwaresystem durch Softwareclustering re-modularisieren, liegt ein Experten-System in der Regel nicht vor.

2.4.3 Offene Aspekte

Obwohl Softwareclustering im Reverse Engineering sich als Disziplin etabliert hat, gibt es einige Aspekte, die noch nicht hinreichend erforscht wurden. Folgende Liste soll einige dieser Aspekte aufzeigen und auch als Anhaltspunkt für die im Rahmen dieser Arbeit zu entwickelnde Clusteranalyse-Umgebung dienen. Die in Anhang A spezifizierten Anforderungen an die Clusteranalyse-Umgebung wurden durch folgende Punkte beeinflusst:

- Auswahl der Daten

  Bisher wurde vorwiegend versucht, die Softwarchitektur von Legacy Systems auf Basis von Variablenzugriffen von Prozeduren oder Aufrufbeziehungen zwischen Prozeduren zu ermitteln. Dabei sollten auch andere Kriterien in Betracht gezogen werden, wie bspw. die Nutzung von Ressourcen. Möchte man eine Architektur in eine Model-View-Controller-Architektur restrukturieren, könnten Zugriffe auf Datenbanken ein geeigneter Ansatzpunkt sein.

- Untersuchung der Eignung verschiedener Clusterverfahren für verschiedene Softwaresysteme bzw. Gruppierungszwecke

  Im Softwareclustering wurde bisher meist, aufgrund der Offenlegung der hierarchischen Struktur der Daten, auf hierarchische Verfahren zurückgegriffen. Die Verwendung anderer Clusterverfahren wie partitionierende oder dichtebasierte Verfahren wurden nur marginal erforscht. Es ist wahrscheinlich, dass bestimmte Clusterverfahren für bestimmte Softwaresysteme und Gruppierungszwecke besser geeignet sind als andere. Eine Kategorisierung von Clusterverfahren nach den Softwaresystemen bzw. Gruppierungszwecken für die diese am Besten geeignet sind, kann zu viel versprechenden Ergebnissen führen und als Grundlage für die zukünftige Arbeit im Softwareclustering dienen.

- Nutzung der Distanz- bzw. Ähnlichkeitsberechnung zum Einbringen von zusätzlichem Anwendungswissen

  Ähnlich verhält es sich mit den Distanz- bzw. Ähnlichkeitsfunktionen. Da keine dem Autor bekannte Clusteranalyse-Software die Behandlung gemischt-skalierter Varia-

blen mit entsprechender Distanz- bzw. Ähnlichkeitsfunktion erlaubt, werden die Datentabellen meist so erstellt, dass sie über einheitliche Variablentypen verfügen. Dies stellt eine Einschränkung dar. Zusätzlich würde eine Gewichtung einzelner Variablen vor der Distanz- bzw. Ähnlichkeitsberechnung dem Anwender weitere Möglichkeiten eröffnen, das Ergebnis durch sein Anwendungswissen positiv zu beeinflussen.

- Überführung von Legacy Systems in neue Technologien

  Die Nutzung von Softwareclustering zur Migration von Legacy Systems auf Web-Applikation oder in eine OO-Architektur birgt großes Potenzial. Oftmals sollen Altsysteme in neue Technologien überführt werden. Vor allem bei der Überführung einer prozeduralen Software in eine OO-Architektur ist eine Modularisierung der Unterprogramme, wie z.B. Funktionen, notwendig. Möchte man ein Altsystem an das Web anbinden, kann das Softwareclustering zum Auffinden der Schnittstellen oder einer Überführung der Software in eine Client-Server-Architektur genutzt werden.

- Nutzung für das Forward Engineering

  Wie in [Lun98] angesprochen, kann Softwareclustering auch im Forward Engineering bspw. zum Entdecken von Entwurfsmustern eingesetzt werden. Untersuchungen zur Nutzung des Softwareclustering in weiteren Bereichen des Software-Lebenszyklus sind angebracht. Vorstellbar ist der Einsatz des Softwareclustering zur Unterstützung bei der Architekturerstellung auf Basis der Beziehung verschiedener Komponenten eines Systems.

3 Anforderungsdefinition

In diesem Kapitel werden die Anforderungen an die Clusteranalyse-Umgebung definiert. Die Anforderungen entstanden durch eine Anforderungserhebung in Form von Interviews mit den Betreuern dieser Arbeit. Diesen Interviews vorausgegangen, war die Studie der theoretischen Grundlagen der Clusteranalyse, sowie die Einarbeitung in den Stand der Technik im Softwareclustering.

Die Anforderungen werden in diesem Kapitel im Fließtext aufgeführt. Dabei wird sich nur auf die wichtigsten Anforderungen bezogen. Die detaillierte Anforderungsliste ist in Anhang A dieser Arbeit zu finden. Die Punkte der dort vorzufindenden Anforderungsliste überschneiden sich teilweise mit den hier im Fließtext aufgeführten Formulierungen.

Die Anforderungen sind in verschiedene Bereiche unterteilt. Zunächst finden sich die Anforderungen an die Architektur der Clusteranalyse-Umgebung. Anschließend werden die Anforderungen an das Reverse Engineering Tool zur Extraktion von Informationen aus Softwareartefakten definiert. Daraufhin finden sich die Anforderungen an die Clusteranalyse-Umgebung bzgl. der Clusteranalyse-Funktionalität. Es folgen die Definition der Anforderungen an die Benutzerschnittstelle (UI) und die Programmierschnittstelle (API) der Clusteranalyse-Umgebung. Abschließend finden sich allgemeine Anforderungen an die Clusteranalyse-Umgebung. Die hier vorgestellte Gliederung gilt sowohl für dieses Kapitel, als auch für die ausführliche Anforderungsliste in Anhang A dieser Arbeit.

Die im Folgenden und auch in Anhang A vorgestellten Anforderungen sind im Sinne einer idealisierten Systemvision zu sehen. Dies spiegelt sich auch durch den recht häufig auftretenden Begriff **soll** und die damit verbundene Verwendung des Konjunktiv-Modus wider, der zum Ausdruck bringt, dass die Umsetzung der Anforderung wünschenswert ist, jedoch nicht garantiert werden kann. Werden Begrifflichkeiten wie **ist** oder **wird** verwendet (Indikativ-Modus), handelt es sich um Muss-Anforderungen.

3.1 Anforderungen an die Architektur

Die zu erstellende Clusteranalyse-Umgebung wird nicht von Grund auf neu entwickelt, sondern macht sich die Funktionalität bestehender Software zunutze. Im Wesentlichen setzt sich die Clusteranalyse-Umgebung aus einem Reverse Engineering Tool zur Extraktion von Informationen aus Softwareartefakten und einer Clusteranalyse-Software zusammen.

Im Vorfeld dieser Arbeit wurde `GUPRO` als Reverse Engineering Tool festgelegt. Der Auswahl der zu verwendenden Clusteranalyse-Software soll ein strukturierter Software-Evaluations-Prozess zugrunde liegen.

Die Kopplung der beiden Komponenten soll möglichst lose sein und über eine kompakte Schnittstellte realisiert werden. Dabei ist ein Datenaustausch zwischen den Komponenten in beide Richtungen möglich. Der Benutzer kann die Clusteranalyse-Umgebung über eine einheitliche Benutzerschnittstelle bedienen.

3.2 Anforderungen an das Reverse Engineering Tool

Die Anforderungen an das Reverse Engineering Tool werden definiert, um die im Vorfeld der Arbeit festgelegte Software `GUPRO` gegen diese Anforderungen zu testen.

Das Reverse Engineering Tool ermöglicht eine Analyse von Softwareartefakten, wie z.B. Quellcode, Präprozessor- oder Datenbankdefinitionen. Diese Softwareartefakte sind Input des Reverse Engineering Tools. Die Art der aus den Softwareartefakten zu extrahierenden Informationen und deren Abstraktionsgrad können vom Benutzer bestimmt werden.

Die Ergebnisse der Analyse von Softwareartefakten können in einem offenen und programmiersprachenunabhängigen Format gespeichert und über eine Schnittstelle ausgelesen werden.

3.3 Anforderungen an die Clusteranalyse-Funktionalität

Die Clusteranalyse-Umgebung soll dem Benutzer größtmögliche Freiheit zur Konfiguration einer Clusteranalyse gewähren und so vielfältiges Experimentieren ermöglichen. Diese Anforderung prägt die weiteren Anforderungen dieses Abschnittes, da ihre Realisierung eine breite Palette an Konfigurationsmöglichkeiten zur Clusteranalyse voraussetzt.

Neben dem Vorhandensein eines Editors zum Betrachten und Editieren der eingelesenen Daten und einer Vielzahl von Clusterverfahren[1] ist eine freie Eingabe einer Funktion zur Distanz- bzw. Ähnlichkeitsberechnung möglich.

Die Ergebnisse von Clusterverfahren können in einer dem jeweiligen Verfahren angemessenen Visualisierungsmethode dargestellt werden. Außerdem können die Ergebnisse exportiert und jederzeit wieder in die Clusteranalyse-Umgebung geladen werden.

Die ausgewählte Software zur Clusteranalyse sollte bereits eine breite Palette an Konfigurationsmöglichkeiten anbieten. Kann die Clusteranalyse-Software geforderte Funktionalitäten nicht aufweisen, wird die Software um diese Funktionalitäten erweitert.

[1] Eine genaue Spezifikation der zu realisierenden Clusterverfahren findet sich in der Anforderungsliste in Anhang A.

3.4 Anforderungen an die Benutzerschnittstelle

Die Clusteranalyse-Umgebung verfügt über eine graphische Benutzeroberfläche (GUI). Die Bedienung der Benutzeroberfläche soll, trotz der vielfältigen Konfigurationsmöglichkeiten, übersichtlich und benutzerfreundlich sein.

Das Erstellen und die Konfiguration einer Clusteranalyse kann dabei in erkennbare und intuitiv erwartete Einzelschritte (Services) unterteilt werden. Dabei kann die Konfiguration einer Clusteranalyse gespeichert und zum wiederholten Ausführen jederzeit wieder geladen werden. Einzelschritte einer Clusteranalyse-Konfiguration sollen dabei ausgetauscht werden können.

3.5 Anforderungen an die Programmierschnittstelle

Die Funktionalität der Clusteranalyse-Umgebung kann vom Benutzer erweitert werden. Außerdem soll die Möglichkeit der Anbindung weiterer Reverse Engineering Tools zur Extraktion von Informationen aus Softwareartefakten offen gehalten werden.

Der Quellcode der Clusteranalyse-Umgebung muss zur Verfügung stehen und es wird eine einheitliche API realisiert. Eine Dokumentation der API steht zur Verfügung (siehe auch Abschnitt A.6).

3.6 Allgemeine Anforderungen

Da die Clusteranalyse-Umgebung über zahlreiche Konfigurationsmöglichkeiten verfügt, findet eine intensive Benutzerinteraktion statt. Dabei kann es zu Fehlkonfigurationen von Seiten des Benutzers kommen, auf die mit einer geeigneten **Fehlerbehandlung** reagiert werden soll.

Die Umsetzung der Erweiterungen der ausgewählten Clusteranalyse-Software werden entwicklungsbegleitend **getestet**. Als primäres Testverfahren werden Black-Box-Tests verwendet. Zusätzlich wird die fertig entwickelte Clusteranalyse-Umgebung in einem realen Anwendungsszenario getestet.

Da die Möglichkeit der Erweiterung der Clusteranalyse-Umgebung bestehen soll, ist eine **Dokumentation** der Architektur und der Schnittstelle der Umgebung vorhanden. Zusätzlich ist eine Dokumentation des gesamten Projektablaufes und der vorgenommenen Erweiterungen vorhanden.

4 Auswahl einer Software zur Clusteranalyse

Das Ziel der Arbeit besteht darin, eine Umgebung zur Clusteranalyse auf Softwareelementen zu erstellen. Wie in Kapitel 1 bereits erwähnt, soll die Clusteranalyse-Umgebung nicht von Grund auf neu entwickelt werden, sondern sich der Funktionalität existierender Software bzw. Bibliotheken zunutze machen (im Folgenden wird nur noch von Software als Sammelbegriff für Software, Umgebung, Bibliotheken, Softwarekomponenten gesprochen).

Dazu muss zunächst die Bandbreite der existierenden Software zur Clusteranalyse ermittelt und aus diesem Pool nach vorher festgelegten Kriterien die für unsere Zwecke leistungsfähigste Software bestimmt werden.

Die letztendlich gewählte Software hat großen Einfluss auf die zu erstellende Umgebung: Die Systemarchitektur, die funktionalen Fähigkeiten, die Wartbarkeit oder auch die Portabilität (um nur einige Aspekte zu nennen) können von der gewählten Software abhängen. Deshalb sollte der Auswahl der Software entsprechende Bedeutung zukommen und ein strukturiertes und objektivierbares Vorgehen zugrunde liegen.

Diesem Aspekt wird hier Rechnung getragen und im Folgenden zunächst ein Modell zur Bewertung von Software hergeleitet und anschließend durch Anwendung dieses Modells die geeignetste Software zur Integration in eine Clusteranalyse-Umgebung bestimmt.

4.1 Software Evaluations Modelle

Die Disziplin der Bewertung und Auswahl von Software wird im Allgemeinen als „Software Evaluation“ bezeichnet. Um Software bewerten zu können, versucht man die Qualität von Software greifbar zu machen. Viele der heutzutage propagierten Ansätze gehen auf das Factor-Criteria-Metrics-Modell (kurz: FCM) zurück, das erstmals in [MRW77] beschrieben wurde. Hierbei werden relevante Qualitätsmerkmale (factors) der Software herausgearbeitet und diese in messbare Teilmerkmale (criteria) unterteilt. Diesen werden Metriken (metrics) zugeordnet, die die Eigenschaften der Teilmerkmale messbar machen. Das Ergebnis dieser Verfeinerung wird als Qualitäts-Modell bezeichnet.

Bislang konnte sich noch kein Modell zur Software Evaluation durchsetzen. Das bekannteste Beispiel für ein Qualitäts-Modell liefert der Standard ISO/IEC 9126 (siehe [ISO01b]), der versucht eine einheitliche Terminologie im Bereich der Software-Qualität zu schaffen und die wichtigsten Qualitätseigenschaften in strukturierter Weise zusammenzufassen. Der Standard ISO/IEC 14598 (siehe [ISO01a]) beschreibt den Prozess der Software Bewertung und teilt das Vorgehen in sequentielle Schritte ein. Der Standard basiert auf dem Qualitäts-Modell des Standard ISO/IEC 9126. Sinn eines solchen Vorgehensmodells der Software

Evaluation ist zum einen dem Anwender ein strukturiertes Vorgehen zu ermöglichen und zum anderen den Bewertungsprozess so objektiv wie möglich zu halten. Da Software in vielen unterschiedlichen Variationen existiert, wird angeraten, die beiden genannten ISO-Standards als Referenzen anzusehen und an die jeweiligen Bedingungen anzupassen.

Ein etwas überschaubareres Modell der Software Evaluation stellt das offene und standardisierte „Business Readiness Rating“[1] Modell (BRR-Modell) dar, das in einer Kooperation zwischen dem Center for Open Source Investigation at Carnegie Mellon West[2] und Intel entstand. Es ermöglicht eine vollständige, einfache, anpassungsfähige und konsistente Methode zur Bewertung von Software, wobei es sich vorwiegend auf Open Source Software spezialisiert, die Bewertung proprietärer Software aber nicht ausschließt. Das BRR-Modell definiert ein Qualitäts-Modell mit 12 Qualitätsmerkmalen (siehe Tabelle 4.1), aus denen die wichtigsten für die jeweilige Software auszuwählen und entsprechend zu gewichten sind. Das Vorgehensmodell unterteilt sich in 4 Phasen (siehe Abbildung 4.1).

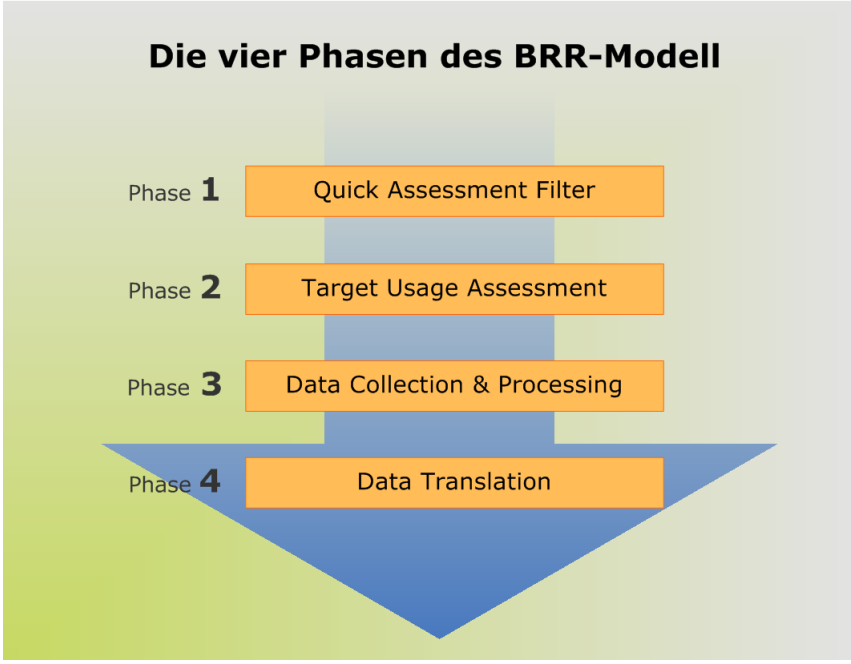


Abbildung 4.1: Die vier Phasen des BRR-Modell

Die Tätigkeiten innerhalb der einzelnen Phasen sind:

1. Quick Assessment Filter

   In dieser Phase findet eine erste Aussortierung von Software aus dem gesammelten Pool mithilfe von initialen groben Filtern statt. Es wird eine Shortlist aus der in die engere Auswahl kommenden Software erstellt.

[1] http://www.openbrr.org
[2] http://cosi.west.cmu.edu

2. Target Usage Assessment

   In der 2. Phase werden die wichtigsten Qualitätsmerkmale ausgewählt und gewichtet. Außerdem werden Teilmerkmale und damit Metriken pro Qualitätsmerkmal bestimmt und deren Gewichtung innerhalb der Qualitätsmerkmale festgelegt.

3. Data Collection and Processing

   Hier werden den Metriken die tatsächlichen Werte der zu überprüfenden Software zugeordnet und der Ergebniswert pro Qualitätsmerkmal berechnet.

4. Data Translation

   In der letzten Phase wird das Endergebnis aus der Summe der einzelnen Qualitätsmerkmale berechnet. Außerdem schlägt das BRR-Modell vor, die Ergebnisse zu veröffentlichen, damit sie von anderen genutzt bzw. geprüft werden können.

Weitere Details bzgl. des OpenBRR-Modells finden sich auf der angegebenen Webseite.

4.2 Ein eigenes Software Evaluations Modell

Das BRR-Modell zur Evaluation von Software ist relativ stark an das von der ISO propagierte Modell aus ISO/IEC 9126 und ISO/IEC 14598 angelehnt. Das ISO-Modell zeugt jedoch von einer Komplexität und Feingranularität die für unsere Zwecke nicht angemessen ist. Das BRR-Modell stellt eine einfache aber vollständige Methode zur Software Evaluation dar und ist gerade aufgrund seiner Anpassungsfähigkeit die ideale Methode zur Bewertung und Vergleich von Clusteranalyse Software. Im Folgenden werden die notwendigen Anpassungen des BRR-Modells für unsere Zwecke beschrieben.

Der ersten Phase des BRR-Modells wird eine Phase „Erstellen einer Longlist“ vorangestellt, in der das Suchen, Sammeln und Dokumentieren jeglicher Software stattfindet, die für den Anwendungsbereich potentiell einsetzbar ist. Diese Phase sollte nach Meinung des Autors ein integraler Bestandteil des Software Evaluation Prozesses sein, damit für den Betrachter des Ergebnisses des Prozesses nachvollziehbar ist, welche Software - und somit auch welche nicht - ursprünglich zur Auswahl stand.

Die Phasen 1, 2 und 3 aus dem ursprünglichen BRR-Modell werden - bis auf die Bezeichnung (siehe Abbildung 4.2) - unverändert übernommen.

Die letzte Phase des BRR-Modell „Data Translation“ wird um eine abschließende Gegenüberstellung der bewerteten Software erweitert, die als Resultat - basierend auf den mathematischen Berechnungen der vorhergehenden Phasen - die am besten geeignete Software hervorbringen soll. Die Phase wird mit „Gegenüberstellen der Ergebnisse“ betitelt und dient vor allem als Zusammenfassung der vorangehenden Auswertung. Dies soll dem Betrachter des Software Evaluations Prozesses die Stärken und Schwächen der einzelnen Softwarelösungen verdeutlichen und einen Überblick über das Ergebnis schaffen.

Zusammenfassend findet sich in Abbildung 4.2 das Vorgehensmodell der Software Evaluation.

Abbildung 4.2: Die fünf Phasen des Vorgehensmodell der Software Evaluation

Das Qualitäts-Modell wird unverändert aus BRR übernommen (siehe Tabelle 4.1).

4.3 Anwendung des Modells auf Software zur Clusteranalyse

Das hier vorgestellte Modell zur Software Evaluation soll nun auf Software zur Clusteranalyse angewandt werden. Da die Funktionalität der Clusteranalyse oftmals in Form von vollständigen Statistikwerkzeugen angeboten wird, werden auch solche Lösungen berücksichtigt. Außerdem werden neben lauffähigen Programmen auch Bibliotheken betrachtet. Die Wahl der Filter in Phase 1 und der Qualitätsmerkmale und Metriken in Phase 3 muss also so gestaltet werden, dass sie auf alle betrachteten Softwarevarianten anwendbar sind.

Im Folgenden werden die Phasen des in Kapitel 4.2 vorgestellten Vorgehensmodells durchgeführt.

An dieser Stelle ist anzumerken, dass die Evaluation der Software hier im Hinblick auf die Nutzung der Software als integraler und wesentlicher Bestandteil einer Umgebung zur Clusteranalyse auf Softwareelementen durchgeführt wird. Die Wahl der Filter und Qualitätsmerkmale sind dementsprechend ausgerichtet. Die folgende Bewertung und das Ergebnis derselben beurteilt nicht die allgemeine Qualität der jeweiligen Software!

Qualitätsmerkmal	Beschreibung
Funktionalität	Wie gut erfüllt die Software die Anforderungen des Benutzers?
Benutzerfreundlichkeit	Wie gut ist das User Interface? Wie einfach ist die Software zu installieren, zu benutzen und zu konfigurieren?
Qualität	Von welcher Qualität ist das Design, der Code und die durchgeführten Tests? Ist die Software fehlerfrei?
Architektur	Wie modular, portabel, flexibel, erweiterbar, offen und integrierbar ist die Software?
Support	Wie gut ist der Support?
Dokumentation	Von welcher Qualität und Umfang ist die Dokumentation der Software?
Sicherheit	Wie sicher ist die Software? Wie geht sie mit Sicherheitslücken um?
Performanz	Wie ist die Performanz der Software? Wie sparsam geht die Software mit Ressourcen um?
Skalierbarkeit	Wie gut kann die Software mit vielen Nutzern oder Betriebsmitteln umgehen?
Akzeptanz	Wie wird die Software von der Community, dem Markt, der Industrie, der Forschung angenommen?
Community	Wie aktiv ist die Community für die Software?
Professionalität	Wie ist der Grad der Professionalität des Entwicklungsprozesses der Software?

Tabelle 4.1: Qualitäts-Modell der Software Evaluation

4.3.1 Erstellen einer Longlist

In dieser Phase findet das Suchen, Sammeln und Dokumentieren von Software zur Clusteranalyse statt. Ergebnis ist eine Longlist von Softwarelösungen, die dem weiteren Software Evaluations Prozess unterzogen werden.

Zum Suchen von Software diente als primäre Quelle das WWW. Als guter Einstiegspunkt eignet sich das Statistikportal http://www.kdnuggets.com/ (7. August 2006), das eine umfangreiche Auflistung von sowohl kommerzieller als auch freier Software rund um Data Mining und Knowledge Discovery bietet. Weiterhin wurden diverse Suchmaschinen eingesetzt. Bei der Suche nach Software zur Clusteranalyse musste ich feststellen, dass sehr wenige Lösungen existieren, die sich ausschließlich mit Clusteranalyse beschäftigen. Vielmehr findet man Software, die die Clusteranalyse als festen Bestandteil einer kompletten Data Mining Lösung beinhalten. Reine Clusteranalyse-Lösungen finden sich fast ausschließlich im Bereich der Bioinformatik bzw. Genforschung, wobei hier - aufgrund der immens großen Datensätze - abgewandelte Algorithmen und vor allem spezielle Ergebnispräsentationen der Clusteranalyse zur Anwendung kommen (vgl. [ESPD98]).

Das Sammeln von Software war geprägt von einem ersten groben Selektionsprozess. Software die sich offensichtlich nicht für unsere Zwecke eignet, wie z.B. die oben angesprochenen Lösungen im Genetik Bereich, die einen zu hohen Spezialisierungsgrad aufweisen, werden bei der folgenden Dokumentation nicht mehr berücksichtigt. Zusätzlich werden in diesem Teilschritt erste Daten - im Sinne von Eigenschaften - von infrage kommenden Softwareprodukten gesammelt.

Die Dokumentation der gesammelten Software findet in Tabellenform statt. Tabelle 4.2 zeigt zunächst die Quellen der einzelnen Lösungen.

Software	**Quelle**
ADaM	http://datamining.itsc.uah.edu/adam/
AlphaMiner	http://www.eti.hku.hk/alphaminer/
Clustan	http://www.clustan.com/
Cluster 3.0	http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/cluster/
Cluto	http://glaros.dtc.umn.edu/gkhome/views/cluto
Clusterer	http://web.mit.edu/polz/clusterer/
Jung	http://jung.sourceforge.net/
Past	http://folk.uio.no/ohammer/past/
PermutMatrix	http://www.lirmm.fr/~caraux/PermutMatrix/EN/index.html
R	http://www.r-project.org/
S-Plus	http://www.insightful.com/products/splus/
SPSS	http://www.spss.com/de/spss/
Tanagra	http://eric.univ-lyon2.fr/~ricco/tanagra/en/tanagra.html
Weka	http://www.cs.waikato.ac.nz/ml/weka/
Yale	http://sf.yale.net

Tabelle 4.2: Clusteranalyse-Software mit Quellenangabe

Abbildung 4.3 stellt eine etwas detailliertere Übersicht über die Softwareprodukte dar, mit ersten Eigenschaften, die die jeweilige Software charakterisiert. Die Bedeutung der einzelnen Spaltenbezeichnungen von Abbildung 4.3 wird im Folgenden kurz erklärt:

1. In der ersten Spalte findet sich der *Name der Software*,
2. gefolgt von dem *Typ*, wobei hier die Bezeichnung übernommen wurde, die der Hersteller der Software ihr gegeben hat.
3. Die folgende Spalte beschreibt die *Schnittstellen* und unterteilt sich in
   a) die *Input*-Dateiformate die unterstützt werden und
   b) einem Flag, das angibt, ob die Software eine *API* zur Verfügung stellt.
4. Die Spalte *Support* teilt sich in *Verfügbarkeit*, *Quellcode*, *Sprache* und *Dokumentation / Manual*.

a) Die *Verfügbarkeit* gibt an, ob es sich um ein kommerzielles oder freies Produkt handelt, wobei bei den freien Produkten, wenn vom Hersteller angegeben, die Lizenz aufgelistet wird, unter die die Software fällt.

b) In der Spalte *Quellcode* wird ersichtlich, ob der Quellcode der Software freigegeben ist.

c) Die Spalte *Sprache* (Programmiersprache) gibt, falls bekannt, die Programmiersprache an, mit der die Software erstellt wurde. Ist diese nicht bekannt, findet sich ein Schrägstrich in der entsprechenden Zelle.

d) In der letzten Spalte wird zwischen *Dokumentation und Manual* unterschieden. Unter Dokumentation wird hier eine Beschreibung der Umsetzung der angewandten Algorithmen verstanden, eine Beschreibung der Programmschnittstellen, bzw. bei Lösungen deren Quellcode verfügbar ist, eine Dokumentation desselben. Ein Manual ist eine Art User Guide, d.h. eine Beschreibung, wie die Software zu benutzen ist.

Software	Typ	Schnittstellen		Support			
		Input[1]	API	Verfügbarkeit	Quellcode	Sprache	Doku/Manual
ADaM	Statistische Analyse	arff, images	ja	frei	nein	C, C++	nein / ja
AlphaMiner	Data Mining Software	arff, db, prop, txt, xls	nein	frei[2]	nein	Java	nein / ja
Clustan	Clusteranalyse Software	unbekannt	nein	kommerziell	nein	/	nein / ja
Cluster 3.0	Clusteranalyse Software	txt	ja	frei[3]	ja	C, Phyton	ja / ja
Cluto	Clusteranalyse Software	txt	ja	frei	ja	C, C++	ja / ja
Clusterer	Clusteranalyse Software	fasta, nexus	ja	frei[4]	ja	Java	nein / ja
Jung	Klassenbibliothek	selbst zu definieren	ja	frei[5]	ja	Java	ja / ja
Past	Data Mining Software	nexus, txt	nein	frei	nein	/	nein / ja
PermutMatrix	Grafische Analyse von Daten	txt	nein	frei	nein	C	nein / nein
R	Statistische Analyse	db, prop, txt, xls, xml	ja	frei[2]	ja	R	ja / ja
S-PLUS	Statistische Analyse	db, prop, txt, xls	nein	kommerziell	nein	S	nein / ja
SPSS	Data Mining Software	db, prop, txt, xls	nein	kommerziell	nein	/	ja / ja
Tanagra	Data Mining Software	arff, txt, xls	ja	frei	ja	Delphi	ja / ja
WEKA	Data Mining Software	arff, db, txt	ja	frei[2]	ja	Java	ja / ja
Yale	Data Mining Software	arff, db, txt, xml	ja	frei[2]	ja	Java	ja / ja

[1] arff/ fasta / nexus= Formate zum Speichern von Matrizen in Textdateien, db=Datenbank, prop=proprietäre Formate, txt=Textdatei
[2] GNU General Public License [3] Phyton License, Artistic License [4] Open Source Compliant License [5] BSD License

Abbildung 4.3: Clusteranalyse-Software Übersicht

4.3.2 Erstellen einer Shortlist

Das Ziel dieser Phase ist die Reduktion der Liste der Software auf diejenigen, die die vielversprechensten Merkmale aufweisen, da eine Auswertung jeder Software gemäß Phase 4 zu aufwendig wäre. Es findet also ein Aussiebungsprozess statt, wobei geeignete Filter gewählt werden müssen, die in diesem Prozess als Sieb fungieren. Das Ergebnisdokument dieser Phase ist eine Shortlist, die diejenigen Lösungen enthält, die in die engere Auswahl kommen. Zunächst werden die Filter definiert und eine kurze Begründung für die Wahl des Filters gegeben. Anschließend wird jede Software gegen jeden Filter getestet und beschrieben, welche Software ausgefiltert wird und welche nicht.

Folgende Filter werden definiert:

- Funktionalität / Erweiterbarkeit
- Dokumentation
- Kosten

Die Filter werden im Folgenden näher erläutert.

Funktionalität / Erweiterbarkeit

Eine Software, die integraler Bestandteil einer Clusteranalyse-Umgebung werden soll, sollte natürlich ein Minimum an Funktionalität diesbezüglich aufweisen können. Dazu gehören:

- Flexibler offener Datenimport
- Erstellen einer Objekt-Eigenschaften Matrix
- Erstellen einer Distanzmatrix über verschiedene Distanzmaße
- Algorithmen
  - Hierarchische Verfahren (Agglomerative,...)
  - Partitionierende Verfahren (K-means,...)
- Algorithmen parametrisierbar mit verschiedenen Linkage Kriterien
  - Single Linkage
  - Complete Linkage
  - Average Linkage
- Bereitstellen der Ergebnisse in einem auslesbaren Format

Fehlt es der Software an einer / einigen dieser Funktionalitäten, sollte sie über die Möglichkeit verfügen, diese Funktionalitäten hinzuzufügen (Erweiterbarkeit).

Dokumentation

Da die Software als Bestandteil einer Clusteranalyse-Umgebung vorgesehen ist, muss sie zur Anbindung an andere Softwarekomponenten über entsprechende Schnittstellen verfügen. Von besonderer Bedeutung sind dabei die Datenimport und -export Eigenschaften (siehe Funktionalität / Erweiterbarkeit). Das Anbinden an andere Softwarekomponenten verlangt mindestens Kenntnisse über diese Schnittstellen, außerdem Kenntnisse über die Interna der Software. Eine Beschreibung der API sollte also in Form einer Dokumentation vorliegen.

Kosten

Da es sich um ein rein wissenschaftliches Projekt handelt, sollten die Kosten für die Software nicht zu hoch sein. Insbesondere dann, wenn vom Umfang her vergleichbare Software frei verfügbar ist.

Tabelle 4.3 zeigt zusammenfassend welche Software durch welchen Filter aussortiert und welche Software in die Shortlist aufgenommen wird. Anschließend folgt für jede Software eine kurze Begründung wieso sie ausgefiltert wird, bzw. aufgrund welcher Eigenschaften sie in die Shortlist aufgenommen wird. Ein (+) hinter der Beschreibung verdeutlicht nochmals die Aufnahme der jeweiligen Software in die Shortlist, ein (-) zeigt an, dass die Software ausgefiltert wird.

	Filter				
Software	**Funktionalität**	**Dokumentation**	**Kosten**	**Andere**	**Shortlist**
ADaM	**x**	**x**			
AlphaMiner	**x**	**x**			
Clustan		**x**	**x**		
Cluster 3.0	**x**	**x**			
Cluto	**x**				
Clusterer		**x**			
Jung	**x**				
Past		**x**			
PermutMatrix	**x**	**x**			
R					**x**
S-Plus			**x**		
SPSS					**x**
Tanagra		**x**		**x**	
Weka				**x**	
Yale					**x**

Tabelle 4.3: Ausfilterung von Software

Die Lösung **ADaM** ist eine Sammlung von Algorithmen zur Statistischen Analyse. Die Algorithmen stehen in Form von ausführbaren Dateien zur Verfügung, die von eigenen Softwarelösungen benutzt werden können. Der Quellcode steht nicht zur Verfügung und eine Anpassung / Erweiterung der Algorithmen ist nicht vorgesehen. Die Wahl eines Linkage Kriteriums oder eines Dinstanzmaßes sind nicht möglich, die Ergebnisinformationen sind wenig umfangreich. Die Dokumentation ist zudem mangelhaft. (-)

AlphaMiner bietet als Clusteringalgorithmus lediglich den K-Means Algorithmus an. Der Quellcode steht nicht zur Verfügung und eine Erweiterung ist nicht vorgesehen. (-)

Clustan ist eine kommerzielle Clusteranalyse Software. Die Kosten für eine akademische Lizenz betragen 125$. Über die Schnittstellen der Software finden sich auf der Webseite keine Angaben. Eine E-Mail-Anfrage an den Hersteller ergab, dass Excel Dateien oder Textdateien (Comma oder Tabulator Seperated Values) verarbeitet werden können. Der Quellcode ist nicht verfügbar, eine Dokumentation über die Interna der Software existiert nicht. Da die Funktionalität der Software diejenige von Open Source Lösungen nicht übersteigt, wird `Clustan` nicht in die Shortlist aufgenommen. (-)

Cluster 3.0 ist eine Software, die für den Forschungsbereich der Genanalyse erstellt wurde. Dies drückt sich z.B. durch den Input aus, denn hier wird eine Sequenz von Geninformationen erwartet. Die Software erweitert die im Bereich der Genforschung bekannte Software von Eisen[3], worunter die Struktur der Software etwas gelitten hat. Die aufgrund der hohen Spezialisierung notwendig werdenden Anpassungen der Software werden durch die schlechte Struktur des Programmcodes und die lückenhafte Dokumentation stark erschwert (-)

Cluto ist ein Clusteranalyse Tool zur Analyse von Gendaten. Das Tool ist kommandozeilenbasiert, als Input Format stehen lediglich softwarespezifische Formate zur Verfügung. Als Clusteralgorithmen werden nur hierarchische Verfahren angeboten, die Ergebnisse werden wieder in einem eigenen Format abgelegt. Der Quellcode und eine Dokumentation sind vorhanden, aufgrund der eingeschränkten Funktionalität wäre der Aufwand der Anpassung aber - im Vergleich zu anderen Lösungen - zu hoch. (-)

Clusterer ist eine kleine Java-Anwendung, die als Client-Applikation, Applet und Web Start verfügbar ist. Es existiert keine Dokumentation, der Quellcode ist zudem mangelhaft kommentiert und dokumentiert. (-)

Die Java-Klassenbibliothek **Jung** (Java Universal Network/Graph Framework) bietet gemeinsame und erweiterbare Konzepte zur Modellierung, Analyse und Visualisierung von Daten die als Graph oder Netzwerk dargestellt werden können. Die Funktionalität, die hier im Bereich der Clusteranalyse zur Verfügung gestellt wird, befindet sich auf einem sehr abstrakten Niveau. `Jung` würde sich ausgezeichnet für die Neuentwicklung eines Clusteranalyse Tools in Java eignen und wird auch von den meisten hier vorgestellten Softwarelösungen, die in Java realisiert sind, genutzt. Als fertige Lösung zur Integration in eine Clusteranalyse-Umgebung reicht die Funktionalität nicht aus. (-)

[3] http://rana.lbl.gov/EisenSoftware.htm

Past wurde für die Datenanalyse im Bereich der Paläontologie entwickelt. Der Quellcode steht nicht zur Verfügung und eine Erweiterung ist nicht vorgesehen. Außerdem fehlt eine Dokumentation der Software. (-)

PermutMatrix ist eine reine Clusteranalyse Software für den Bereich der Genforschung. Es wird lediglich ein hierarchischer Clusteralgorithmus angeboten, der Quellcode steht nicht zur Verfügung und es existiert weder eine Dokumentation noch ein Manual. (-)

R 2.3.0 ist sowohl eine eigene Programmiersprache als auch eine Umgebung für statistische Berechnungen und Grafiken. R ist eine Open Source Software unter den Bedingungen der GNU General Public License. Die Software stellt bereits eine breite Palette an statistischen Algorithmen bereit - Clusteranalyse inbegriffen - und ist zudem in hohem Maße über Packages erweiterbar. C und Fortran Programme können direkt auf R-Objekte zugreifen, eine Anbindung an Java ist ebenfalls realisiert, dieses Projekt steckt jedoch noch in den Kinderschuhen[4]. Die Software an sich ist kommandozeilenbasiert und zur Benutzung benötigt man eine gewisse Einarbeitungszeit. Die Mächtigkeit und Erweiterbarkeit von R wird in Fachkreisen geschätzt. Somit erfreut sich die Software einer recht großen Community und infolgedessen einer Vielzahl an Erweiterungen (packages). Die Dokumentation zu der Basissoftware und der R-Sprache ist umfangreich. Es erscheinen in regelmäßigen Abständen neue Versionen mit neuer Funktionalität und Fehlerbehebungen. Aufgrund der Funktionalität von R, seiner Erweiterbarkeit, der guten Dokumentation und der freien Verfügbarkeit übersteht R alle Filter und wird in die Shortlist aufgenommen. (+)

S-Plus ist eine mächtige kommerzielle Software zur Statistischen Datenanalyse. Im Bereich der Clusteranalyse werden hierarchische (agglomerativ, divisiv) und partitionierende (K-means, medoids) Algorithmen angeboten. Die Kosten für eine einjährige Lizenz betragen 475$. Da S-Plus die Funktionalität der ebenfalls kommerziellen Software SPSS im Bereich Clusteranalyse nicht übersteigt, für SPSS jedoch bereits eine Lizenz vorliegt (siehe unten), wird von einer Anschaffung von S-Plus abgesehen. (-)

Die kommerzielle Software **SPSS 14.0** (Statistical Package for the Social Sciences) ist eine der am weit verbreitesten Data Mining Software weltweit. Die Funktionalität ist extrem umfangreich, der Bereich der Clusteranalyse nimmt jedoch einen nur sehr kleinen Anteil an der Gesamtfunktionalität ein. Dennoch fällt SPSS auch in diesem Bereich mit drei Clusteralgorithmen, die in vielfältigerweise parametrisierbar sind und eine ordentlich aufbereitete Ergebnispräsentation aufweisen, positiv auf. Der Datenimport unterstützt als Dateiformate MS Excel (.xls), .txt (Comma oder Tabulator Seperated Values), MS Access (.mdb), einige proprietäre Formate (SPSS, SAS, Lotus,...) und weitere Datenbanken, die über den ODBC Treiber angebunden werden können. Als Export Formate kommen XML, HTML, Textdateien oder proprietäre Formate in Frage, wobei verschiedene Formate nur eingeschränkte Teile des Ergebnisses darstellen können (z.B. Abbildungen). Eine Erweiterung von SPSS sowie eine externe Nutzung der Funktionalität ist - nach Rücksprache mit Experten und Studium von Dokumentation - nicht oder nur sehr eingeschränkt möglich (Erklärung siehe weiter unten im Anhang zu SPSS). Es existieren zahlreiche Dokumentation

[4] SJava 0.69, http://www.omegahat.org/RSJava/

und Publikationen (auch in Form von Büchern) über SPSS. Aufgrund der Kommerzialität des Produktes wird die Software regelmäßig gewartet und der Support ist entsprechend. Da die Hochschule, an der der Autor dieser Arbeit tätig ist[5], im Besitz einer Lizenz von SPSS ist, kann diese für die Arbeit genutzt werden. (+)

Tanagra ist eine Open Source Data Mining Software. Das Tool wurde in C++ und Delphi realisiert und ist eine Entwicklung einer Privatperson. Ein erstes Experimentieren an der Software zeigte, dass diese recht fehleranfällig ist. Die Dokumentation ist sehr knapp und zudem teilweise nur in Französisch erhältlich. (-)

Die freie Java-Lösung **Weka** erfreut sich in der Data Mining Szene großer Beliebtheit. Sie bietet neben dem K-means Algorithmus noch weitere partitionierende Algorithmen, ein hierarchischer Algorithmus fehlt jedoch. Der Quellcode steht inklusive ausführlicher Dokumentation zur Verfügung und eine API zur Erweiterung der Software existiert. Die ebenfalls in diesem Kapitel vorgestellte Software Yale nutzt Weka als Klassenbibliothek und bietet alle Operatoren und Funktionalität an die Weka zur Verfügung stellt. Yale kann vor allem im Bereich der Clusternanalyse gegenüber Weka mit zusätzlicher Funktionalität aufwarten. Weka übersteht alle Filter, wird jedoch nicht in die Shortlist aufgenommen, da Yale die Funktionalität von Weka vollständig integriert und zusätzlich erweitert. (-)

Die Open Source Software **Yale 3.2** ist eine Umgebung für Data Mining Anwendungen. Der Source Code ist verfügbar unter der GNU General Public License. Yale ist eine Java-Anwendung und der Clusteranalyse Teil ist als eigenständiges Modul (jar-Datei) realisiert. Es werden zahlreiche Clusteralgorithmen angeboten, die Möglichkeiten der Parametrisierung der Algorithmen mit Distanzmaßen und Linkage Kriterien fallen etwas knapp aus. Yale zeichnet sich durch seine Erweiterbarkeit und Nutzung als Bibliothek für eigene Lösungen aus. Die ausführliche Dokumentation zu Yale beinhaltet kurze Beschreibungen, wie die Yale-API für diese Zwecke genutzt werden kann. Yale unterstützt die Durchführung von Data Mining Experimenten auf verschiedene Weise, z.B. sind alle elementaren Operationen als eigene Komponente realisiert und ergeben erst durch Aneinanderkettung eine vollständige Datenanalyse. Die Ergebnisse aller Zwischenschritte können abgefragt und elementare Operationen in der Kette ausgetauscht werden. Yale nutzt intensiv XML, so stellt XML eine Möglichkeit zum Import von Daten und Export von Ergebnissen dar und wird außerdem genutzt, um die Ablauffolge eines Experimentes - d.h. die Aneinanderreihung von Operationen - zu beschreiben. Die Benutzung der GUI ist nicht intuitiv und Bedarf einiger Einarbeitungszeit. Yale wird im wissenschaftlichen Bereich weitverbreitet eingesetzt, wird jedoch auch von Unternehmen verwendet. (+)

Die Shortlist besteht also aus den Softwareprodukten

- R,
- SPSS und
- Yale,

[5] http://www.uni-koblenz.de

die bereits etwas näher beschrieben wurden. An diesen 3 Lösungen wird der folgende Evaluationsprozess durchgeführt.

4.3.3 Qualitätsmerkmale und Metriken

In dieser Phase werden die wichtigsten Qualitätsmerkmale der Zielsoftware aus Tabelle 4.1 ausgewählt und prozentual gewichtet, so dass die Summe der Einzelgewichtungen 100% ergibt. Innerhalb der einzelnen Qualitätsmerkmale werden Metriken definiert, denen wiederum eine prozentuale Gewichtung innerhalb des Qualitätsmerkmals zugeordnet wird. Den Metriken werden dann in der nächsten Phase Punkte zwischen 1 und 5 zugeordnet. Hier wird also ein Template erstellt, das als Grundlage für die einheitliche Bewertung der Softwareprodukte der Shortlist dient.

Die Auswahl der relevanten Qualitätsmerkmale, die Gewichtung derselben und das Zuordnen von Metriken ähnelt dem Erstellen einer Anforderungsliste. Das Template beschreibt, mit welchen Eigenschaften die gewünschte Software optimalerweise ausgestattet sein soll (und zwar jene Eigenschaften, denen die maximale Punktzahl zugeordnet ist).

In Abbildung 4.4 finden sich zunächst die gewünschten Qualitätsmerkmale der Software mit der jeweiligen Gewichtung. Die Funktionalität der zu bewertenden Software stellt mit 35% Gewichtung das wichtigste Qualitätsmerkmal dar, da die Funktionalität - konkreter: die Funktionalität der Clusteranalyse - der Teil ist, der von der zu erstellenden Software genutzt werden soll. Die Architektur der Software bildet ebenfalls einen gewichtigen Teil, da sie wichtige Eigenschaften für die nahtlose Einbindung der Software in eine Umgebung charakterisiert. Eine ausgesprochen gute Dokumentation ist notwendig, um die Software zu verstehen und damit die Integration in eine Umgebung in angemessener Zeit erst zu ermöglichen und zu unterstützen. Der Support und die Community sind verwandte Merkmale, da die Community häufig zum Support beiträgt, vor allem bei Open Source Software. Der Support wird notwendig, wenn sich Probleme ergeben, die der Anwender selbst nicht mehr oder nur unter sehr großem Zeitaufwand lösen kann und auf fremde Hilfe angewiesen ist.

Den gewichteten Qualitätsmerkmalen müssen nun noch Metriken zugeordnet werden. Da die entstehende Liste sehr umfangreich ist, wurde sie aus Gründen der Übersichtlichkeit nicht in dieses Paper eingebunden. Das komplette Dokument, das in der folgenden Phase als Template zur Bewertung der Software aus der Shortlist verwendet wird, findet sich auf der dieser Arbeit beiliegenden CD unter dem Dateinamen „Software Evaluation_Clusteranalyse_Template.xls“. Es wurde versucht, die Metriken so objektiv wie möglich erfassbar zu machen. Die Punkteskala der Metriken erstreckt sich zwischen 1 und 5, wobei es sich hier um eine ordinale Skala handelt. Die einzelnen Punkte haben folgende Bedeutung:

- 1 = inakzeptabel
- 2 = schlecht
- 3 = akzeptabel

Software Evaluation

Komponenten Name: Clusteranalyse Software

Rang	Qualitätsmerkmal	Gewichtung
1	Funktionalität	35,00%
2	Architektur	30,00%
3	Dokumentation	20,00%
4	Support	10,00%
5	Community	5,00%
		Gewichtung Total
		100,00%

Abbildung 4.4: Qualitätsmerkmale

- 4 = sehr gut
- 5 = ausgezeichnet

Die Bewertung des Qualitätsmerkmales Funktionalität unterscheidet sich etwas von den übrigen Merkmalen, da die Funktionalität von Software auf einer ordinalen Skala schlecht zu erfassen ist. Eine Software besitzt eine Funktionalität (Eigenschaft) oder nicht, somit handelt es sich hier um eine rein binäre Bewertung der Form „hat Eigenschaft“ oder „hat Eigenschaft nicht“. Zusätzlich gibt es Eigenschaften, die für die Zielsoftware sehr wichtig sind und solche, die weniger wichtig („nice to have“) sind. Das Tabellenblatt „Funktionalität“ der Excel-Datei (siehe CD) beschäftigt sich mit der gesonderten Behandlung des Qualitätsmerkmals Funktionalität. Zunächst wird eine Liste von Funktionalitäten erstellt, die die Zielsoftware besitzen sollte. Die einzelnen Funktionalitäten werden gemäß ihrer Bedeutung in Form von Punkten gewichtet, mit 1 für weniger wichtig bis 3 für sehr wichtig. Besitzt die zu bewertende Software die Eigenschaft, werden ihr die Punkte gutgeschrieben, ansonsten erhält sie 0 Punkte. Durch die Aufsummierung der einzelnen Punkte erhält man die Gesamtpunktzahl der Funktionalität, die an der maximal erreichbaren Punktzahl gemessen wird. Dieser erreichte Prozentsatz der maximalen Punktzahl muss nun noch auf die ordinale Bewertungsskala zwischen 1 und 5 der anderen Qualitätsmerkmale normiert werden. Die Umrechnungstabelle findet sich im Excel-Template.

4.3.4 Datensammlung und Auswertung

In dieser Phase wird ermittelt, welche Software der im Template definierten optimalen Software am nächsten kommt, oder anders ausgedrückt: Welche Software die geeignetsten Ei-

genschaften aufweist, um einen integralen Bestandteil einer Umgebung zur Clusteranalyse zu bilden.

Dazu muss jede Software detailliert untersucht und dabei den gelisteten Qualitätsmerkmalen und ihren Metriken besondere Beachtung geschenkt werden. Die detaillierten Ergebnisse der Bewertung finden sich als Excel-Datei mit dem Dateinamen
„Software Evaluation_Clusteranalyse_Auswertung.xls“.

Die Bewertung der Ergebnisse und eine Gegenüberstellung der untersuchten Softwareprodukte werden in der folgenden Phase durchgeführt.

4.3.5 Gegenüberstellen der Ergebnisse

Abschließend sollen hier die berechneten Ergebnisse in einer Gegenüberstellung der einzelnen Softwareprodukte kurz kommentiert und anschließend die Software, die sich als die geeignetste darstellt, gewählt werden. In Abbildung 4.5 findet sich zunächst eine Zusammenfassung der Bewertungen der einzelnen Softwareprodukte.

Erwartungsgemäß schneidet die kommerzielle Software `SPSS` im Bereich der Funktionalität sehr gut ab und kann sich im Vergleich mit den beiden anderen Lösungen durchsetzen. Der entscheidende Minuspunkt von `SPSS` für unsere Zwecke ist die Architektur, wobei hier besonderer Wert auf eine offene Architektur gelegt wurde. Leider bietet `SPSS` keine Möglichkeit die Software zu erweitern, anzupassen oder Funktionalität extern zu nutzen. Somit würde sich eine Integration von `SPSS` in eine Umgebung als problematisch erweisen. In diesem Bereich liegen die Stärken von `R` und `Yale`, und somit konnten diese Lösungen im Qualitätsmerkmal Architektur die volle Punktzahl erzielen (siehe Anhang). `Yale` bietet im Bereich der Clusteranalyse gegenüber `R` etwas mehr an Funktionalität, vor allem die Unterstützung von Experimenten in `Yale` ist ein Pluspunkt. In den Qualitätsmerkmalen Dokumentation und Support konnten alle Lösungen punkten und unterscheiden sich kaum. Die Community ist bei `Yale` wesentlich kleiner als bei den beiden anderen Lösungen, der Trend ist jedoch steigend und die ausgesprochen gute Dokumentation lässt hoffen, dass eine Zuhilfenahme der Community selten notwendig ist. Trotzdem verfügt `Yale` über einen guten Support: Forum, Mailing List, Bug Report und Feature Request-Möglichkeiten sind vorhanden, zuständige Admins reagieren schnell auf eingestellte Fragen.

`Yale` wurde in der mächtigen und vielseitigen Programmiersprache Java realisiert, `R` in einer eigenen Sprache, die stark auf statistische Anwendungen spezialisiert ist. Das Arbeiten mit einer in Java implementierten Software erhöht die Flexibilität in Bezug auf Erweiterungen, Wartung und Anbindung an andere Software.

`Yale` stellt sich gemessen an den ausgewählten Qualitätsmerkmalen und Metriken als die geeignetste Software zur Integration in eine Umgebung zur Clusteranalyse auf Softwareelementen heraus. Die oben aufgeführten Argumente untermauern die Ergebnisse des Software Evaluationsprozesses.

Software Evaluation

Komponenten Name: R 2.3.0

Qualitätsmerkmal	ungewichtete Wertung	Gewichtung	gewichtete Wertung
Funktionalität	3	35,00%	1,05
Architektur	5	30,00%	1,5
Dokumentation	5	20,00%	1
Support	3	10,00%	0,3
Community	3	5,00%	0,15
		Gewichtung Total	**GESAMT**
		100,00%	**4**

Software Evaluation

Komponenten Name: SPSS 14.0

Qualitätsmerkmal	ungewichtete Wertung	Gewichtung	gewichtete Wertung
Funktionalität	5	35,00%	1,75
Architektur	1,8	30,00%	0,54
Dokumentation	4	20,00%	0,8
Support	5	10,00%	0,5
Community	5	5,00%	0,25
		Gewichtung Total	**GESAMT**
		100,00%	**3,84**

Software Evaluation

Komponenten Name: Yale 3.2.0

Qualitätsmerkmal	ungewichtete Wertung	Gewichtung	gewichtete Wertung
Funktionalität	4	35,00%	1,4
Architektur	5	30,00%	1,5
Dokumentation	5	20,00%	1
Support	5	10,00%	0,5
Community	2	5,00%	0,1
		Gewichtung Total	**GESAMT**
		100,00%	**4,5**

Abbildung 4.5: Ergebnis der Software Evaluation

5 Die Software Yale

5.1 Einführung

Yale ist eine Abkürzung für „Yet Another Learning Environment“ und wurde an der Artificial Intelligence Unit der Universität Dortmund[1] entwickelt. Die Software stellt eine Plattform zur Durchführung von Data Mining Analysen zur Verfügung.

Yale ist eine freie Open Source Software unter den Bedingungen der GNU General Public License. Vom Autor wurde Yale in der Version 3.2.0 untersucht. Da Yale komplett in Java realisiert wurde, ist die Software plattformunabhängig, was eine einfache Portabilität der Software garantiert. Der Quellcode, sowie die mit dem Tool JavaDoc generierte Dokumentation, sind auf der Yale Homepage[2] verfügbar. Die Software liegt in einem komprimierten Archiv vor. Um die Software zu installieren muss man lediglich das Archiv entpacken, wodurch eine Verzeichnisstruktur angelegt wird. Gestartet wird die Software entweder über die Datei yale.jar (plattformunabhängig) oder über eine plattformabhängige Batchdatei aus dem Verzeichnis <yale_home>/bin, wobei hier explizit die Möglichkeit besteht Yale mit oder ohne GUI zu starten.

Im Folgenden werden in Kapitel 5.2 zunächst die charakteristischen Konzepte von Yale behandelt. Dem Leser werden anhand eines konkreten Beispiels einer Data Mining Analyse, durch das Überführen des Beispiels in eine Yale-konforme Darstellung in mehreren Schritten, die Konzepte von Yale näher gebracht. Dabei wird der Aufbau der Software und der Ablauf bei ihrer Anwendung deutlich. In Kapitel 5.3 werden die möglichen Benutzerschnittstellen aufgezeigt und jeweils kurz erklärt. Abschließend behandelt Kapitel 5.4 die von Yale zur Verfügung gestellte Programmierschnittstelle.

Als Quelle für folgende Ausführungen dienten im wesentlichen der Quellcode der Software und dessen Dokumentation sowie die Arbeiten [MFK06], [Wur06] und [RKFM01].

5.2 Konzepte

Das Verständnis der Konzepte von Yale und deren Eignung für Data Mining Aufgaben setzt Kenntnisse über den groben Ablauf von Data Mining Analysen voraus. Für Leser, die sich mit diesem Themengebiet noch nicht befasst haben, wird im Folgenden eine kurze Einführung gegeben.

[1] http://www-ai.cs.uni-dortmund.de/index.html
[2] http://yale.sf.net

Data Mining Analysen gliedern sich häufig in die Anwendung mehrerer Verfahren, die aufgrund der Art ihrer Tätigkeiten in verschiedene Gruppen eingeteilt werden können. Häufig vorzufindende Gruppen - im Sinne von Oberbegriffen für Verfahren mit ähnlicher Tätigkeit - im Bereich Data Mining sind die Datenvorverarbeitung, die Lernprozesse[3], die Auswertung und die Visualisierung.

Data Mining Analysen werden - wie der Name schon vermuten lässt - auf Daten ausgeführt, um aus diesen neue Erkenntnisse zu gewinnen. Die Daten liegen dabei meistens in einer Datentabelle vor, wobei jede Zeile ein Objekt repräsentiert und die Spalten die Eigenschaften (Attribute) dieser Objekte. Jedes Objekt ist somit definiert als ein Vektor aus Attribut-Werte-Paaren. Ein Objekt hat dabei keine eindeutige Identifikation und es kann zu Mehrfachvorkommen von Objekten kommen. Die Reihenfolge der Objekte innerhalb der Tabelle spielt keine Rolle. In der Datenvorverarbeitung sollen die vorliegenden Daten auf den jeweiligen Lernprozess - dies kann z.B. ein Clusterverfahren sein - angepasst werden. Nach der Datenvorverarbeitung wird der eigentliche Lernprozess durchgeführt. Zur Auswertung des Ergebnis des Lernprozesses existieren verschiedene Verfahren, die versuchen, die Qualität des Resultats zu beurteilen. Eine angemessene Visualisierung der gewonnenen Erkenntnisse sollen diese dem Benutzer leichter zugänglich machen. Die Anwendung der einzelnen Verfahren sind - mit Ausnahme der Lernprozesse, die die eigentliche Aufgabe der Informationsgewinnung durchführen - optional. Es können jedoch in einer Data Mining Analyse auch mehrere Verfahren der Datenvorverarbeitung, der Auswertung oder der Visualisierung zum Einsatz kommen. Eine Data Mining Analyse kann also nicht als *eine* elementare Methode angesehen werden, sondern setzt sich aus mehreren Schritten verschiedener Verfahren zusammen.

Ein konkretes Beispiel einer Data Mining Analyse ist eine Sequenz aus einer Ähnlichkeitfunktion (Verfahren der Datenvorverarbeitung), die aus der Datentabelle eine Ähnlichkeitsmatrix berechnet, dem KMedoids Clusterverfahren (Lernprozess) und einer Dichtefunktion (Auswertung), die die durchschnittliche Dichte der Objekte innerhalb der entstehenden Cluster berechnet.

`Yale` versucht durch verschiedene Konzepte eine effiziente und flexible Durchführung von Data Mining Analysen zu unterstützen.

5.2.1 Operatoren und Operatorketten

Operatoren und Operatorketten spielen in `Yale` eine zentrale Rolle. Im Folgenden wird schrittweise an das Operatorenkonzept herangeführt, vom Zusammenhang zwischen `Yale`-Operatoren und Verfahren des Data Mining, der Beschreibung eines Operators und dessen konkrete Umsetzung bis zur Zusammensetzung von Operatoren zu komplexen Operatorketten, um Data Mining Analysen durchführen zu können.

[3] Wird auch als „Maschinelles Lernen“ (engl. machine learning) bezeichnet und ist ein Oberbegriff für die maschinenunterstützte Generierung unbekannter Informationen aus Daten (siehe [Mit97]).

Operatoren

Im vorangehenden Abschnitt wurde bereits erwähnt, dass eine Data Mining Analyse sich als Sequenz bzw. Kombination verschiedener Verfahren beschreiben lässt. In `Yale` wird jedes solche Verfahren als Operator angesehen, unabhängig davon welcher Gruppe es angehört. Eine Aneinanderreihung mehrerer Operatoren zu einer Sequenz bildet eine Operatorkette und ist vergleichbar mit einer Data Mining Analyse, die sich ebenfalls aus mehreren Verfahren zusammensetzt.

Ein Operator in `Yale` erwartet einen Input, führt einen Algorithmus aus und liefert einen Output. Einige Operatoren erwarten zusätzlich die Angabe von Parametern durch den Benutzer, die die Abarbeitung des Algorithmus beeinflussen.

Abbildung 5.1 (A) zeigt den Aufbau eines abstrakten Operators in `Yale` als Black Box, die innere Funktionsweise hängt von der Art des Operators ab. Dieser Aufbau ist allen Operatoren in `Yale` gemein. Abbildung 5.1 (B) zeigt die Aneinanderreihung von Operatoren zu einer Operatorkette. Die Operatoren tauschen Input- / Outputobjekte aus. Von den verschiedenen Typen von Objekten bzw. Daten, die Operatoren verarbeiten und austauschen wird hier zunächst abstrahiert und lediglich von *IOObject*, einem Interface das alle Input- / Outputobjekte in `Yale` implementieren, als Abstraktion von zugrunde liegenden Typen gesprochen. Im folgenden Kapitel 5.2.2 wird dieser Begriff konkretisiert. Operatoren tauschen jedoch niemals einzelne *IOObject*s aus, sondern einen *IOContainer*, der *IOObject*s in einer geordneten Liste kapselt. Auf die Darstellung von Parametern der Operatoren in Abbildungen von Operatorketten wird im Folgenden, zur besseren Übersicht, verzichtet. Parameter werden einem Operator direkt zugewiesen und sind keine *IOObject*s.

Die Architektur von `Yale` folgt dem Pipes and Filters (siehe [BMR+96]) Architekturmuster. Dieses Muster beschreibt die Struktur von Systemen, die Datenströme verarbeiten und aus den Elementen Filter und Pipe bestehen. Filter verfügen über einen Dateneingang und einen Datenausgang und wandeln in einem Verarbeitungsschritt die Daten um. Pipes stellen die Verbindung zwischen zwei Filtern dar und repräsentieren den Datenfluss. Der Ausgang eines Filters ist gleichzeitig der Eingang des nachfolgenden Filters. Das Ergebnis des Systems ist linear abhängig von der Eingabe. Abbildung 5.1 (B) zeigt den Aufbau eines Pipe and Filter Systems am Beispiel `Yale`, wobei die Operatoren die Aufgabe der Filter und die *IOContainer* aus *IOObject*s die Aufgabe der Pipes übernehmen.

Ein Operator kann mehrere verschiedene Typen von *IOObject*s im *IOContainer* als Input erwarten und genauso mehrere verschiedene Typen von *IOObject*s erzeugen und als Output bereitstellen. Ein Operator benötigt zur Ausführung seines Algorithmus ein oder mehrere festgelegte Typen von *IOObject*s als Input. Ist ein notwendiger Typ nicht im *IOContainer* des Input enthalten, kann der Operator den Algorithmus nicht ausführen und bricht ab. Sind im Input *IOObject*s enthalten, die der Operator nicht benötigt, werden diese ignoriert und als Output weitergegeben. Ob ein Operator ein *IOObject*, das er zur Berechnung benötigt, konsumiert oder als Output weiterreicht (verändert oder unverändert), hängt von der Implementierung des Operators ab.

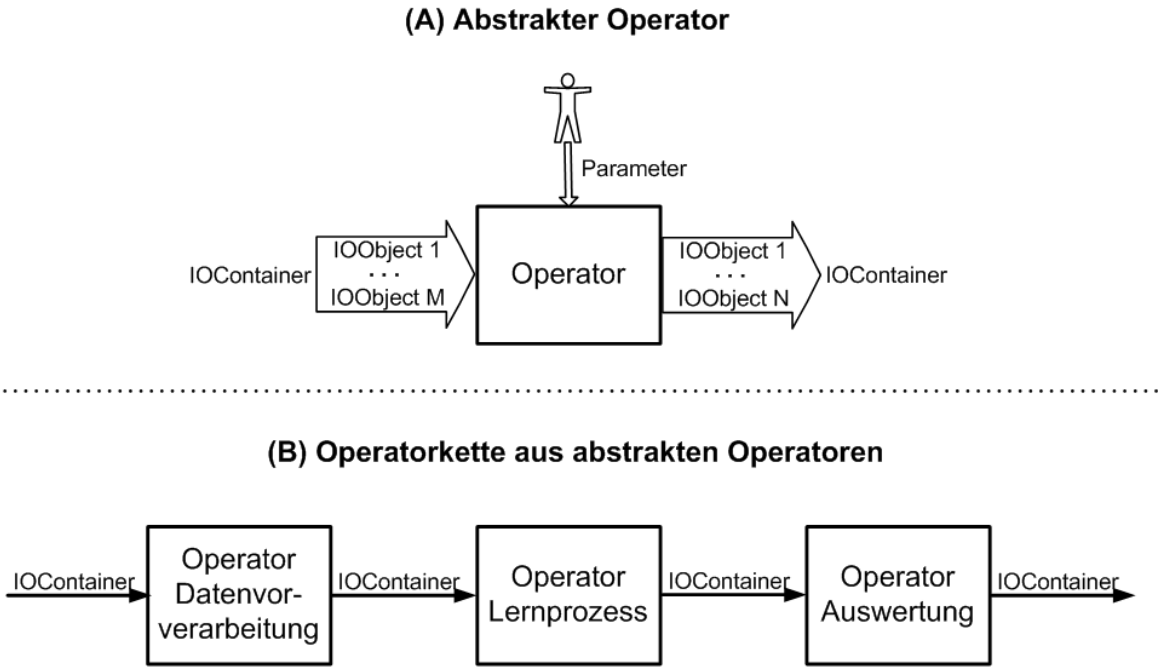


Abbildung 5.1: Aufbau eines abstrakten Operator in `Yale` (A) und eine Operatorkette aus abstrakten Operatoren (B).

Umsetzung

Jeder Operator ist in `Yale` als eigene Klasse realisiert. Ein Operator stellt eine Komponente dar, die mit ihrer Umwelt über Inputschnittstellen in Form von *IOObject*s und Parametern und über Outputschnittstellen in Form von *IOObject*s kommuniziert. Da diese Eigenschaften allen Operatoren in `Yale` gemein sind, werden die Operatoren in einer Hierarchie - im objekt-orientierten Sinne - angeordnet, wodurch sich eine Klassenhierarchie mit Vererbungsstrukturen ergibt.

Abbildung 5.2 zeigt einen Ausschnitt aus dieser Klassenhierarchie als UML-Klassendiagramm. Die oberste Klasse in der Klassenhierarchie ist die abstrakte Klasse *Operator*, von der alle Operatoren abgeleitet sind. Als Subklassen von *Operator* finden sich einige weitere abstrakte Klassen, die *Operator* spezifische Eigenschaften und Fähigkeiten hinzufügen und somit Ausgangspunkt für Operatoren spezieller Gruppen bilden. Als Beispiel sei die abstrakte Klasse *FeatureFilter* genannt, die Superklasse für einige Operatoren ist, die Datenvorverarbeitungsverfahren implementieren. Die abstrakte Klasse *Clustering* ist Superklasse aller Operatoren, die Clusterverfahren implementieren und die abstrakte Klasse *AbstractLearner* Superklasse vieler Lernprozesse. Die abstrakte Klasse *OperatorChain* ist Ausgangspunkt für spezielle Operatoren, die in diesem Kapitel unter dem Punkt „Operatorketten“ vorgestellt werden. Operatoren, die keine zusätzlich zu denen in *Operator* im-

plementierten Eigenschaften und Fähigkeiten benötigen, sind direkt von dieser Klasse abgeleitet. So z.B. die Klasse *ExampleSet2Similarity*, die eine Ähnlichkeitsmatrix aus einer Datentabelle berechnet.

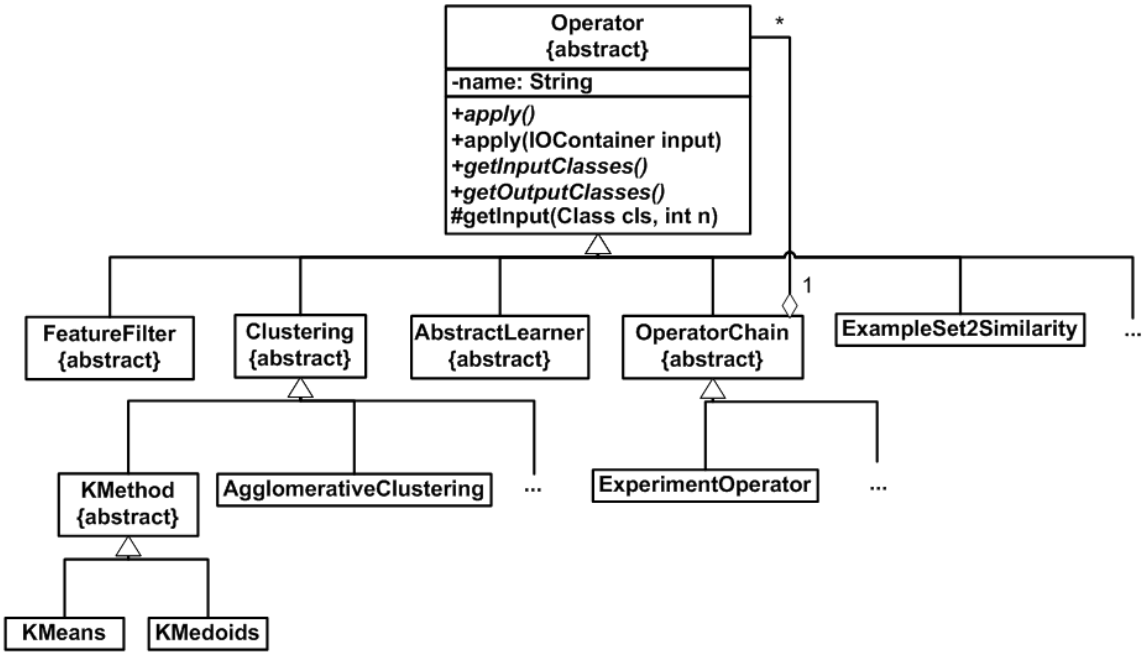


Abbildung 5.2: Ausschnitt aus der Klassenhierarchie der Operatoren als UML-Klassendiagramm

Die Klasse *Operator* definiert drei abstrakte Methoden, die von jeder konkreten Subklasse implementiert werden müssen. Die Signaturen der Methoden lauten:

```
public abstract IOObject[] apply();
public abstract Class[] getInputClasses();
public abstract Class[] getOutputClasses();
```

Die Methode *apply()* implementiert die eigentliche Funktionalität, also den Algorithmus des Operators und gibt einen Array von *IOObject*s zurück. Die eigentliche Aufruf-Methode über die ein Operator aktiviert wird, ist die thread-sichere Methode

```
public synchronized final IOContainer apply(IOContainer input);
```

die als Parameter einen *IOContainer* übergeben bekommt. Diese Methode ist also für den Austausch von *IOObject*s mit anderen Operatoren verantwortlich und bildet somit die wesentliche (Aufruf-)Schnittstelle eines Operators. Die Methode ist als final deklariert und darf

somit nicht überschrieben werden. Aus dieser Methode wird die Methode apply() des aktuellen Operators - hier greift das Konzept der Polymorphie (spätes Binden) - zur Abarbeitung des Algorithmus aufgerufen[4]. Jeder Operator berechnet also die Funktion *IOContainer* → *IOContainer*.

Die Methoden *getInputClasses()* und *getOutputClasses()* definieren die Anzahl und Typen - im Sinne von Java-Typen (Class) - von *IOObject*s, die der Operator als Input erwartet bzw. als Output erstellt. Dies ermöglicht unter anderem die automatische Überprüfung, ob alle *IOObject*s, die Operatoren einer Operatorkette als Input erwarten, auch zur Verfügung stehen, bevor die Operatorkette ausgeführt wird.

Die Methode

public List<ParameterType> getParameterTypes();

erlaubt das Deklarieren von obligatorischen und optionalen Parametern für den jeweiligen Operator. Diese Methode ist nicht abstrakt, da nicht jeder Operator Parameter besitzt und diese Methode somit nicht implementieren muss. Die abstrakte Klasse *ParameterType* ist eine Abstraktion spezieller Typen von Parametern. Sie ist Oberklasse von bspw. *ParameterTypeInt* oder *ParameterTypeString*, die herkömmliche Java-Typen, wie Integer oder String, kapseln. `Yale` nutzt eigene Typen für Parameter, um nur eine definierte Menge von Parametertypen zuzulassen. Parameter werden dem jeweiligen Operator direkt zugeordnet, sind also strikt von *IOObject*s zu trennen, da Parameter nicht zwischen Operatoren weitergereicht werden.

Die intern genutzte Methode

protected IOObject getInput(Class cls, int n);

erlaubt jedem Operator, sich aus der Menge der ihm vom vorangehenden Operator zur Verfügung gestellten *IOObject*s, die für seine Berechnungen notwendigen Typen auszuwählen. Enthält der *IOContainer* mehrere *IOObject*s vom selben Typ, kann über einen Index *n* das n-te Vorkommen des Typs aus der geordneten Liste der *IOObject*s des *IOContainer* ausgelesen werden. *IOObject*s der Inputmenge, die vom Operator nicht benötigt werden, werden als Output weitergereicht.

Die Klasse *Operator* implementiert noch weitere Methoden (zur Fehlerbehandlung, Logging, etc.), die hier genannten bilden jedoch das Grundgerüst eines jeden Operators und reichen aus, einfache Operatoren zu erstellen. Insbesondere stellen die genannten Public-Methoden im wesentlichen die Schnittstelle eines Operators dar.

Die neu gewonnenen Erkenntnisse werden in die Abbildung 5.1 (B) eingebaut und diese um konkrete Operatoren erweitert. Abbildung 5.3 zeigt eine beispielhafte Data Mining

[4] Die in diesem Abschnitt beschriebene Vorgehensweise ist angelehnt an das Command-Pattern (siehe [Gam97]).

Analyse wobei *ExampleSet2Similarity* eine Ähnlichkeitsmatrix berechnet, *KMedoids* eine partitionierende Clusteranalyse durchführt und der Operator *ClusterDensityEvaluater* die durchschnittliche Dichte innerhalb der entstehenden Cluster bewertet. Die Sicht auf die ausgetauschten Daten bleibt hier abstrakt.

Abbildung 5.3: Operatorkette aus konkreten Operatoren

Operatorketten

Im vorangehenden Abschnitt wurde deutlich, dass es sich beim Operatorenkonzept um eine komponentenbasierte Architektur (siehe [CH01]) handelt. Operatoren stellen Komponenten dar, die untereinander verbunden werden können und über Schnittstellen miteinander kommunizieren, d.h. konkret: *IOContainer* austauschen. Die Möglichkeiten und die Umsetzung Operatoren zu verbinden werden in diesem Abschnitt aufgezeigt.

Bisher wurde immer von einem sequentiellen Ablauf einer Data Mining Analyse mit flachen linearen Operatorketten ausgegangen (siehe Abbildung 5.3). `Yale` ermöglicht zusätzlich verschachtelte Operatoren, d.h. es existieren Operatoren, die innere Operatoren bzw. eine innere Operatorkette enthalten können. Operatoren, die diese Fähigkeit besitzen, müssen Subklasse der abstrakten Klasse *OperatorChain*[5] sein, die selbst Subklasse von *Operator* ist (siehe Abbildung 5.2). In der Abbildung wird deutlich, dass *OperatorChain* Objekte vom Typ *Operator* enthalten kann, welche entweder einfache Operatoren oder wieder Verschachtelungsoperatoren sein können. Dieser Ansatz folgt dem bekannten Composite-Pattern (siehe [Gam97]), das zur Beschreibung von hierarchisch strukturierten Typen angewandt wird und in seiner UML-Beschreibung die Struktur der Klassen Kompositum, Komponente und Blatt darstellt. In der `Yale`-Struktur ist die Klasse *OperatorChain* das Kompositum, *Operator* die Komponente und einfache Operatoren die Blätter.

Die Objekte dieser Klassen spannen einen Baum auf, dessen Wurzel ein Objekt des Typs Kompositum, also in der `Yale`-Struktur der Klasse *OperatorChain*, ist. Der Wurzelknoten des Baumes ist in `Yale` immer der Operator *ExperimentOperator*, eine konkrete Subklasse von *OperatorChain* (siehe Abbildung 5.2). Jeder Blattknoten ist ein einfacher Operator und jeder innere Knoten ein Verschachtelungsoperator.

[5] Da der Begriff „operator chain“ (zu deutsch: Operatorkette) von den `Yale`-Autoren bereits für einfache Sequenzen von Operatoren verwendet wird, werden Operatoren, die innere Operatoren enthalten können, im Folgenden als Verschachtelungsoperatoren (englisch: Nesting Operator) bezeichnet. Für Sequenzen von Operatoren wird - wie bisher - weiterhin der Begriff Operatorkette verwendet.

Abbildung 5.4 zeigt die Darstellung zweier Operatorketten als Baum und den Datenfluss durch diesen. Der Baum aus Abbildung 5.4 (A) repräsentiert die Operatorkette aus Abbildung 5.3. Abbildung 5.4 (B) zeigt einen Baum aus abstrakten Operatoren mit einem Verschachtelungsoperator als inneren Knoten. Die Pfeile zeigen den Fluss des *IOContainer*s, die Nummerierung stellt die Reihenfolge dar, in der der *IOContainer* weitergegeben wird. Jeder Elternknoten führt seine Kindknoten durch Aufrufen der bekannten Methode

public synchronized final IOContainer apply(IOContainer input);

nacheinander aus. Dabei übergibt er ihnen den aktuellen *IOContainer* und erhält als Ergebnis einen neuen *IOContainer* zurück. Dieser wird dem nächsten Kind wieder als Input übergeben. Nachdem der letzte Kindknoten ausgeführt wurde, gibt der Elternknoten den *IOContainer* an seinen Elternknoten bzw. an den Operator, von dem er aufgerufen wurde, zurück. Die Verschachtelungsoperatoren steuern also den Ablauf einer Operatorkette. Außerdem ermöglichen sie zusätzlich zu einem rein sequentiellen Ablauf einer Operatorkette ein wiederholtes Ausführen einer inneren Operatorkette und damit die Schleife als Kontrollstruktur[6]. Verzweigungen oder Sprünge sind nicht möglich.

Die Baumdarstellung einer Operatorkette wird in `Yale` auch als Experiment - im Sinne eines Data Mining Experimentes - bezeichnet. Ein Experiment wird in einem XML-Format gespeichert, wobei Operatoren als XML-Element dargestellt werden. Diese können Elemente vom Typ Parameter umgeben. Beide Elementtypen können über Attribute spezifiziert werden.

Die Klasse *Experiment* lädt die XML-Darstellung eines Experimentes und erzeugt Instanzen der beteiligten Operatoren. Anschließend startet die Klasse das Experiment durch Anstoßen des ersten Operators *ExperimentOperator* entweder mit einem gefüllten oder einem leeren *IOContainer* (Pfeil 1 in Abbildung 5.4 (A) und (B)). Nach Abschluss des Experiments erhält die Klasse *Experiment* die finale Liste von *IOObject*s in einem *IOContainer* zurück (Pfeil 8 bzw. 10 in Abbildung 5.4 (A) bzw. (B)).

Der gesamte Funktionalitätsumfang von `Yale`, der Data Mining Analyse betrifft, ist als Operatoren realisiert. `Yale` stellt insgesamt über 350 Operatoren zum Erstellen von Data Mining Analysen zur Verfügung. Diese könnnen durch das vorgestellte Operatorenkonzept in vielfältiger Weise miteinander kombiniert werden. Bedingung für das Zusammensetzen von Operatoren zu funktionierenden Experimenten ist, dass jedem Operator der von ihm erwartete Input zur Verfügung gestellt wird. In [MFK06] findet man eine Referenz der `Yale`-Operatoren. Zur besseren Übersicht sind die Operatoren, je nach ihrer Tätigkeit, in Gruppen eingeteilt. Die wichtigsten Gruppen sind Input / Output, Datenvorverarbeitung, Lernprozesse, Auswertung und Visualisierung. Die Verfahren der Clusteranalyse sind einer eigenen Gruppe namens Clustering zugeordnet.

[6] Das in diesem Kapitel vorgestellte Pipes and Filters Architekturmuster verlangt eine sequentielle Anordnung der Filter (Operatoren bzw. Kompositum von Operatoren in `Yale`). Durch das vorgestellte Konzept der Verschachtelungsoperatoren ist diese nicht mehr unbedingt gegeben. Das „Tee and Join Pipeline System“ ([BMR+96]) stellt eine Variante des „Pipes and Filters“ Architekturmusters dar und erlaubt Schleifen im Datenfluss.

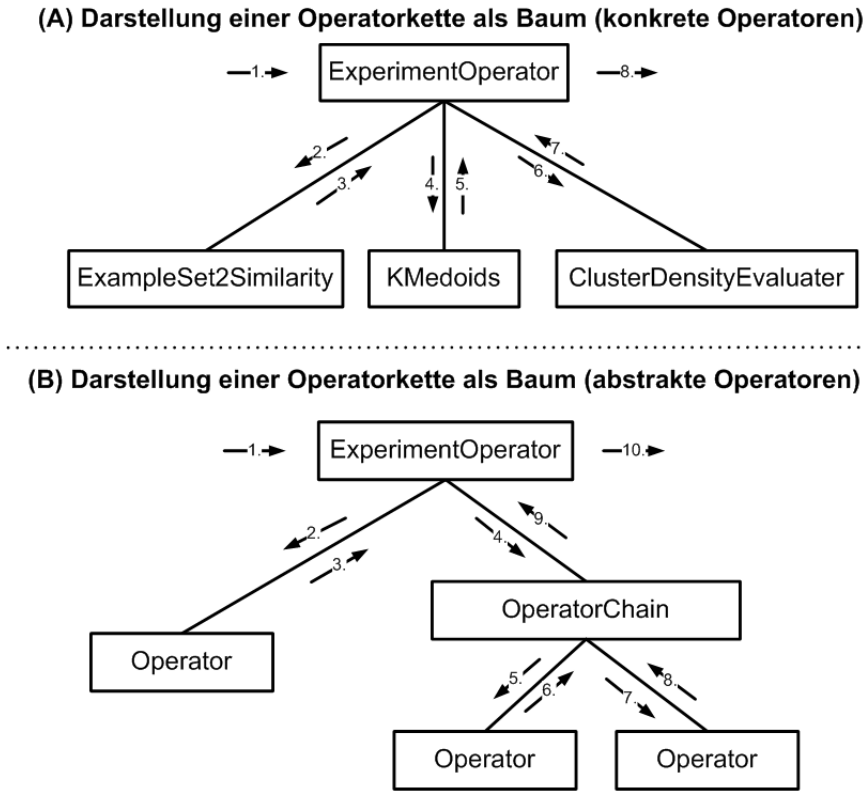


Abbildung 5.4: Darstellung einer Operatorkette als Baum

5.2.2 Datenhaltung

Bisher wurde von den Datentypen, die zwischen Operatoren ausgetauscht werden, abstrahiert und verallgemeinernd von *IOObject*s gesprochen. Dieser Begriff wird im Folgenden konkretisiert. Zunächst werden grundlegende Konzepte der Datenhaltung in `Yale` vorgestellt und die wichtigsten Datentypen aufgezeigt. Ausgangspunkt einer jeden Data Mining Analyse sind Datentabellen, die in `Yale` importiert werden müssen. Im Laufe einer Data Mining Analyse wird die ursprünglich eingelesene Datentabelle von unterschiedlichen Operatoren in `Yale` transformiert und / oder neue Daten aus ihr generiert. Dadurch entstehen neue Datentypen, die zwischen Operatoren ausgetauscht oder als Ergebnis einer Analyse exportiert werden können. Angelehnt an diesen typischen Ablauf einer Data Mining Analyse wird in diesem Kapitel die Datenhaltung in `Yale` vorgestellt. Abschließend wird die Darstellung der Operatorkette aus Abbildung 5.3 um konkrete Datentypen erweitert.

Ähnlich wie die Operatoren sind auch die Datentypen in einer Hierarchie angeordnet (siehe Abbildung 5.5). Das Interface *IOObject* bildet dabei den Ausgangspunkt dieser Hierarchie und muss von jeder Klasse, die einen zwischen Operatoren austauschbaren Datentyp dar-

stellt, implementiert werden (ermöglicht Polymorphie). Die Klasse *IOContainer* ist eine Containerklasse zum Verwalten einer geordneten Liste von *IOObject*s.

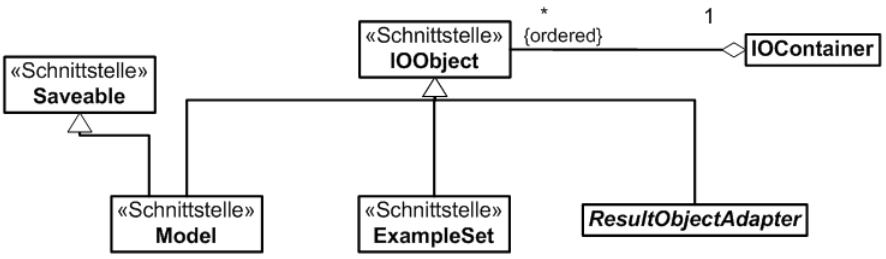


Abbildung 5.5: Datentypen in `Yale` als UML-Klassendiagramm

Zusätzlich zu der Fähigkeit aller *IOObject*s über *IOContainer* zwischen Operatoren austauschbar zu sein, verfügen sie über Methoden, die für die Visualisierbarkeit der Objekte sorgen.

Wie aus Abbildung 5.5 ersichtlich wird, wird *IOObject* von den Interfaces *Model* und *ExampleSet* erweitert und von der abstrakten Klasse *ResultObjectAdapter* implementiert. Dies sind die drei Basistypen, die allen zwischen Operatoren austauschbaren Datentypen in `Yale` zugrunde liegen.

Das Interface *ExampleSet* wird von allen Klassen implementiert, die Daten der ursprünglich eingelesenen Datentabelle referenzieren. Wie diese Datentabellen in `Yale` importiert werden und wie `Yale` diese Daten intern behandelt, wird in diesem Kapitel in einem gesonderten Abschnitt beschrieben.

Das Interface *Model* erweitert zusätzlich zu *IOObject* das Interface *Saveable*. *Saveable* ermöglicht allen Klassen die es implementieren, ihre Daten in einer Datei zu speichern und somit aus `Yale` zu exportieren. Ein *Model* ist das Ergebnis eines Lernprozesses - also einer Transformation einer Datentabelle in einen neuen Datentyp. Wie ein konkretes *Model* letztendlich aussieht, hängt von der Funktionalität des Operators ab, der das *Model* erzeugt. Die einzigen Methoden, die das Interface definiert, sind solche, die das Exportieren des *Model*s in eine Datei, in ein menschenlesbares Format, ermöglichen sollen.

Alle Datentypen, die kein *Model* und kein *ExampleSet* sind, erweitern nur die Klasse *ResultObjectAdapter*. Daten dieses Typs können nicht exportiert werden. Es handelt sich bei diesen Datentypen meist um Zwischenergebnisse die zwischen Operatoren ausgetauscht werden oder um Ergebnisse kleiner Berechnungen, die visualisiert werden können. Als Beispiel sei hier die Klasse *PerformanceVector* genannt, die Ergebnisse einer Auswertung eines *Model*s (also einer Berechnung eines Lernprozesses) speichert.

Wie bereits kurz erwähnt, wird die einer Data Mining Analyse zugrunde liegende Datentabelle in `Yale` intern durch ein *ExampleSet* repräsentiert. Ein *ExampleSet* stellt jedoch

lediglich eine Sicht (view) auf die jeweilige Datentabelle dar. Nach dem Einlesen der Datentabelle wandelt `Yale` diese in eine interne Speicherstruktur um. Die Daten werden intern in einer Tabelle durch die Klasse *ExampleTable* gespeichert und verwaltet. Um Daten für eine Data Mining Analyse angemessen beschreiben zu können, stellt die Klasse *ExampleTable* eine Liste von Attributen (Spalten der Tabelle) und eine Liste von Objekten (Zeilen der Tabelle) zur Verfügung. Die Attribute werden von Instanzen des Typs *Attribute* verwaltet, die Metainformationen - wie Name, Attributtyp, Datentyp - zu den Attributen speichern. Die Objekte, und damit die eigentlichen Daten, liegen in Form von Instanzen des Typs *DataRow* vor.

Ein *ExampleSet* beinhaltet selbst keine Daten, stellt jedoch Methoden bereit, um über die Attribute und Objekte der *ExampleTable* zu iterieren (Iterator-Pattern). Ein *ExampleSet* kann dabei eventuell nur eine eingeschränkte Sicht auf die Tabelle repräsentieren, z.B. auf eine begrenzte Anzahl an Attributen oder Objekten. Zu einem Zeitpunkt können mehrere Sichten auf die Tabelle existieren. Vorteil dieses Konzeptes ist, dass die Tabelle nicht jedes Mal kopiert werden muss, sondern lediglich die Referenz einer bestimmten Sicht auf die Tabelle, also ein *ExampleSet*, zwischen den Operatoren weitergereicht wird. Ein *ExampleSet* ist die Schnittstelle zwischen Data Mining Operatoren und den eigentlichen Daten in der *ExampleTable*.

Das Importieren von Datentabellen geschieht über spezielle Operatoren. `Yale` unterstützt die Dateiformate arff [7], csv [8] und ein eigenes XML-basiertes Format. In diesem Format werden in einer XML-Datei („attribute description file“) Metainformationen der Attribute der Datentabelle beschrieben. Außerdem wird ein relativer Pfad zu einer ASCII-Datei angegeben, welche die eigentlichen Daten speichert. Dem Operator, der die „attribute description file“ einliest, wird per Parameter ein regulärer Ausdruck übergeben, der beschreibt, wie die Datendatei geparst werden soll. Zusätzlich zum Import über eine Datei können Daten über Datenbanken importiert werden (Oracle, MySQL). Für jedes dieser von `Yale` unterstützte Dateiformat bzw. Inputquelle existiert ein eigener Operator. All diese Operatoren erhalten als Parameter die Angabe der Quelle der Daten und transformieren die Daten in die vorgestellte interne Speicherstruktur. Als Output erzeugen sie ein *ExampleSet*, das eine Sicht auf die komplette *ExampleTable* darstellt. Die Quelle der Daten ist damit transparent für die Operatoren.

Das Exportieren von Daten geschieht ebenfalls über Operatoren. Exportiert werden können alle Datentypen die das Interface Saveable implementieren und somit alle *Model*s (also Ergebnisse von Lernprozessen) und *ExampleSets*. Das Speichern der Daten in einer Datei geschieht entweder manuell über die GUI (siehe Kapitel 5.3), wobei beim Anklicken des „Save“-Buttons ein entsprechender Operator angestoßen wird oder automatisch durch Einbauen eines entsprechenden Operators in die Opertorkette. Ein *ModelWriter* speichert bspw. alle aktuellen *Model*s des *IOContainer* in einer über einen Parameter spezifizierten Datei.

[7] Eine Beschreibung des arff (attribute relation file format) Dateiformat findet sich unter http://www.cs.waikato.ac.nz/ ml/weka/arff.html.

[8] Eine csv-Datei (character separated values) dient dem Austausch strukturierter Daten. Die einzelnen Werte werden durch ein beliebiges Zeichen getrennt. Ein allgemeiner Standard für das Dateiformat existiert (bewusst) nicht.

In Abbildung 5.6 wird die Operatorkette aus Abbildung 5.3 aufgegriffen und um einen fehlenden Import-Operator ergänzt. Der Datenfluss zwischen den Operatoren wird mit konkreten *IOObject*s beschrieben. Wie bereits erwähnt, tauschen Operatoren eigentlich nur *IOContainer* aus, hier werden aber zum besseren Verständnis die einzelnen *IOObject*s visualisiert.

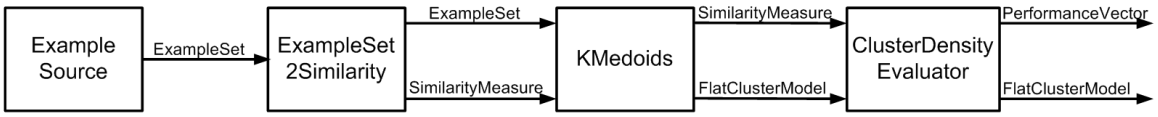


Abbildung 5.6: Darstellung einer Operatorkette

Der Operator *ExampleSource* liest eine Datentabelle (Quelle wird über Parameter angegeben) ein, wandelt diese in die interne Speicherstruktur um und erzeugt ein *ExampleSet*. *ExampleSet2Similarity* erzeugt ein *SimilarityMeasure* (Ähnlichkeitsmatrix) und belässt das *ExampleSet* im *IOContainer*. *KMedoids* berechnet ein *FlatClusterModel* und stellt dieses, zusammen mit dem *SimilarityMeasure* als Output bereit. *ClusterDensityEvaluater* berechnet auf Basis des Inputs einen *PerformanceVector* (durchschnittliche Dichte der Objekte innerhalb der Cluster) und gibt diesen zusammen mit *FlatClusterModel* als Output weiter. Die finalen *IOObject*s im *IOContainer* stehen nun als Output zur Verfügung und können z.B. visualisiert werden.

5.3 Benutzerschnittstelle

Als Benutzerschnittstelle (User Interface) stehen die Ausführung von `Yale` über eine GUI oder über Kommandozeile zur Verfügung.

Die GUI ermöglicht dem Benutzer im wesentlichen zwei Dinge: Das einfache Erstellen, Speichern und Laden eines Experimentes und eine grafische Visualisierung der Ergebnisse einzelner Operatoren.

Mithilfe der GUI kann ein Experiment, bzw. die Experimentbeschreibung im XML-Format, „zusammengeklickt“ werden, indem man Operatoren aus einer Liste wählt und diese in einer Reihenfolge anordnet. Zusätzlich wird zu jedem ausgewählten Operator eine Liste seiner Parameter angezeigt, sodass ein Operator über die GUI konfiguriert werden kann (siehe Abbildung 5.7 (A)). Wurde ein Experiment erstellt, kann über einen Button geprüft werden, ob die Operatoren zueinander passen, d.h. ob jedem Operator sein notwendiger Input zur Verfügung steht. Anschließend kann das Experiment gestartet werden. In einer Konsole wird der Status der Abarbeitung des Experimentes angezeigt. Über einen Parameter des Root-Operators *ExperimentOperator* kann der Benutzer den „verbosity-level“, d.h. den Detailliertheitsgrad der Ausgabe in der Konsole, festlegen. Bei einem hohen Detailliertheitsgrad werden Informationen über interne Berechnungen einiger Operatoren in der Konsole ausgegeben.

Nachdem das Experiment ausgeführt wurde, wechselt die Ansicht automatisch in den Results-Bildschirm (siehe Abbildung 5.7 (B)). Jedem *IOObject* ist eine spezifische Visualisierungsmethode zugeordnet. Gehört ein *IOObject* zum finalen Output des Experimentes, wird es, je nach seiner Visualisierungsmethode, dargestellt. Handelt es sich bei den visualisierten *IOObject*s um solche, die das Interface *Saveable* implementieren, erscheint im zugehörigen Bildschirm ein „Save"-Button, über den die Ergebnisse exportiert werden können.

Ein zusätzliches Feature von `Yale` besteht darin, dass man nach jedem Operator der Operatorkette einen „Breakpoint" setzen kann, bei dem die Ausführung des Experimentes stoppt und die aktuell sich im *IOContainer* befindlichen *IOObject*s visualisiert werden. Die Ausführung des Experimentes kann jederzeit fortgesetzt werden.

Als Alternative zur GUI, besteht die Möglichkeit `Yale` über die Kommandozeile zu starten. Dabei muss entweder eine Experimentbeschreibung in Form einer XML-Datei per Hand erstellt werden oder bereits vorliegen. `Yale` wird dann mit der XML-Datei als Parameter gestartet. Dabei sollten Operatoren im Experiment enthalten sein, die die (Zwischen-)Ergebnisse anderer Operatoren als Ausgabe in einer Datei speichern. Grafische Visualisierungen der Ergebnisse sind nicht möglich.

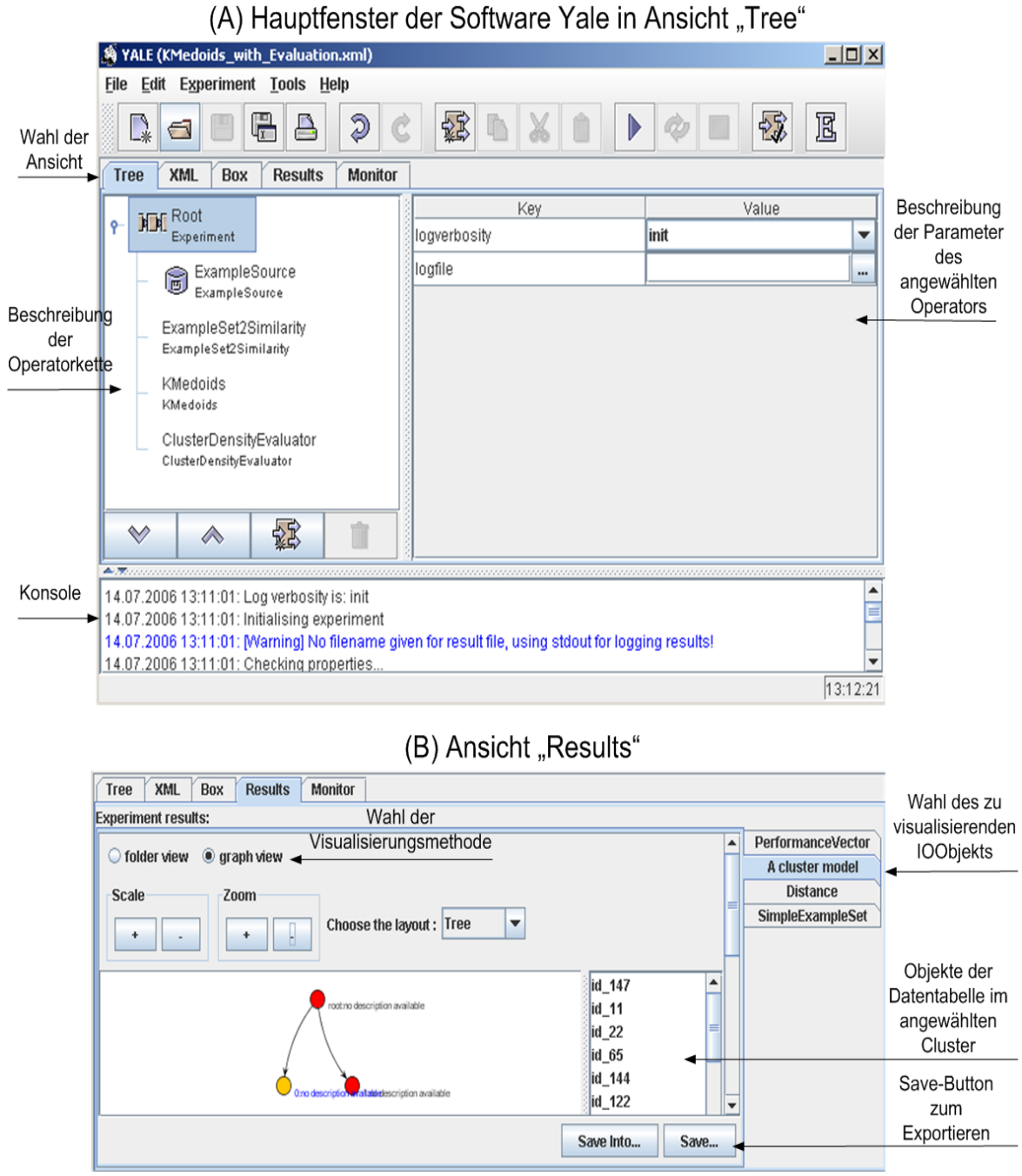


Abbildung 5.7: Experiment aus Abbildung 5.6 im Hauptfenster der Software `Yale` in Ansicht „Tree“ (A) und Visualisierung der Ergebnisse des Experimentes in Ansicht „Results“ (B)

5.4 Programmierschnittstelle

Yale ist eine freie Software, die für Erweiterungen vorgesehen ist. Dafür wurde von den Autoren eine API zur Verfügung gestellt, die die Erweiterung bzw. Anpassung von Yale an eigene Bedürfnisse in drei Richtungen ermöglicht:

- Eine direkte Erweiterung durch Eingreifen in den Quellcode,
- Sammeln von Funktionalität (in Form von Klassen) in einem JAR-Archiv (von den Yale-Autoren als „Plugin“ bezeichnet) und
- Aufrufen von Yale aus externen Programmen.

Diese 3 Möglichkeiten werden im Folgenden vorgestellt.

Direkte Erweiterung

Yale ist eine Plattform, die als Kern das Datenmanagement und das Operatorenkonzept bereitstellt. Aufbauend darauf bietet Yale einige Operatoren an. Da Operatoren die wesentliche Funktionalität von Yale ausmachen, setzt die API zur Erweiterung der Software an den Operatoren an.

Soll ein neuer Operator erstellt werden, muss zunächst überlegt werden, welche bestehende Operatorklasse erweitert werden soll. Handelt es sich um einen Verschachtelungsoperator, muss die Klasse *OperatorChain* oder eine ihrer Subklassen erweitert werden, ansonsten reicht die Klasse *Operator* oder eine ihrer Subklassen (außer *OperatorChain*). Ist diese Entscheidung getroffen, müssen mindestens die drei in Kapitel 5.2.1 (Abschnitt Umsetzung) vorgestellten abstrakten Methoden der Klasse *Operator* implementiert werden. In der Methode *apply()* wird die Funktionalität des neuen Operators implementiert. Besitzt der neue Operator Parameter, muss die Methode *getParameterTypes()* überschrieben werden. Es können noch zusätzliche Funktionalitäten wie gesonderte Fehlerbehandlung oder Konsolenausgaben implementiert werden, darauf soll hier jedoch nicht näher eingegangen werden. Damit Yale den neuen Operator kennt, muss er registriert werden. In der Datei operators.xml befindet sich eine Auflistung aller Operatoren in Yale. Zu jedem Operator muss dessen Name (Anzeige in der GUI) und dessen Quelle (Klassenname als Paketpfad) angegeben sein, optional können eine Beschreibung (Anzeige in der GUI) und ein Icon angegeben werden. Bei jedem Start wird diese Datei gescannt und alle aufgelisteten Operatoren geladen, die dann in Yale zur Verfügung stehen. Da der Quellcode komplett verfügbar ist, kann natürlich auch sonstige Funktionalität von Yale modifiziert und erweitert werden. Dafür steht jedoch keine ausgewiesene Programmierschnittstelle zur Verfügung.

Plugin

Als Plugin wird in Yale eine Sammlung von Operatoren und zugehörigen Klassen in einem JAR-Archiv verstanden. Die Klassen, die Operatoren repräsentieren, werden wie im vorangehenden Abschnitt „Direkte Erweiterung“ beschrieben, implementiert. Die Registrierung der Operatoren wird jedoch in einer separaten Datei operators.xml vorgenommen, die sich im „META-INF“ Ordner des JAR-Archivs befinden muss. Das JAR-Archiv wird nun in die Verzeichnisstruktur von Yale, in den Ordner „plugins“, kopiert. Dieser Ordner wird beim Start gescannt und aus den gefundenen JAR-Archiven die Datei operators.xml eingelesen. Somit sind alle Operatoren des Plugin in Yale verfügbar. Man kann im laufenden Betrieb nicht mehr unterscheiden, welche Operatoren zum Yale-Kern gehören und welche über ein Plugin importiert wurden.

Das Plugin-Konzept in Yale erlaubt nur eine Erweiterung der Software um Operatoren und die von ihnen benötigten Konzepte (wie Parameter, *IOObject*s, ...). Eine Erweiterung sonstiger Funktionalität, wie z.B. des Datenmanagements oder grundlegenden Komponenten der GUI, sind darüber hinaus nicht möglich.

Ein Plugin sollte ausschließlich Operatoren eines Themenkomplexes bzw. einer Gruppe von Verfahren zur Data Mining Analyse enthalten, um dem Benutzer des Plugin mit einem Begriff bzw. einer kurzen Erklärung den Inhalt des Plugin zu verdeutlichen.

Die Operatoren zur Clusteranalyse und die zugehörigen Konzepte sind als Plugin mit dem Namen „Clustering“ realisiert. Dieses kann von der Yale Homepage heruntergeladen werden.

Aufrufen von Yale aus externen Programmen

Yale kann, transparent für den Benutzer, aus externen Java-Programmen aufgerufen werden. Nachdem Yale über den Befehl *Yale.init()* initialisiert wurde, können über die Klasse *Experiment* Experimente durchgeführt werden. Entweder übergibt man der Klasse eine Experimentbeschreibung in Form einer XML-Datei als Parameter oder erstellt ein Experiment über die Operator-Factory-Klasse *OperatorService*. Die Methode *run()* bzw. *run(IOContainer input)* der Klasse *Experiment* startet ein Experiment ohne bzw. mit Input und gibt einen *IOContainer* aus *IOObject*s zurück. Diese können in der eigenen Anwendung weiterverarbeitet werden.

Möchte man Yale aus einer Anwendung nutzen, die nicht in Java realisiert ist, kann man die kommandozeilenbasierte Version von Yale nutzen und als Parameter die Experimentbeschreibung übergeben. Die Ergebnisse liegen dann in Dateien vor, die von Operatoren des Experimentes exportiert wurden.

5.5 Clusteranalyse-Plugin

Die Operatoren zur Clusteranalyse und ihre zugehörigen Konzepte sind in Yale als „Plugin“ realisiert (zur Beschreibung des Plugin-Konzeptes in Yale siehe Kapitel 5.4). Die Konzepte des Plugin fügen sich weitestgehend in die zuvor vorgestellten Konzepte von Yale ein. So sind z.B. alle Operatoren des Plugin (direkte oder indirekte) Subklassen von *Operator* (siehe Abbildung 5.2). Die Ergebnisse von Operatoren zur Clusteranalyse implementieren allerdings nicht das Interface *Model*, sondern das Interface *ClusterModel*, das selbst die Interfaces *IOObject* und *Saveable* implementiert. Ein *ClusterModel* verwaltet das Ergebnis einer Clusteranalyse und stellt Methoden zur Visualisierung und dem Exportieren der Ergebnisse bereit.

Eine Besonderheit aller Operatoren der Clusteranalyse in Yale ist, dass die Datentabelle auf der sie arbeiten, ein Attribut vom Typ *id* beinhalten muss, damit jedes Objekt der Datentabelle eindeutig über diese *id* identifizierbar ist. Dieser Attributtyp kann in Yale in der Klasse *Attribute* verwaltet werden. Verfügen die Daten einer eingelesenen Datentabelle nicht über dieses Attribut, kann es durch einen speziellen Operator namens *IDTagging* automatisch hinzugefügt werden. Dieser Operator erstellt das Attribut und iteriert über die Objekte der Datentabelle um ihnen, beginnend bei dem Wert 1, das ganzzahlige Inkrement des Vorgängerobjektes zuzuordnen.

Clusterverfahren beruhen in ihren Berechnungen auf Distanzen bzw. Ähnlichkeiten. Dabei gilt es zu unterscheiden zwischen Distanzen bzw. Ähnlichkeiten von Objekten und solchen von Clustern.

Der Operator *ExampleSet2Similarity* berechnet Distanzen bzw. Ähnlichkeiten zwischen Objekten aus einer Datentabelle und erstellt eine Distanz- bzw. Ähnlichkeitsmatrix. Über einen Parameter wird diesem Operator die Distanz- / Ähnlichkeitsfunktion übergeben, die er für seine Berechnungen verwendet. Zur Verfügung stehen mit *EuclideanDistance*, *ManhattanDistance* und *CosineSimilarity* lediglich Funktionen, die für quantitative Variablen bestimmt sind. Es werden also alle Variablentypen wie quantitative Variablen behandelt (siehe Kapitel 2.2).

Verfahren zur Berechnung von Distanzen bzw. Ähnlichkeiten zwischen Clustern (Linkage-Kriterien) werden für interne Berechnungen von hierarchischen Clusterverfahren benötigt. Das vom jeweiligen Clusterverfahren zu verwendende Linkage-Kriterium wird über einen Parameter des Operators definiert. Zur Verfügung stehen *SingleLinkage* und *CompleteLinkage*.

Ein *ClusterModel* verwaltet das Ergebnis einer Clusteranalyse in einer Liste aus Instanzen vom Typ *Cluster*. Ein *Cluster* beinhaltet die id´s der Objekte der Datentabelle. Handelt es sich um ein *HierarchicalClusterModel*, sind die *Cluster* als Baum angeordnet, wobei ein Elternkonten seine Kindknoten kennt. Der Wurzelknoten des Baumes beinhaltet alle Objekt id´s. Ein *HierarchicalClusterModel* ist das Ergebnis hierarchischer Clusterverfahren. Alle nicht-hierarchischen Clusterverfahren generieren als Output ein *FlatClusterModel*. In diesem haben die einzelnen Cluster keine Beziehung zueinander.

Zur Visualisierung von *ClusterModel*s stehen zwei Visualisierungsmethoden zur Verfügung, die „folder view“ und die „graph view“ (siehe Abbildung 5.7 (B)).

Im „folder view“ werden Cluster als Ordner dargestellt, die Objekte (*FlatClusterModel*) als Elemente oder zusätzlich weitere Ordner (*HierarchicalClusterModel*) enthalten.

Im „graph view“ liegt das Ergebnis als Baumdarstellung vor, wobei die Knoten ein Cluster repräsentieren, die Objekte enthalten. Die in einem Knoten enthaltenen Objekte (bzw. ihre id´s) werden durch Anklicken des entsprechenden Knoten in einem kleinen Fenster aufgelistet. Bei dieser Darstellung wird auch bei einem *FlatClusterModel* ein Wurzelknoten dargestellt, obwohl dieses Cluster eigentlich nicht zum Ergebnis der Berechnungen des zugrunde liegenden Clusterverfahrens gehört.

6 Die Software GUPRO

Clusteranalysen sollen auf Objekten (Dateien, Variablen, Methoden,...) aus Programmcode und deren Beziehungen (inkludiert, ruft auf, benutzt,....) durchgeführt werden. Diese Objekte müssen zunächst aus dem Programmcode extrahiert und in ein Format konvertiert werden, das von Clusteranalyseverfahren, also in diesem konkreten Fall von Yale, verarbeitet werden kann[1]. Um eine Umgebung zur Clusteranalyse auf Softwareelementen zu vervollständigen, muss also eine geeignete Software gefunden werden, die benutzerdefinierte Softwareelemente aus Programmcode und zugehörigen Softwareartefakten extrahiert und eine Schnittstelle bereitstellt, um diese extrahierten Informationen in einem geeigneten Format auszulesen.

An der Universität Koblenz-Landau wurde im Rahmen des GUPRO-Projektes[2] das gleichnamige Werkzeug GUPRO (Generische Umgebung zum PROgrammverstehen) [EKW98] entwickelt. Die Software ermöglicht die Analyse von Programmcode und stellt die extrahierten Informationen in einem Repository bereit. Zu Beginn dieser Arbeit wurde beschlossen, GUPRO als Ausgangspunkt für Clusteranalysen auf Softwareelementen zu nutzen.

In den folgenden Abschnitten wird GUPRO und die zugehörige Anfragesprache GReQL kurz vorgestellt und eine Bewertung bzgl. der Eignung von GUPRO als Bestandteil einer Umgebung zur Clusteranalyse abgegeben.

6.1 GUPRO

GUPRO verfolgt das Ziel, den Prozess des Programmverstehens im Reverse Engineering zu unterstützen. Vor allem bei Legacy Systems bzw. älterer Software allgemein fehlt es oftmals an Dokumentation und somit ist der Quellcode die einzige Informationsquelle. GUPRO ermöglicht verschiedene Artefakte von Softwaresystemen, wie z.B. Quellcode oder Datenbankdefinitionen, zu analysieren und die extrahierten Informationen in einem Repository persistent zu speichern. Die Informationen liegen dabei als TGraphen (siehe [DEF+98]) vor, die Objekte (Dateien, Klassen, Methoden...) und ihre Beziehungen (inkludiert, ruft auf, benutzt,....) in einem graphenbasierten Format beschreiben und eine Abstraktion von Programmcode darstellen. TGraphen sind typisierte, attributierte, gerichtete und geordnete Graphen. Das Schema der TGraphen im Repository und der Abstraktionsgrad der von ihnen dargestellten Programminformationen hängt von einem sogenannten Konzeptmodell (auch Schema genannt) ab, das vom Benutzer erstellt wird und GUPRO neben dem Quellcode als

[1] Um Verwechselungen mit dem Objekt aus der objektorientierten Programmierung vorzubeugen, werden die Objekte aus Programmcode und deren Beziehungen im Folgenden als Softwareelemente bezeichnet.

[2] http://www.gupro.de

Eingabe dient. Das Konzeptmodell beschreibt in einer EER[3]-ähnlichen Notation, welche Informationen aus dem Programmcode extrahiert werden sollen und bestimmt damit den Parser, der den Quellcode analysiert und die TGraphen erstellt. Das Konzeptmodell gewährleistet somit die generische Anwendbarkeit von GUPRO für unterschiedliche Programmiersprachen und Reengineering-Aufgaben (durch das Ermöglichen verschiedener Sichten auf die Software). Abbildung 6.1 zeigt eine Darstellung des Werkzeugs GUPRO als datenflussartige Architektur.

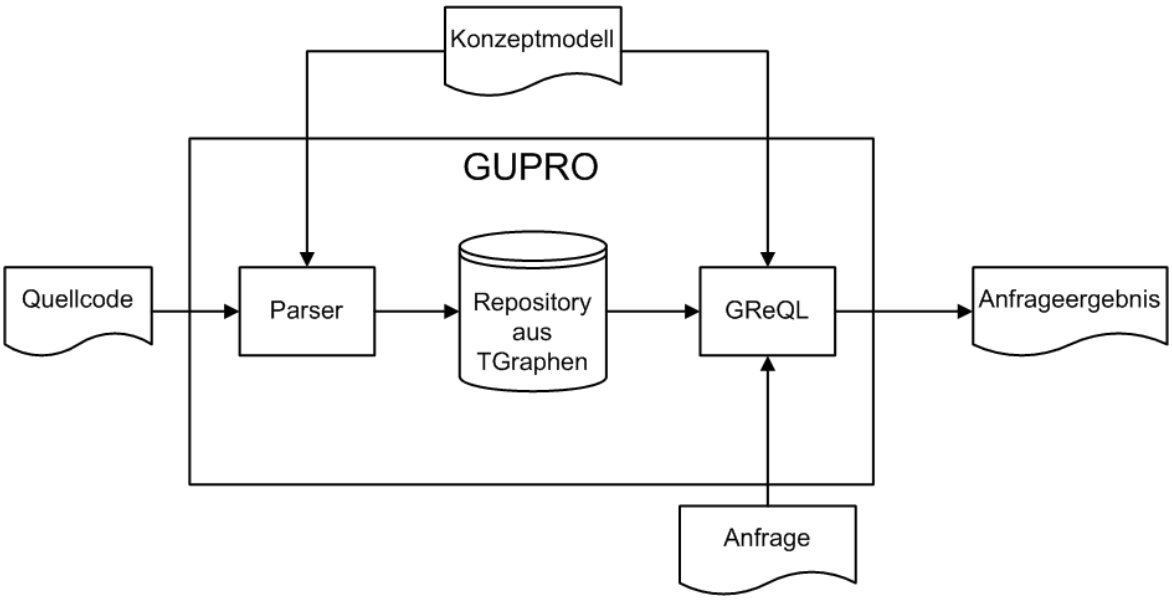


Abbildung 6.1: Architektur von GUPRO

Die TGraphen im Repository sind als Dateien mit der Endung .tg abgelegt. Die Klassenbibliothek JGraLab (Java GRAphenLABor, siehe [Kah06]) bietet Operationen an, um TGraphen zu laden, traversieren, manipulieren und speichern. Die geladenen TGraphen werden von JGraLab als Java-Objekte zur Verfügung gestellt.

6.2 GReQL

Die Sprache GReQL (Graph Repository Query Language) [KK01, Mar06], die ebenfalls im Rahmen des GUPRO-Projektes entwickelt wurde, ist eine deklarative Anfragesprache für TGraphen, die das Traversieren und Analysieren von TGraphen ermöglicht. Innerhalb einer GReQL-Anfrage können Prädikate über Graphen, reguläre Pfadausdrücke und Funktionen auf Graphen formuliert und somit vielfältige Informationen aus dem Repository gewonnen werden. Dabei muss das den TGraphen zugrunde liegende Konzeptmodell bekannt sein, um

[3] Erweiterte Entity-Relationship-Diagramme (EER) schließen Konzepte wie Vererbung und Aggregation mit ein.

zu wissen, welche Informationen aus den TGraphen extrahiert werden können. Als Rückgabe erhält man Daten in Form einer Tabelle, entweder als Menge oder als Bag von Tupeln[4]. Auch verschachtelte Strukturen sind möglich, so kann z.B. eine Tabelle als Element wiederum eine Tabelle enthalten.

GReQL ermöglicht die Extraktion von inhaltlichen, strukturellen und aggregierten Informationen aus TGraphen und damit aus dem GUPRO-Repository. Die Sprache ist - im Gegensatz zu SQL - eine reine Anfragesprache zur Extraktion von Daten aus dem Datenbestand, erlaubt also keine Manipulation wie Löschen, Einfügen oder Ändern der Daten.

Die Semantik der Sprache basiert im Wesentlichen auf der Sprache GRAL (Graph Specification Language, [Fra97]), die die Formulierung komplexer Prädikate über Graphen ermöglicht und einen umfangreichen Satz von Funktionen auf Graphen zur Verfügung stellt. Die Syntax von GReQL ist angelehnt an SQL. Jede Anfrage wird als Ausdruck angesehen, wobei Konstanten, Variablen, Funktionen und so genannte FWR-Ausdrücke (From With Report) als Ausdrücke gelten und Verschachtelungen dieser wiederum einen Ausdruck bilden. Ein FWR-Ausdruck ähnelt dabei dem aus SQL bekannten SFW-Ausdruck (Select From Where). Jede Anfrage richtet sich an genau einen Graphen.

6.3 Eignung von GUPRO für die Clusteranaylse-Umgebung

GUPRO ermöglicht die Extraktion von Programminformationen unterschiedlicher Art und Abstraktionsstufen aus bestehendem Programmcode. Diese können als Grundlage weiterer analytischer Untersuchungen, wie z.B. Clusteranalysen, dienen. Die Anfragesprache GReQL kann als Schnittstelle zwischen GUPRO und Yale fungieren, um die relevanten Programminformationen aus dem von GUPRO generierten Repository auszulesen und Yale zur Weiterverarbeitung zur Verfügung zu stellen.

GUPRO - in Kombination mit der Anfragesprache GReQL - erfüllt alle in Anhang A (Abschnitt A.2) formulierten Anforderungen. Die Software entstand aus einem Gemeinschaftsprojekt der Volksfürsorge Deutsche Lebensversicherungen AG Hamburg, dem Wissenschaftlichen Zentrum Heidelberg und dem Institut für Softwaretechnik der Universität Koblenz-Landau und wurde im praktischen Umfeld bereits erfolgreich eingesetzt und erprobt (siehe [EKW98]).

Ziel von GUPRO ist die Unterstützung des Programmverstehens im Kontext des Reverse Engineering. Durch die Einbettung der Software in eine Umgebung zur Clusteranalyse werden die Möglichkeiten des Programmverstehens um eine zusätzliche Ebene erweitert. Somit sind die Ziele von GUPRO und der zu entwickelnden Clusteranalyse-Umgebung nicht nur vereinbar (neutral), sondern ergänzen sich gegenseitig (komplementär).

[4] In einem Bag sind Mehrfachvorkommen von Tupeln erlaubt, in einer Menge nicht.

7 Entwurf der Clusteranalyse-Umgebung

Das Hauptziel der Arbeit ist die Entwicklung einer Umgebung zur Clusteranalyse auf Softwareelementen. Diese Umgebung soll jedoch nicht von Grund auf neu entwickelt werden, sondern möglichst unter Wiederverwendung bestehender Software(komponenten) entstehen.

In Kapitel 4 wurde `Yale` aus einer Bandbreite existierender Software zur Clusteranalyse ausgewählt und in Kapitel 5 vorgestellt. Als zweite Komponente wurde `GUPRO` und die dazugehörige Anfragesprache `GReQL` in Kapitel 6 beschrieben. Diese beiden Komponenten müssen nun über ihre Schnittstellen gekoppelt und dem Benutzer ein einheitliches User Interface zur Verfügung gestellt werden.

Außerdem sind einige Erweiterungen und Modifikationen der Software `Yale` notwendig, damit diese den Anforderungen entspricht. In Kapitel 7.4 wird über die Notwendigkeit einiger Erweiterungen und Modifikation diskutiert und die Umsetzung derselben vorgestellt.

7.1 Architektur der Clusteranalyse-Umgebung

Abbildung 7.1 zeigt die datenflussartige Architektur der aus `GUPRO` und `Yale` entstehenden Clusteranalyse-Umgebung.

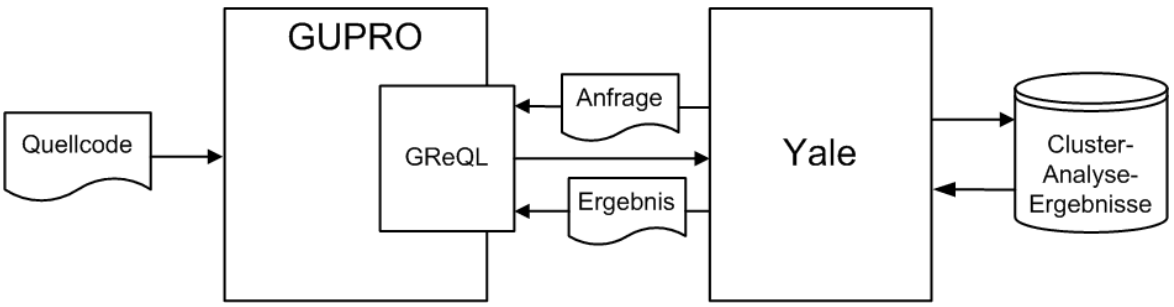


Abbildung 7.1: Architektur der Clusteranalyse-Umgebung

Ausgehend von einem oder mehreren Quellcode-Dokument(en) wird von `GUPRO` der durch ein Schema beschriebene TGraph erstellt und in einem Repository gespeichert. Durch Absetzen einer `GReQL`-Anfrage aus `Yale` (siehe Kapitel 7.2) werden von `GReQL` die in der Anfrage spezifizierten Informationen aus dem TGraphen im Repository extrahiert und die Ergebnisse an `Yale` weitergegeben. Hier können Data Mining Analysen, bzw. im speziellen Fall Clusteranalysen, auf den Daten durchgeführt und deren Ergebnisse exportiert werden.

Die Ergebnisse können von Yale jederzeit wieder geladen werden, um diese zu visualisieren oder weitere Analysen auf den Daten durchzuführen. Außerdem können die Ergebnisse einer Clusteranalyse wieder zurückgeschrieben werden, d.h. das Cluster, in das ein Softwareelement eingeteilt wurde, wird diesem Softwareelement zugeordnet.

Die Anfragesprache GReQL bildet die Schnittstelle zwischen Yale und GUPRO. Bei den Überlegungen zum vorgestellten Architekturentwurf wurde stark auf eine lose Kopplung der beiden Komponenten geachtet (Zerlegungsprinzip, Trennung der Belange). Sowohl GUPRO als auch Yale sollen unbeeinflusst von der Kopplung als eigenständige Software einsetzbar sein. Die konkrete Umsetzung der Kopplung der beiden Komponenten wird in Kapitel 8.1 beschrieben.

Die Universität Koblenz-Landau verwaltet bereits ein GUPRO-Repository, das TGraphen als Ergebnis der Analyse verschiedener Programme enthält. Der Einstiegspunkt der Clusteranalyse-Umgebung kann für den Benutzer also durchaus das Absetzen einer GReQL-Anfrage auf bestehende Programmrepräsentationen in Form von TGraphen sein.

Die vorgestellte Architektur bietet in einigen Punkten durchaus noch Alternativen für die konkrete Umsetzung bzw. bedarf einiger Entwurfsentscheidungen. Dies soll im Folgenden behandelt werden.

7.2 Benutzerschnittstelle

Die vorgestellte Architektur einer Clusteranalyse-Umgebung setzt sich aus zwei Komponenten zusammen (sofern man GReQL als festen Bestandteil von GUPRO ansieht). Dem Benutzer sollen alle Services zum Abwickeln einer Clusteranalyse - vom Einlesen der Daten über das Erstellen und Durchführen einer Clusteranalyse bis zur Visualisierung und Exportieren der Ergebnisse - möglichst unter einer Oberfläche ermöglicht werden. Nun ergeben sich durch die zugrunde liegende Architektur drei Möglichkeiten, die sich als zentrale Benutzerschnittstelle anbieten:

- GUPRO,
- Yale,
- ein zusätzliches Tool, das die Services von GUPRO und Yale unter einer Oberfläche vereint (bspw. als Webinterface).

Aufgrund der Schnittstellen von GUPRO und Yale sind alle genannten Möglichkeiten technisch realisierbar. Im Hinblick auf die Vereinigung aller oben genannten Services unter einer Benutzerschnittstelle finden sich für die Yale-Alternative die vielversprechendsten Vorteile.

Yale stellt im Kern das Datenmanagement und Operatoren zur Durchführung von Data Mining Analysen zur Verfügung. Über die bereitgestellten Schnittstellen können diese Konzepte von externen Programmen genutzt werden.

Die Yale-GUI vereint jedoch zusätzlich bereits zwei der drei genannten Services unter einer Oberfläche, das flexible Erstellen einer Clusteranalyse und das Visualisieren von Ergebnissen. Diese Services müssten bei einer Nutzung von GUPRO oder einem Webinterface als Benutzerschnittstelle neu implementiert werden, was einen nicht zu vernachlässigenden Entwicklungsaufwand bedeuten würde. Nach etwas Einarbeitungszeit und Verständnis der Konzepte von Yale erweist sich die Bedienung der Oberfläche als benutzerfreundlich und ermöglicht vor allem ein flexibles Erstellen von Data Mining Analysen. Die Visualisierung in Yale ist in einigen wenigen Bereichen, vor allem was die Visualisierung hierarchischer Clusteranalyse-Ergebnisse betrifft, verbesserungswürdig. Aufgrund der Erweiterungsmöglichkeiten von Yale können die bestehenden Visualisierungsmöglichkeiten erweitert oder neue erstellt werden (siehe Kapitel 9).

Den noch fehlenden Service, das Einlesen von Daten aus dem GUPRO-Repository, kann über einen neu erstellten Input-Operator (siehe Kapitel 8.1) über die vorhandene Programmierschnittstelle realisiert werden. Dazu wird die bestehende Anfragesprache GReQL genutzt, wodurch eine lose Kopplung von GUPRO und Yale realisiert wird. Ein Eingriff in den Quellcode von GUPRO oder GReQL ist nicht notwendig, die erforderlichen Erweiterungen von Yale finden sich in Kapitel 8.1.

Vor allem im Hinblick auf die zukünftige Nutzung der Clusteranalyse-Umgebung bietet sich Yale als Benutzeroberfläche an. Der Forschungsbereich der Anwendung von Data Mining für Reverse Engineering-Zwecke beschränkt sich aktuell weitestgehend auf Clusteranalysen. Der Autor kann sich jedoch gut vorstellen, weitere Verfahren des Data Mining zur Gewinnung von Informationen aus Programmcode sinnvoll einzusetzen. Da Yale eine Vielzahl von Operatoren für Data Mining Analysen unter einer Oberfläche vereint, werden durch die Nutzung dieser Oberfläche die Möglichkeiten für weitere Forschungen in diesem Gebiet, ohne zusätzlichen Entwicklungsaufwand, offen gehalten.

Die Verwendung der bestehenden und nicht exportierbaren Services der Yale-GUI werden also einer Neuimplementierung derselben vorgezogen.

7.3 Die Schnittstelle GReQL

Da die Graphanfragesprache GReQL als Schnittstelle zwischen GUPRO und Yale fungiert, wird sie in diesem Kapitel detailliert vorgestellt und im Besonderen auf die Schnittstellen von GReQL zur Anbindung an Yale eingegangen.

GReQL besteht neben der Sprachbeschreibung aus einem Interpreter, einer Funktionsbibliothek und einem Wertesystem. Der Interpreter setzt sich wiederum zusammen aus den Komponenten Auswerter, der eine Anfrage umsetzt und Optimierer, der die Struktur einer Anfrage (als Syntaxbaum) für den Auswerter optimiert. Die Funktionsbibliothek stellt Funktionen auf Graphen bereit, die in Anfragen verwendet werden können. Das Wertesystem regelt die Handhabung von Ergebnissen einer Anfrage.

Interessant für die Nutzung von GReQL im Kontext dieser Arbeit sind vor allem der Auswerter (bzw. dessen Schnittstelle), da dieser festlegt welche Angaben zum Ausführen einer Anfrage benötigt werden bzw. möglich sind und das Wertesystem, das definiert wie die Rückgabe einer Anfrage aussieht.

GReQL wurde erstmals vor ca. 10 Jahren konzipiert und seitdem kontinuierlich verbessert und erweitert. Mittlerweile liegt GReQL in zwei Hauptversionen vor:

- GReQL1 (siehe [KK01])
- GReQL2 (Sprachbeschreibung siehe [Mar06], Interpreter und Wertesystem siehe [Bil06])

Hier gilt es zu entscheiden, welche Version für die Clusteranalyse-Umgebung verwendet werden soll.

GReQL2 wurde entwickelt, um seinen Vorgänger um einige Funktionalitäten zu erweitern und um einige Aufgaben, die in GReQL1 nur umständlich zu lösen sind, umgänglicher zu gestalten. Zu den für diese Arbeit wesentlichen Veränderungen zählen die Umgestaltung der zu erstellenden Anfragen in eine benutzerfreundlichere Syntax (siehe [Mar06]) und die Realisierung einer einheitlichen API (siehe [Bil06]). Zudem wurde das Wertesystem von GReQL2 in Java realisiert (zuvor C++), wodurch die Verarbeitung der Rückgabe durch das ebenfalls in Java realisierte Yale stark vereinfacht wird.

Aus den genannten Gründen wird GReQL2 verwendet, und im Folgenden die für die Verwendung der Sprache als Schnittstelle zwischen GUPRO und Yale wichtigsten Konzepte kurz vorgstellt.

Die Aufrufschnittstelle von GReQL2 findet sich in der Auswerter-Komponente. Konkret wird von der Klasse *GreqlEvaluator* eine Instanz erzeugt und dabei dem Konstruktor die Anfrage als String und der TGraph als JGraLab-Graph-Objekt übergeben[1]. Durch Aufrufen der Methode

public boolean startEvaluation();

des erzeugten *GreqlEvaluator*-Objektes wird die Anfrage ausgeführt und über den Rückgabewert angegeben, ob die Ausführung erfolgreich war (*true*). Bei Misserfolg wird eine Java-Exception geworfen, die anzeigt an welcher Stelle der Anfrage ein Fehler vorliegt. Über die Methode

public JValue getEvaluationResults();

erhält man das Ergebnis der Anfrage als *JValue*-Objekt zurück. Der Auswerter berechnet also die Funktion

[1] JGraLab (siehe [Kah06]) ist die Java-Version des Graphenlabors, das die Datenstruktur für TGraphen beschreibt und Algorithmen auf diesen bereitstellt.

String x *Graph* → *JValue*

wobei String für eine Anfrage steht und *Graph* Stellvertreter für ein `JGralab`-Graph-Objekt aus dem `GUPRO`-Repository ist. Auf den Rückgabewert *JValue* wird im Folgenden näher eingegangen.

Die Klassenbibliothek mit dem Namen `JValue` stellt das Wertesystem von `GReQL2` dar. Die TGraphen im `GUPRO`-Repository besitzen unter anderem die Eigenschaft, dass sowohl Knoten (Vertex) als auch Kanten (Edge) attributiert sein können. Der Auswerter von `GReQL2`, der auf TGraphen zugreift und deren Attribute ausliest, verwaltet deren Werte in Objekten vom Typ *JValue*. Abbildung 7.2 zeigt das Klassendiagramm der `JValue`-Architektur. Die genaue Beschreibung der `JValue`-Klassenbibliothek findet sich in [Bil06].

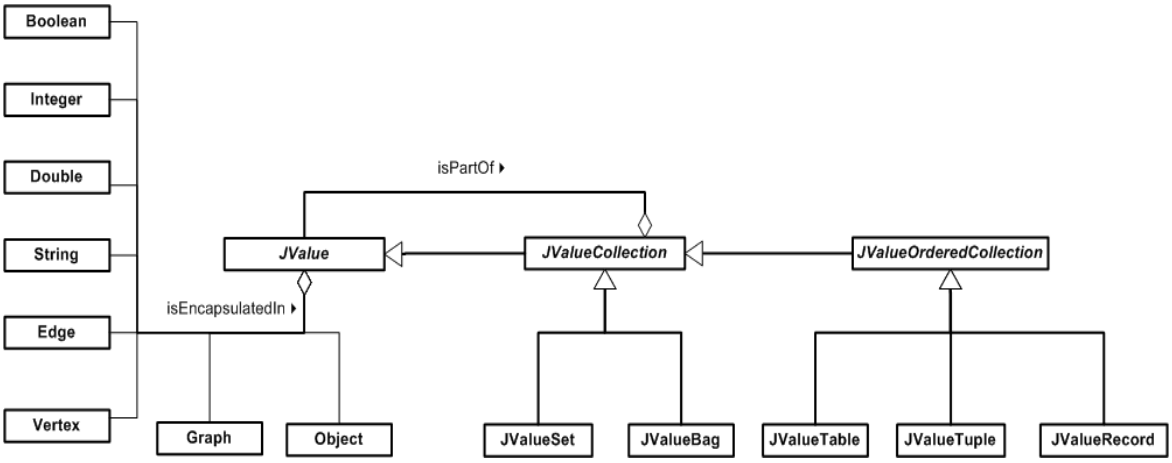


Abbildung 7.2: Klassendiagramm der JValue-Architektur (übernommen aus [Bil06])

Ein Objekt vom Typ *JValue* kann sowohl eine Collection, als auch ein Objekt, das die eigentlichen Daten repräsentiert, darstellen. Über eine Referenz auf ein Objekt der von *JValue* im Klassendiagramm referenzierten Klassen werden die Daten im *JValue*-Objekt gekapselt. Die primitiven Datentypen werden ihrerseits in der jeweiligen Java Wrapper-Klasse (die Java-Standardklassen *Integer*, *Double*,...) gekapselt. Die Java-Klasse *Object* steht für beliebige Objekte. Die `JValue`-Klassenbibliothek unterstützt somit folgende Datentypen:

- Null (kein Element enthalten)
- Boolean
- Integer
- Double

- String
- Object (beliebiges Objekt)

Die Collection-Klassen erben alle von der abstrakten Klasse *JValueCollection*. Eine weitere Spezialisierungsebene eröffnet die abstrakte Klasse *JValueOrderedCollection*, von der alle Collections erben, die eine Ordnung in ihrer Sammlung aufweisen. Eine Collection kann wiederum Objekte vom Typ *JValue* enthalten (Composite-Pattern, siehe [Gam97]), also auch Objekte vom Typ *JValueCollection*. Somit sind beliebig verschachtelte Strukturen möglich.

7.4 Erweiterungen von `Yale` - konzeptionelle Entscheidung

Um `Yale` optimal als zentrale Benutzerschnittstelle einer Clusteranalyse-Umgebung nutzen zu können, bedarf es einiger Erweiterungen und Anpassungen der Software. Welche Erweiterungen und Anpassungen dies im Einzelnen sind, die Notwendigkeit der neu eingeführten Funktionalität und deren Anforderungen werden in folgenden Abschnitten vorgestellt.

Doch zunächst stellt sich die konzeptionelle Frage, wie diese Erweiterungen und Anpassungen vorgenommen werden. In Kapitel 5.4 wurde die Programmierschnittstelle von `Yale` vorgestellt. Insgesamt bieten sich drei Möglichkeiten an, im Einzelnen sind dies

- die direkte Erweiterung von `Yale` durch Eingreifen in den Quellcode,
- das Erstellen eines Plugins oder
- das Aufrufen von `Yale` aus externen Programmen.

Da in Kapitel 7.2 beschlossen wurde, `Yale` als zentrale Benutzerschnittstelle zu nutzen, scheidet die letztgenannte Alternative aus.

`Yale` soll Bestandteil einer Forschungsplattform zur Clusteranalyse auf Softwareelementen sein. Diese Plattform soll auch anderen interessierten Institutionen zugänglich gemacht werden. Die Architektur soll dabei so offen wie möglich gehalten werden. Eine direkte Erweiterung von Yale duch Eingreifen in den Quellcode impliziert das Kompilieren des gesamten `Yale`-Quellcodes, wodurch eine neue Version der Software entsteht. Die selbst vorgenommenen Erweiterungen durch direktes Eingreifen in den Quellcode können somit nur noch schwer oder garnicht in die offiziellen neuen Releases der Software integriert werden. Außerdem wird die Nachvollziehbarkeit der Veränderungen bei direktem Eingreifen in den Quellcode immens erschwert. Durch ein Ausgliedern der Erweiterungen in ein Plugins ist die Integration derselben in neue Releases von *Yale* und eine leichte Nachvollziehbarkeit der Erweiterungen gewährleistet.

Aus den genannten Gründen, und um die Architektur auch für eventuelle zukünftige Erweiterungen über diese Arbeit hinaus offen zu halten, werden die Erweiterungen als *Yale*-Plugin realisiert.

Infolgedessen wird das Plugin-Konzept von `Yale` genauer betrachtet. Wie in Kapitel 5.4 beschrieben, werden alle Klassen eines Plugin in ein Jar-Archiv gepackt, in dem sich eine Datei *operators.xml* befindet. Diese Datei spezifiziert den Pfad der Operatorklassen, die von `Yale` geladen werden. Alle weiteren Klassen des Plugin müssen von den Operatorklassen selbst geladen werden. Somit bilden die Operatorklassen des Plugin die einzige Schnittstelle zu Yale und jegliche Funktionalität des Plugin, wie z.B. neue Visualisierungsmethoden oder Input- / Outputobjekte (Subklassen von *IOObject*), wird von den Operatorklassen des Plugin gesteuert.

8 Das Plugin „GReQL-Interface“

Da Yale die zentrale Benutzerschnittstelle der Clusteranalyse-Umgebung darstellt, die alle Services unter einer Oberfläche vereinen soll, muss eine Anbindung an GUPRO realisiert werden, um die notwendigen Daten zu erhalten. Wie in Abbildung 7.1 bereits dargestellt, wird GReQL als Schnittstelle zwischen Yale und GUPRO genutzt. In Kapitel 7.3 wurde eine Entscheidung für die Verwendung von GReQL2 getroffen und begründet.

Im Folgenden wird die Realisierung der Schnittstelle zwischen Yale und GReQL2 beschrieben. In Kapitel 7.4 wurde beschlossen, Erweiterungen als Plugin zu realisieren. Alle in diesem Kapitel neu eingeführten Konzepte und Klassen werden in einem Plugin mit dem Namen „GReQL-Interface“ zusammengefasst. Abbildung 8.1 zeigt die Integration des Plugin über die Plugin-Schnittstelle in Yale als UML-Komponentendiagramm. Das Plugin „GReQL-Interface“ greift auf Funktionalitäten der Komponenten GReQL2 und JGraLab zu, kann also nur verwendet werden, wenn diese beiden Komponenten vorhanden sind. Zusätzlich wird das bestehende Yale-Plugin „Clustering“ gezeigt, das die Clusteranalyse in Yale ermöglicht. Das Plugin „GReQL-Interface“ verwendet Klassen aus dem Plugin „Clustering“ (siehe Operator *Cluster2JValue*, Abschnitt 8.2). Es besteht also eine Abhängigkeit zu diesem Plugin.

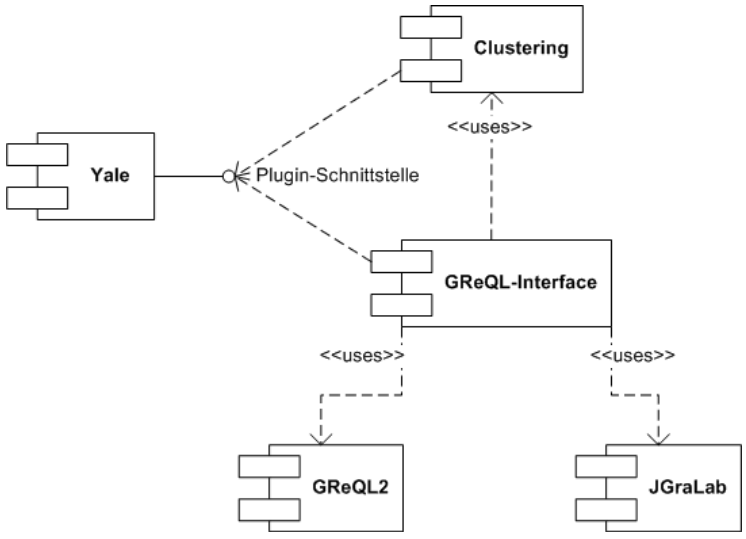


Abbildung 8.1: UML-Komponentendiagramm der Integration des Plugin „GReQL-Interface“ in die Software Yale

8.1 Der Operator *GreqlExampleSource*

Anforderungen

Um `GReQL2`-Anfragen aus Yale heraus absetzen zu können, ist eine Erweiterung der Software notwendig. Bei der neu zu implementierenden Funktionalität handelt es sich um den Import von Daten. Es bietet sich an, dafür einen neuen Operator zu erstellen, der folgende Muss-Anforderungen erfüllt:

1. Laden eines TGraphen, auf dem eine Anfrage ausgeführt wird.
2. Absetzen einer `GReQL2`-Anfrage.
3. Überprüfung, ob es sich beim Ergebnis um verschachtelte Strukturen handelt. Ist dies der Fall, bricht der Operator seine Berechnungen mit einer geeigneten Fehlermeldung ab.
4. Überprüfen, ob ein spezifiziertes Attribut sich als *id*-Attribut eignet, d.h. ob die Werte des spezifizierten Attributes eindeutig sind. Ist dies der Fall, wird das Attribut als *id*-Attribut deklariert, ansonsten bricht der Operator seine Berechnungen mit einer geeigneten Fehlermeldung ab.
5. Umwandeln des Ergebnisses in die interne Speicherstruktur von `Yale`.
6. Bereitstellen der Daten für folgende Operatoren.

Entwurfsentscheidungen

Der zu erstellende Operator - und damit die zugehörige Java-Klasse - wird als *GreqlExampleSource* bezeichnet und gehört zur Gruppe der IO-Operatoren. Diese Namensgebung ist angelehnt an die Bezeichnungen, die von den `Yale`-Autoren für die existierenden Input-Operatoren gewählt wurden. So wird ein Input-Operator als *XyExampleSource* bezeichnet, wobei Xy für die jeweilige Quelle der Daten steht, bspw. *ArffExampleSource* für eine arff-Datei oder *DatabaseExampleSource* für eine Datenbank.

Die Schnittstellen eines Operators in Yale bilden Parameter und *IOObject*s (siehe Kapitel 5.2.1). Über Parameter kann ein Operator konfiguriert werden, ein *IOObject* stellt ein zwischen Operatoren austauschbares Input- / Outputobjekt, also die eigentlichen Daten auf denen die Berechnungen ausgeführt werden, dar.

Da die Inputdaten von *GreqlExampleSource* von einer externen Quelle importiert werden, kann die reguläre Inputschnittstelle des Operators nicht genutzt werden, denn diese lässt nur Yale-interne Objekte vom Typ *IOObject* zu. Somit müssen dem Operator die notwendigen Informationen zum Laden der Daten über Parameter mitgeteilt werden. Einen Input in Form eines *IOObject*s erwartet der Operator nicht.

Oberflächensignatur und Spezifikation

Im Folgenden wird die Oberflächensignatur der Klasse *GreqlExampleSource* als Java-Pseudocode (siehe Code-Listing 8.1) und anschließend eine natürlichsprachliche Spezifikation der Methoden notiert.

```
public class GreqlExampleSource extends Operator{

public IOObject[] apply();
public Class[] getInputClasses();
public Class[] getOutputClasses();
public List<ParameterType> getParameterTypes();
private Graph loadGraph(File graphPath);
private JValue executeQuery(Graph graph, String query);
private boolean checkQueryResultStructure(JValue queryResult);
private ExampleTable convertQueryResult(JValue queryResult);
private ExampleSet generateExampleSet(ExampleTable table);

}
```

Quellcode 8.1: Oberflächensignatur der Klasse *GreqlExampleSource* als Java-Pseudocode

Die aufgelisteten public-Methoden überschreiben die jeweiligen abstrakten Methoden der Oberklasse *Operator* (siehe Kapitel 5.2.1) und müssen von jeder Operatorklasse implementiert werden. Sie stellen die Schnittstelle des Operators dar. Die private-Methoden implementieren die eigentliche Logik des Operators und führen spezifische Berechnungen durch. Im Folgenden wird eine natürlichsprachliche Spezifikation der Methoden aufgeführt:

- *public IOObject[] apply();*

  Dies ist die Aufrufschnittstelle der Klasse. Sie steuert den Ablauf der Berechnungen durch Aufrufen der private-Methoden und gibt das Resultat der Berechnungen als Array aus *IOObject*s zurück.

- *public Class[] getInputClasses();*

  Schnittstelle der Klasse, um Anzahl und Typen der benötigten Inputobjekte zu definieren. Die Typen der benötigten Inputobjekte werden als Array vom Java-Typ *Class* zurückgegeben (muss Subtyp von *IOObject* sein). Dieser Operator erwartet kein Inputobjekt und somit wird ein leerer Array zurückgegeben.

- *public Class[] getOutputClasses();*

  Schnittstelle der Klasse, um Anzahl und Typen der erstellten Outputobjekte zu definieren. Die Typen der erstellten Outputobjekte werden als Array vom Java-Typ *Class*

zurückgegeben (muss Subtyp von *IOObject* sein). Dieser Operator erstellt ein Outputobjekt vom Typ *ExampleSet*.

- *public List<ParameterType> getParameterTypes();*

  Schnittstelle der Klasse, um die benötigten Parameter zu deklarieren und zurückzugeben. Die erstellten Parameter müssen vom Typ *ParameterType*, einer Oberklasse einiger herkömmlichen Java-Typen, sein. Die Parameter werden in einer Liste zurückgegeben.

- *private Graph loadGraph(File graphFile);*

  Interne Methode, um das TGraphen-Objekt vom Typ *Graph* aus der Datei „graphFile“ zu laden, auf dem die Anfrage ausgeführt werden soll.

- *private JValue executeQuery(Graph graph, String query);*

  Interne Methode, um eine *GReQL2*-Anfrage „query“ auf dem *Graph*-Objekt „graph“ abzusetzen und das Anfrageergebnis als Objekt vom Typ *JValue* zurückzuliefern.

- *private boolean checkQueryResultStructure(JValue queryResult);*

  Interne Methode, um die Struktur des Anfrageergebnisses vom Typ *JValue* zu überprüfen. Liegt eine verschachtelte Struktur vor, so wird false zurückgegeben, eine Fehlermeldung erzeugt und die Berechnungen des Operators abgebrochen. Ansonsten wird true zurückgegeben.

- *private ExampleTable convertQueryResult(JValue queryResult);*

  Interne Methode, um das Anfrageergebnis „queryResult“ vom Typ *JValue* in die interne Speicherstruktur von Yale, eine *ExampleTable*, zu konvertieren.

- *private ExampleSet generateExampleSet(ExampleTable table);*

  Interne Methode, um ein *ExampleSet*, eine Sicht auf eine *ExampleTable*, zu erzeugen und zurückzugeben.

Umsetzung

Abbildung 8.2 (A) und (B) zeigt den aus den folgenden Ausführungen resultierenden Aufbau und die Datentypen der Schnittstelle des Operators *GreqlExampleSource*. Wie in Kapitel 5.2.1 bereits erwähnt, werden die Typen der Parameter in `Yale` in Objekte vom Typ *ParameterType* gekapselt. *ParameterTypeString* kapselt den herkömmlichen Java-Typ String, *ParameterTypeFile* den Java-Typ File. Die **fett** geschriebenen Parameter repräsentieren die verpflichtenden, die normal geschriebenen die optionalen Parameter.

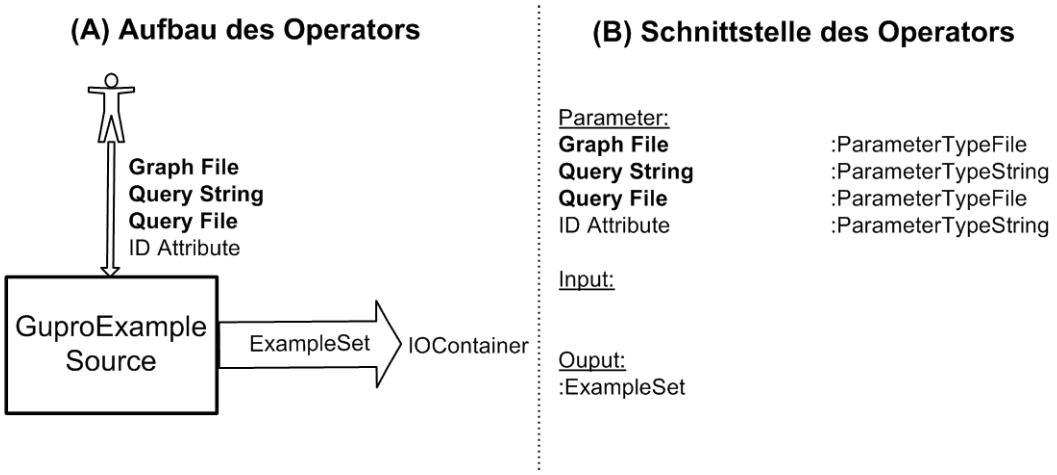


Abbildung 8.2: Aufbau (A) und Schnittstelle (B) des Operators *GuproExampleSource*

Umsetzung der Parameter

Im Folgenden soll die Umsetzung der Parameter näher beschrieben werden:

- „Graph File“
  Zunächst muss dem Operator bekannt sein, auf welchem TGraphen die Anfrage ausgeführt werden soll. Über einen Parameter wird dem Operator der Pfad zur Datei, die den TGraphen speichert, als File-Objekt mitgeteilt. Der Parameter wird „Graph File“ genannt. Die Angabe dieses Parameter ist verpflichtend, da der Operator ohne diese Informationen nicht arbeiten kann.

- „Query String“ und „Query File“
  Weiterhin muss eine `GReQL2`-Anfrage angegeben werden, die die gewünschten Informationen aus dem TGraphen spezifiziert. In der Yale-GUI ist für die Angabe von Parametern vom Typ String ein einzeiliges Textfeld vorgesehen, welches in seiner Länge begrenzt ist. Die Eingabe einer (umfangreichen) `GReQL2`-Anfrage gestaltet sich damit als sehr unübersichtlich. Als Alternative wurde deswegen ein zusätzlicher Parameter erstellt, der als Wert den Pfad zu einer ASCII-Datei erwartet, in der die `GReQL2`-Anfrage spezifiziert ist. Hier werden also zwei Parameter realisiert, die für die Angabe desselben Wertes zuständig sind. Der Parameter „Query String“ erhält die Anfrage als String durch direkte Eingabe im Textfeld der GUI, der Parameter „Query File“ erhält die Anfrage über eine Datei, die in einem Datei-Browser ausgewählt wird. Dabei hat die Angabe des Parameter „Query String“ Priorität, d.h. wurde für „Query String“ ein Wert spezifiziert, wird der Parameter „Query File“ ignoriert. Werden für beide Parameter keine Angaben gemacht, wird ein Fehler ausgegeben, da der Operator nicht ausgeführt werden kann. Hier ist also die Angabe von einem-aus-zwei Parametern verpflichtend.

- „ID Attribute“
  Der Operator *GreqlExampleSource* stellt Daten bereit, auf denen Clusteranalysen durchgeführt werden sollen. Clusteranalyseverfahren (in `Yale`) erwarten in der zugrunde liegenden Datentabelle ein Attribut vom Typ *id*, über dessen Wert jedes Objekt (jede Zeile) der Tabelle eindeutig identifiziert werden kann. Dies ist notwendig, damit die Objekte, nachdem diese in Cluster eingeteilt wurden, über ihre *id* angesprochen und mehrere Objekte mit eventuell gleichen Werten unterschieden werden können. Über den optionalen Parameter „ID Attribute“ kann der Name eines Attributes angegeben werden, das als *id*-Attribut fungieren soll. Es muss geprüft werden, ob die Werte des vom Benutzer angegebenen Attributes eindeutig sind. Ist dies der Fall, wird das Attribut als *id*-Attribut deklariert, ansonsten kann das Attribut nicht als solches verwendet werden. Obwohl der Operator *GreqlExampleSource* in diesem Fall seine Berechnungen fortsetzen könnte, wird hier ein Abbruch erzwungen, da spätestens ein folgender Operator zur Clusteranalyse eine Abbruch-verursachende Ausnahme aufgrund eines fehlenden *id*-Attributes auslösen würde. Wird vom Benutzer kein Wert für diesen Parameter angegeben, wird kein Attribut als *id*-Attribut deklariert. In diesem Fall wird jedoch kein Abbruch erzwungen, da davon ausgegangen wird, dass der Benutzer vor Durchführung einer Clusteranalyse dafür sorgt, dass die Datentabelle mit einem (künstlichen) *id*-Attribut versehen wird[1]. Aus diesem Grund ist dieser Parameter optional.

An dieser Stelle soll festgehalten werden, dass der Operator die Funktion

String x *Graph* → *ExampleSet*

berechnet, wobei der Datentyp String eine `GReQL2`-Anfrage repräsentiert, die der Operator über den Parameter „Query String“ oder „Query File“ (die Anfrage in der angegebenen Datei wird als String ausgelesen) erhält. *Graph* ist Stellvertreter für ein `JGraLab`-Graph-Objekt, das über den in „Graph File“ spezifizierten Pfad geladen wird. Als Ausgabe wird ein Objekt vom `Yale`-Typ *ExampleSet* erstellt.

Nachdem die Schnittstellen und die wichtigsten Methoden der Operatorklasse nun festgelegt sind, zeigt Abbildung 8.3 einen gedachten Durchlauf nach Aufruf des Operators *GreqlExampleSource* als Sequenzdiagramm. Dabei wird die Kommunikation zwischen den Komponenten `Yale`, `JGraLab` und `GReQL2` deutlich.

[1] Der bestehende Operator *IDTagging* erzeugt ein *id*-Attribut mit streng monoton steigenden Werten ($\epsilon\mathbb{N}$), beginnend bei eins.

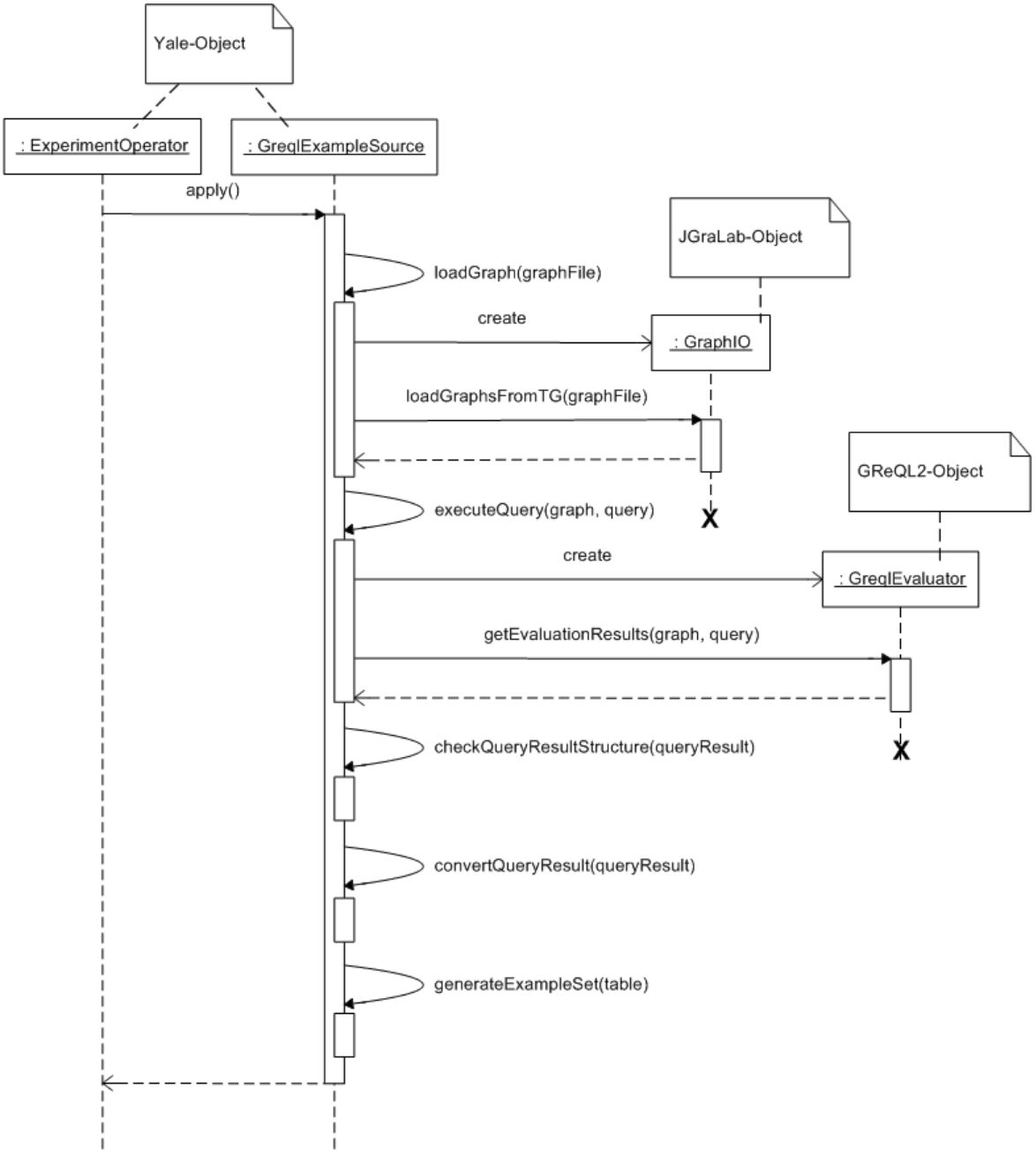


Abbildung 8.3: Sequenzdiagramm der Ausführung einer `GReQL2`-Anfrage aus `Yale`

Umsetzung der Methoden

Im Folgenden soll die Umsetzung einiger Methoden näher beschrieben werden:

- *checkQueryResultStructure(JValue queryResult)*
  Wie in Kapitel 6.2 bereits kurz erwähnt, können die Tupel des Ergebnisses einer `GReQL2`-Anfrage wiederum Tupel als Elemente enthalten. Diese verschachtelten Strukturen können von Clusteranalyseverfahren, und auch von `Yale`, nicht verarbeitet werden. Der Operator muss also dafür sorgen, dass die Tupel der Ergebnistabelle einer Anfrage nur Elemente von einfachen Datentypen (solche die keine weiteren Daten enthalten können) beinhalten. Enthält das Anfrageergebnis verschachtelte Strukturen, wird eine Fehlermeldung ausgegeben.

  Für die Realisierung dieser Anforderung bieten sich zwei Alternativen an: Das Überprüfen des Anfrageergebnisses anhand der spezifizierten Anfrage bevor der Operator ausgeführt wurde oder das Überprüfen des Anfrageergebnisses anhand des erhaltenen Ergebnisobjektes selbst nachdem der Operator ausgeführt wurde. Zwar wäre die zuerst genannte Alternative wünschenswert, doch gestaltet sich das maschinelle Überprüfen der Struktur des Anfrageergebnisses anhand einer `GReQL`-Anfrage als so komplex[2], dass der gewonnene Vorteil den dadurch entstehenden Aufwand nicht rechtfertigt.

  Die Überprüfung der Struktur des Anfrageergebnisses findet also anhand des Ergebnisobjektes selbst statt, nachdem der Operator ausgeführt wurde. Das Anfrageergebnis, das als Objekt vom Typ *JValue* (siehe Kapitel 7.3, Abbildung 7.2) vorliegt, muss daraufhin überprüft werden, ob ein Objekt vom Typ *JValueCollection* wiederum ein Objekt dieses Typs als Element beinhaltet (Durchlaufen der *JValueCollection* über einen *Iterator*), was bedeutet, dass eine verschachtelte Struktur vorliegt. Ist dies der Fall, bricht der Operator seine Berechnungen ab und gibt eine Fehlermeldung aus.

- *convertQueryResult(JValue queryResult)*
  Das Ergebnis einer `GReQL2`-Anfrage liegt als Java-Objekt vom Typ *JValue* vor (siehe [Bil06]). Diese Darstellung der Daten muss in die interne Speicherstruktur von `Yale` (siehe Kapitel 5.2.2) umgewandelt werden. In beiden Fällen liegen die Daten als Tabelle vor. Dabei müssen den n Elementen der *JValue*-Ergebnistabelle Metainformationen wie Attributnamen und -typen entnommen und als `Yale`-Objekt vom Typ *Attribute*, das die Metainformationen einer Spalte der Tabelle verwaltet, gespeichert werden. Die Attributnamen der Elemente der *JValue*-Ergebnistabelle können einfach übernommen werden. Bei den Attributtypen kann es sich um die in Kapitel 7.3 aufgelisteten `JValue`-Datentypen handeln. Diese Datentypen überschneiden sich größtenteils mit den von `Yale` unterstützen Datentypen bzw. können mithilfe einer Typumwandlung (cast) in einen „höheren“ Datentyp konvertiert werden. Code-Listing 8.2 zeigt die Konvertierung der `JValue`-Datentypen in `Yale`-Datentypen, wobei die

[2] Der Anfragetext müsste in einen Syntaxbaum übersetzt und dieser analysiert werden.

Variable *element* den Typ des in *JValue* gekapselten Wertes darstellt und die Variable *type* eine Integer-Darstellung der zugehörigen `Yale`-Typen ist. `Yale` wandelt die Integer-Darstellung *Ontology.<Datentyp>* über eine Abbildung von Integer-Werten auf konkrete Java-Typen intern in den jeweiligen <Datentyp> um. Die `JValue`-Typen *Edge* und *Vertex* repräsentieren einen eindeutigen Namen des jeweiligen Elements in `JValue` und werden deshalb als String übernommen. Diese Variablen sind potenzielle Kandidaten für das „ID Attribute“. Definiert man einen Typ als *Ontology.ATTRIBUTE_VALUE*, versucht Yale den Datentyp des Attributs selbst zu bestimmen. Gelingt dies nicht, wird eine Exception geworfen.

```
switch (element) {
        case BOOLEAN:   type = Ontology.BOOLEAN; break;
        case STRING:    type = Ontology.STRING; break;
        case INTEGER:   type = Ontology.INTEGER; break;
        case DOUBLE:    type = Ontology.REAL; break;
        case ENUMVALUE: type = Ontology.ORDERED; break;
        case EDGE:      type = Ontology.STRING; break;
        case VERTEX:    type = Ontology.STRING; break;
        default:        type = Ontology.ATTRIBUTE_VALUE;
}
```

Quellcode 8.2: Typkonvertierung der `JValue`- in `Yale`-Datentypen

Die eigentlichen Daten der einzelnen Tupel werden jeweils in einem `Yale`-Objekt vom Typ *DataRow* gespeichert, das eine Zeile der Tabelle darstellt. Jedes Element einer *DataRow* besitzt den Datentyp, der vom - für die jeweilige Spalte zuständigen - *Attribute* festgelegt ist. Eine *ExampleTable* verwaltet die Menge der Objekte vom Typ *Attribute* und *DataRow*.

- *generateExampleSet(ExampleTable table)*
  Damit andere Operatoren auf den eingelesenen Daten arbeiten können, stellt der Operator ein Objekt vom Typ *ExampleSet*, das eine Sicht auf eine zugrunde liegende *ExampleTable* repräsentiert und Subtyp von *IOObject* ist (siehe Kapitel 5.2.2), als Output zur Verfügung. Das *ExampleSet* ist in einem *IOContainer* gekapselt.

In der Klassenhierarchie der `Yale`-Operatoren findet sich keine bestehende Input-Operatorklasse die Teile der beschriebenen Funktionalität implementiert. Deswegen wird der Operator direkt von der abstrakten Klasse *Operator* abgeleitet.

8.2 Der Operator *Cluster2JValue*

Anforderung

Eine Clusteranalyse soll auf Softwareelementen durchgeführt werden. Die Softwareelemente werden von `GReQL2` als Objekte vom Typ *JValue* in einer Datentabelle bereitgestellt (siehe Kapitel 8.1), wobei die Softwareelemente jeweils als Vektor aus Attribut-Werte Paaren beschrieben sind. Es wäre wünschenswert, wenn das Ergebnis einer Clusteranalyse wieder „zurückgeschrieben“ werden könnte, d.h. es soll dem jeweiligen Datensatz (Vektor) in der Datentabelle des *JValue*-Objektes, das entsprechende Cluster, in das ein Softwareelement eingeteilt wurde, zugeordnet werden.

Entwurf

Für die Umsetzung dieser Anforderung wird ein Operator mit dem Namen *Cluster2JValue* erstellt. Die Schnittstelle - also die benötigten und erstellten *IOObject*s und die Parameter - des Operators, wird im Folgenden hergeleitet. Abbildung 8.4 zeigt vorgreifend den Aufbau und die Datentypen der Schnittstelle des Operators *Cluster2JValue*.

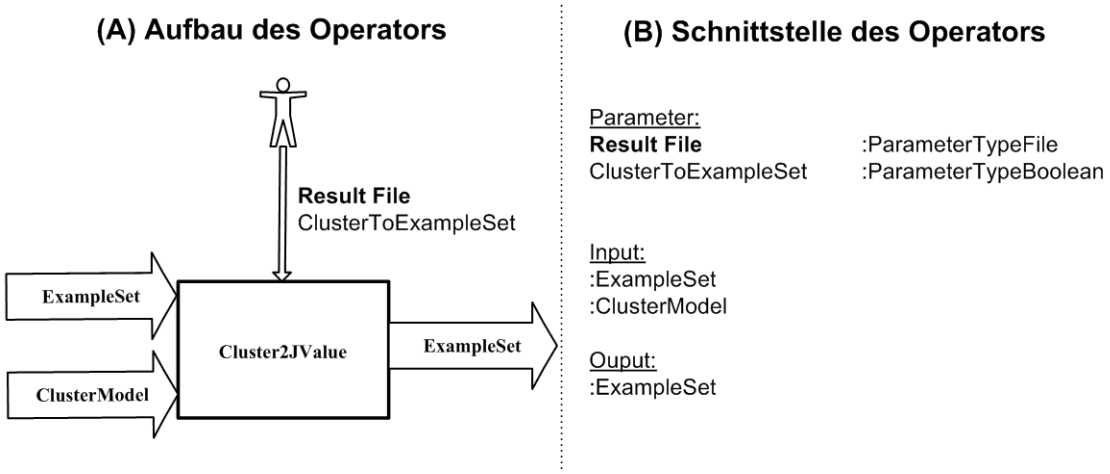


Abbildung 8.4: Aufbau (A) und Schnittstelle (B) des Operators *Cluster2JValue*

Input

Wie aus Abbildung 8.4 zu entnehmen ist, werden Input-Objekte vom Typ *ExampleSet* und *ClusterModel* benötigt. Folgende Ausführungen begründen dies und beschreiben, wie die Input-Objekte dazu genutzt werden, ein Rückschreiben der Ergebnisse einer Clusteranalyse in ein Objekt vom Typ *JValue* zu ermöglichen.

ExampleSet
Notwendig ist die ursprüngliche Datentabelle, die von *GreqlExampleSource* eingelesen wurde, da den Objekten dieser Datentabelle die Cluster zugeordnet werden sollen. Die Datentabelle wird von `GReQL2` als *JValue*-Objekt an *GreqlExampleSource* übergeben (siehe Kapitel 8.1) und in eine *ExampleTable* konvertiert. Als Output-Objekt wird von *GreqlExampleSource* ein *ExampleSet*, als eine spezielle Sicht auf eine *ExampleTable*, erstellt. Das *ExampleSet* wird durch die Operatorkette durchgereicht und von verschiedenen Operatoren modifiziert.

Der Operator *Cluster2JValue* erhält dieses *ExampleSet* als Input und rekonstruiert die ursprüngliche *ExampleTable* aus diesem. Dies ist möglich, da jedes *ExampleSet* über eine Referenz auf die *ExampleTable*, aus der es erzeugt wurde, verfügt. Die *ExampleTable* wird in ein *JValue*-Objekt konvertiert. Somit wurde die ursprünglich eingelesene Datentabelle als *JValue*-Objekt wieder hergestellt. Diese muss noch mit einem zusätzlichen Attribut versehen werden, dass die Clusterzugehörigkeit des jeweiligen Objektes speichert. Abbildung 8.5 verdeutlicht den beschriebenen Vorgang.

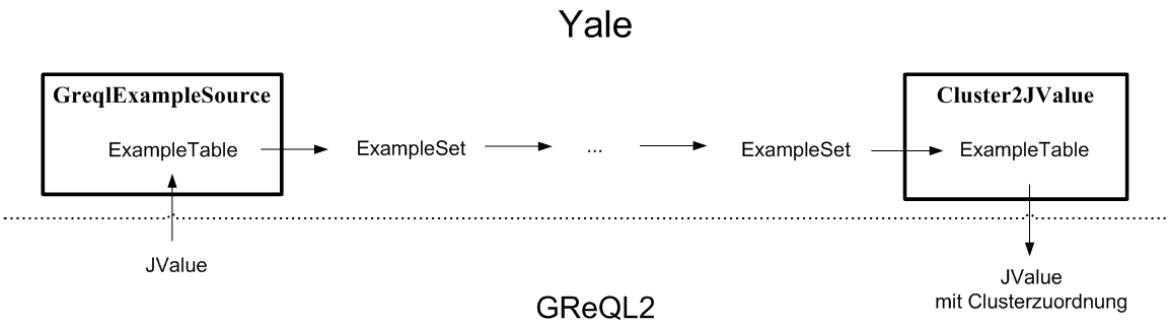


Abbildung 8.5: Konvertierung der Datentabelle in verschiedene Datentypen

ClusterModel
Weiterhin wird die Information benötigt, in welchem Cluster sich welches Softwareelement befindet. Diese Informationen sind in einem Objekt vom Typ *ClusterModel* (siehe Kapitel 5.5) gespeichert, das eine Zuordnung der Softwareelemente der Datentabelle - über dessen *id* - zu Clustern verwaltet und somit das Ergebnis eines Clusterverfahren-Operators darstellt. Die id´s müssen durch den Operator *Cluster2JValue* den zugehören Softwareelementen der Datentabelle zugeordnet werden. Als weiteres Input-Objekt wird also ein *ClusterModel* benötigt.

Zuordnung von Objekten zu einem Cluster
Es gilt zu unterscheiden zwischen einem *FlatClusterModel* und einem *HierachicalClusterModel* (siehe Kapitel 5.5). Ein *FlatClusterModel* speichert eine eindeutige Zuordnung von id´s zu Clustern, sodass die *JValue*-Datentabelle mit einem zusätzlichen Attribut „cluster_id“ vom Typ String versehen werden kann, das für das jeweilige Softwareelement das Cluster angibt, dem es zugeteilt wurde.

Anders sieht es bei einem *HierachicalClusterModel* - welches das Ergebnis eines hierarchischen Clusterverfahrens ist - aus. Hier wird ein Softwareelement nicht eindeutig in ein Cluster eingeteilt, sondern wird im Laufe des Verfahrens mehreren Clustern zugeordnet, da bei hierarchischen Verfahren keine feste Anzahl k an Clustern berechnet wird, sondern Lösungen für $1 \leq k \leq n$ (siehe Kapitel 2.3.2). Da die Reihenfolge, in der ein Softwareelement im Laufe des Verfahrens verschiedenen Clustern zugeordnet wird, eine wichtige Information darstellt, soll diese beim Rückschreiben der Ergebnisse berücksichtigt werden. Abbildung 8.6 zeigt das Ergebnis einer beispielhaften hierarchischen Clusteranalyse als Dendrogramm. Es handelt sich bei den zu gruppierenden Objekten im Beispiel nicht um Softwareelemente, sondern um Personen. Diese werden im Folgenden verallgemeinert als Objekte bezeichnet.

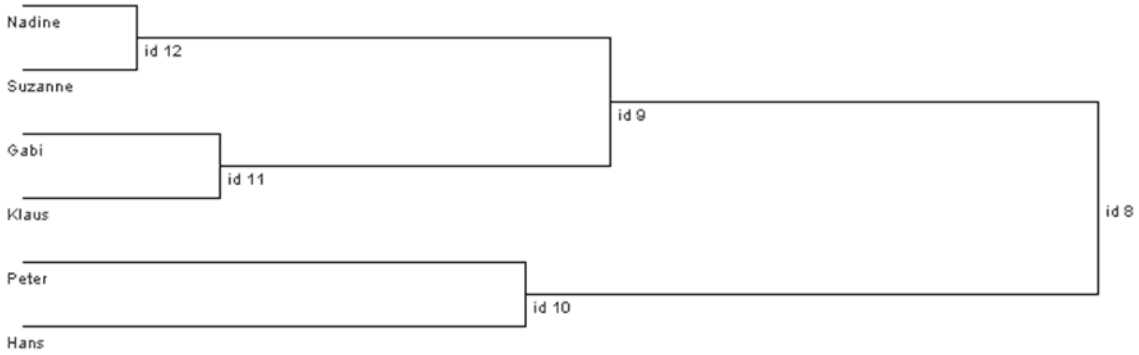


Abbildung 8.6: Ergebnis einer beispielhaften hierarchischen Clusteranalyse

Eine geeignete Lösung stellt ein Vorgehen nach dem Algorithmus der Union-Find-Datenstruktur (siehe [CLR00], Kapitel 22) dar, die eine Zerlegung einer Partition in disjunkte Mengen beschreibt. Dabei wird für jede disjunkte Menge ein Objekt der Menge als Repräsentant bestimmt. Zunächst bildet jedes Objekt eine eigene Menge, von der es selbst der Repräsentant ist. Bei Vereinigung (Union) zweier Mengen wird aus der neu entstehenden Menge, nach einem bestimmten Kriterium, ein Repräsentant gewählt (Find). Die Vereinigung von Mengen wird iterativ fortgeführt, bis sich letztendlich alle Objekte in einer Menge befinden. Über den Repräsentanten eines Objektes, kann dessen Zuordnung zu verschiedenen Mengen nachvollzogen werden.

Dieses Vorgehen wird auf das Ergebnis hierarchischer Clusterverfahren übertragen und davon ausgegangen, dass sich jedes Objekt zu Beginn in einem eigenen Cluster befindet. Nach Anwendung des Union-Find-Algorithmus auf die Datenstruktur verfügt jedes Objekt über genau einen Repräsentanten, über den die schrittweise Zuordnung des Objektes zu verschiedenen Clustern nachvollzogen werden kann. Abbildung 8.7 zeigt die Anwendung des Algorithmus auf das Clusteranalyse-Ergebnis aus Abbildung 8.6. Als Repräsentant eines Cluster wird das Objekt gewählt, dessen Anfangsbuchstabe im Alphabet zuerst vorkommt.

Tabelle 8.1 zeigt eine detaillierte Auflistung der einzelnen Iterationsschritte des hierarchisch agglomerativen Clusteranalyse-Algorithmus und der Zuordnung von Repräsentanten zu Objekten.

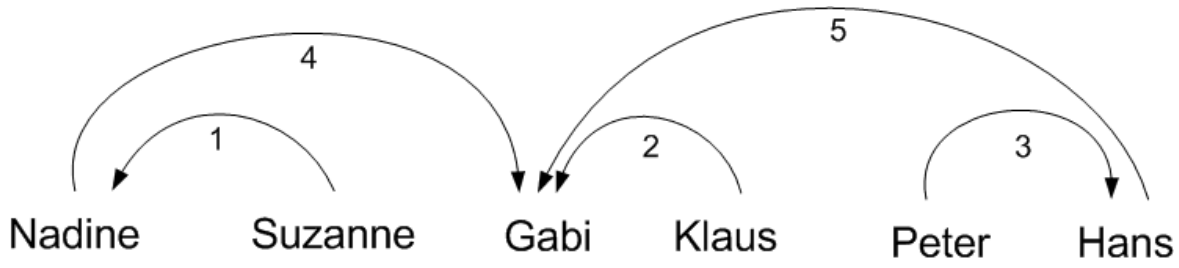


Abbildung 8.7: Union-Find-Algorithmus auf das Ergebnis einer hierarchischen Clusteranalyse

Iterationsschritt	Cluster A	Cluster B	Distanz	Neues Cluster (Repräsentant)
1	Nadine	Suzanne	$7,1$	Nadine
2	Gabi	Klaus	$12,4$	Gabi
3	Peter	Hans	$31,5$	Hans
4	Nadine	Gabi	$36,8$	Gabi
5	Hans	Gabi	67.3	Gabi

Tabelle 8.1: Iterationsschritte eines hierarchischen Clusterverfahrens

Liegt also ein *HierarchicalClusterModel* vor, wird jedes Softwareelement mit einer String-Variablen „Repräsentant“ versehen, die seinen Repräsentanten und zusätzlich den Iterationsschritt, in dem die Zuordnung zu diesem Repräsentanten stattfand, speichert. Das Objekt „Nadine“ aus Abbildung 8.7 wird also mit dem Wert „Gabi (4)“ versehen.

Output

Als Output wird das *JValue*-Objekt in einer Datei gespeichert, die der Benutzer über einen Parameter spezifizieren kann. Das *JValue*-Objekt wird unter Verwendung der Klasse *JValueXMLOutputVisitor* der `GreQL2`-Software (siehe [Bil06]) im XML-Format abgelegt. Es könnte der Bedarf bestehen, die Daten dieser Datei wieder in Yale zu laden, um z.B. weitere Verfahren, wie Visualisierung oder Evaluation der Clusterergebnisse, auf die Daten anzuwenden. In der `GReQL`-Klassenbibliothek existiert eine Klasse *ValueXMLLoader*, die das Laden des *JValue*-Objektes aus einer mit *JValueXMLOutputVisitor* erzeugten XML-Datei ermöglicht. Letztgenannte Funktion ist nicht Aufgabe dieses Operators und muss von einem zusätzlichen Operator realisiert werden.

Als Output-Objekt vom Typ *IOObject* wird - falls der boolsche Parameter „ClusterToExampleSet“ (siehe unten) auf true gesetzt wird - das Input-Objekt vom Typ *ExampleSet* modifiziert und weitergereicht. Ist der angesprochene Parameter mit false belegt, wird das *ExampleSet*, genauso wie das Input-Objekt vom Typ *ClusterModel*, unverändert weitergereicht.

Parameter

Ein verpflichtender Parameter „Result File“ ermöglicht die Angabe einer Datei, in die das *JValue*-Objekt gespeichert werden soll.

Ein boolscher Parameter „ClusterToExampleSet“ wird eingeführt, um zusätzlich zum Festschreiben der Clusterergebnisse in der Datentabelle des *JValue*-Objektes, ein Festschreiben in der Datentabelle des *ExampleSet*-Objektes zu ermöglichen (true). In diesem Falle wird das *ExampleSet* mit einem zusätzlichen Attribut versehen und das modifizierte Objekt als Output weitergegeben. `Yale` erlaubt das Speichern eines *ExampleSet* in einer so genannten „attribute description file“ (siehe Kapitel 5.2.2).

Der Operator wird direkt von *Operator* abgeleitet, da sich keine Operatoren in der `Yale`-Klassenhierarchie finden, die Teile der Funktionalität implementieren.

9 Das Plugin „Softwareclustering“

Die Funktionalität der Clusteranalyse ist in `Yale` als Plugin mit dem Namen „Clustering“ realisiert. Das Gebiet der Clusteranalyse ist ein sehr weites Feld: Um eine Menge von Objekten in Cluster einzuteilen und das Ergebnis auszuwerten, gibt es unzählige Konfigurationsmöglichkeiten, z.B.

- die Wahl eines Distanz- / Ähnlichkeitsmaßes,
- die Wahl eines Clusterverfahrens,
- bei hierarchischen Clusterverfahren die Wahl eines Linkage-Kriteriums und
- die Wahl einer geeigneten Visualisierungsmethode, um aus den Ergebnissen möglichst effizient neue Informationen gewinnen zu können.

Je nach Anwendungsgebiet und Untersuchungsziel eignen sich verschiedene Zusammensetzungen und Ausprägungen der genannten Punkte.

Die Vielfalt der Konfigurationsmöglichkeiten einer Clusteranalyse führt dazu, dass eine Software zur Clusteranalyse meist nur einen Teil der gesamten Palette abdecken kann. `Yale` unterstützt zwar bereits eine Vielzahl von Verfahren zur Konfiguration einer Clusteranalyse, doch sind einige Modifikationen bzw. Erweiterungen notwendig, um `Yale` optimal zur Durchführung von Clusteranalysen auf Softwareelementen nutzen zu können. Diese notwendigen Modifikationen bzw. Erweiterungen werden im Folgenden vorgestellt und in einem Plugin mit dem Namen „Softwareclustering“ zusammengefasst.

Abbildung 9.1 zeigt die Beziehungen der Software `Yale` und der Plugins „Clustering“ und „Softwareclustering“ als UML-Komponentendiagramm. Zusätzlich wird das Plugin „GReQL-Interface“ aufgeführt, das in Kapitel 8.1 vorgestellt wurde. Alle Komponenten werden über die Plugin-Schnittstelle in `Yale` integriert. Das Plugin „Softwareclustering“ nutzt bestehende Konzepte des `Yale`-Plugin „Clustering“ und erweitert diese. Es ist somit nur einsetzbar, wenn das Plugin „Clustering“ vorhanden ist. Die Plugins „Softwareclustering“ und „GReQL-Interface“ sind völlig unabhängig voneinander.

In den folgenden Abschnitten werden die UNZULÄNGLICHKEITEN der bestehenden Implementationen in `Yale` diskutiert. In den Abschnitten MODIFIKATION BZW. ERWEITERUNGEN werden die aus Sicht des Autors notwendigen Modifikationen bzw. Erweiterungen angesprochen und ihre konkrete Umsetzung in Form eines Softwareentwurfs beschrieben.

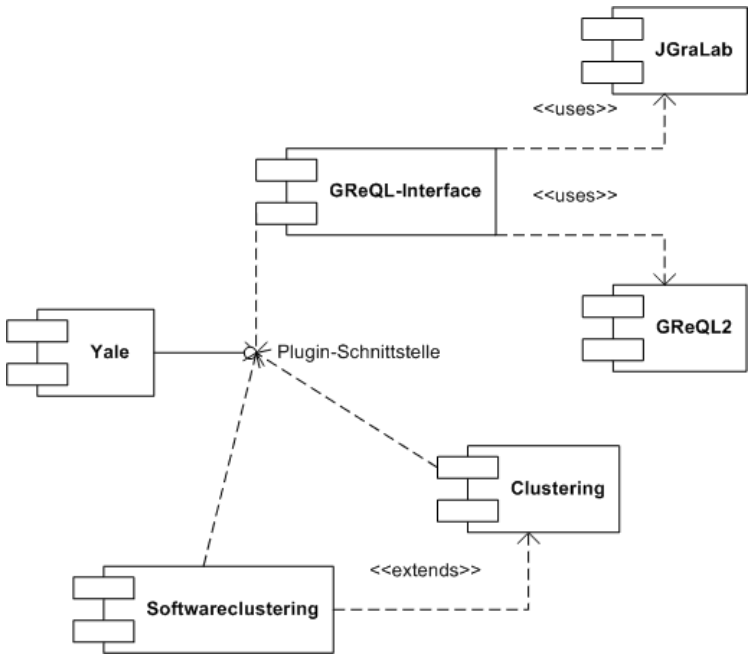


Abbildung 9.1: UML-Komponentendiagramm der Integration des Plugin „Softwareclustering“ in die Software `Yale`

9.1 Distanz- / Ähnlichkeitsfunktionen

Distanz- bzw. Ähnlichkeitsfunktionen berechnen aus der Datentabelle eine Distanz- bzw. Ähnlichkeitsmatrix, die fast allen Clusterverfahren als Grundlage ihrer Berechnung dient. Die Theorie bzgl. dieser Funktionen wurde ausführlich in Kapitel 2.2 behandelt. Im Folgenden soll untersucht werden, wie die Berechnung der Distanzen bzw. Ähnlichkeiten in `Yale` umgesetzt ist und welche Erweiterungen bzw. Modifikationen notwendig sind.

9.1.1 Unzulänglichkeiten

Alle Distanz- bzw. Ähnlichkeitsfunktionen sind jeweils als eigene Klasse realisiert. Es stehen die Funktionen *EuclidianDistance*, *ManhattanDistance* (siehe jeweils Kapitel 2.2, Formeln 2.9 und 2.10) und *CosineSimilarity* für quantitative Variablen zur Verfügung. Für qualitative Variablen wurde die Klasse *NominalDistance* realisiert, die für zwei Objekte o_i, o_j mit jeweils m Variablen die Distanz d als

$$d_{ij} = m - a$$

berechnet, wobei a für die Anzahl der Variablen steht, in denen sich o_i und o_j im gleichen Zustand befinden. Der Distanzwert, der aus dieser Funktion resultiert, ist stark abhängig

von der Anzahl m der Variablen: Je mehr nominale Variablen (je größer m), desto höher kann der Distanzwert werden. Um eine Normierung des Distanzwertes auf das Intervall [0,1] zu erreichen, sollte der Wert noch durch die Anzahl m der Variablen geteilt werden ($d_{ij} = \frac{m-a}{m}$).

Wie die Distanz- bzw. Ähnlichkeitswerte den einzelnen Clusterverfahren in `Yale` zur Berechnung übergeben werden, unterscheidet sich je nach Clusterverfahren. Folgende Umsetzungen sind zu finden:

- Der Operator *ExampleSet2Similarity* berechnet aus einer Datentabelle eine Distanz- bzw. Ähnlichkeitsmatrix. Die Datentabelle wird in `Yale` in einem Objekt vom Typ *ExampleSet* gespeichert, welches der Operator als Input erhält. Die berechnete Distanz- bzw. Ähnlichkeitsmatrix wird als Objekt vom Typ *SimilarityMeasure* als Output des Operators bereitgestellt. Die zu verwendende Distanz- bzw. Ähnlichkeitsfunktion wird dem Operator als Parameter übergeben. Zur Verfügung stehen *EuclidianDistance*, *ManhattanDistance* und *CosineSimilarity*. Einige Clusterverfahren-Operatoren erwarten ein *SimilarityMeasure* als Input (z.B. *AgglorematiyeClustering*, *DBScanClusering*, *KMedoids*).

- Einem Clusterverfahren-Operator kann die zu verwendende Distanz- bzw. Ähnlichkeitsfunktion selbst als Parameter übergeben werden: Der Operator *UPGMAClustering* (Variante des hierarchisch agglomerativen Verfahrens) erlaubt *EuclidianDistance* und *NominalDistance*, und berechnet über diese Funktionen selbst eine Distanzmatrix.

- Einige Operatoren, wie z.B. der *KMeans*-Operator arbeitet ausschließlich mit einer Distanz- bzw. Ähnlichkeitsfunktion (Quadratische Euklidischen Distanz). Die Anwendung dieser Funktion ist im Quellcode festgeschrieben. Dies ist im Algorithmus des K-Means Clusterverfahrens auch so vorgesehen, da die Distanz hier zu „imaginären“ Objekten während des Algorithmus immer wieder neu berechnet werden muss (siehe Kapitel 2.3.1). Es wird also keine Distanzmatrix im Voraus berechnet.

Egal welche der aufgeführten Varianten zugrunde liegt, bei jeder werden alle Variablentypen der Datentabelle gleich behandelt, im Falle von *EuclidianDistance*, *ManhattanDistance* und *CosineSimilarity* also als intervall-skalierte Variablen, und im Falle von *NominalDistance* als nominale Variablen. Wie in Kapitel 2.2.4 bereits beschrieben, liefert die Behandlung aller Variablen als intervall-skalierte Variablen durchaus sinnvolle Ergebnisse, ausgenommen für asymmetrisch binäre und nominale Variablen. Folgende Ausführungen verdeutlichen dies an einem Beispiel.

In `Yale` werden alle Daten als Double-Werte gespeichert. Dabei existiert für die Speicherung von String-Werten (die meist Werte einer nominalen Variable sind) eine injektive map-Funktion, die nummerische Werte auf entsprechende String-Werte abbildet. So könnten z.B. die Double-Werte 1.0, 2.0, 3.0 und 4.0 für die Haarfarben rot, schwarz, blond und braun stehen. Wird die Variable Haarfarbe wie eine intervall-skalierte Variable behandelt, ergibt sich für die Haarfarben rot und braun eine größere Distanz, als für rot und schwarz.

Die Behandlung aller Variablentypen als nominale Variablen liefert - vor allem für intervallskalierte Variablen - völlig verzerrte Ergebnisse und führt zu Informationsverlust. Zwei intervall-skalierte Werte 1 und 100 führen ebenso wie zwei Werte 1 und 1.1 zu einer Nicht-Übereinstimmung und damit zu einer gleich großen Distanz.

Die Verwendung einer einheitlichen Distanz- bzw. Ähnlichkeitsfunktion für alle Variablen (unterschiedlichen Typs) einer Datentabelle ist in der Praxis durchaus üblich. Die bekannte Data Mining Software `SPSS 14.0` erlaubt zwar die Wahl verschiedener Distanzfunktionen sowohl für intervall-skalierte, nominale und binäre Variablen, jedoch kann vom Benutzer nur eine dieser Funktionen gewählt werden, die dann einheitlich für alle Variablen der Datentabelle genutzt wird. Auch die fest implementierte Verwendung der Quadratischen Euklidischen Distanz für das K-Means-Clusterverfahren findet sich in `SPSS`. Dem Autor sind - auch nach Rücksprache mit einem Data Mining Experten der Universität Koblenz-Landau - keine weit verbreiteten Statistiksysteme bekannt, die für gemischte Variablentypen die vom jeweiligen Datentyp abhängigen Distanzen bzw. Ähnlichkeiten berechnen.

9.1.2 Modifikationen bzw. Erweiterungen

Die Ergebnisse einer Clusteranalyse hängen stark von den berechneten Distanzen bzw. Ähnlichkeiten ab. Die Behandlung von nominalen oder asymmetrischen binären Variablen als intervall-skalierte Variablen kann ein Ergebnis stark verzerren. Eine Datentabelle, die aus einer `GReQL2`-Anfrage resultiert, kann durchaus verschiedene Variablentypen aufweisen. Die Objekte dieser Datentabelle repräsentieren Softwareelemente. Handelt es sich bei diesen Softwareelementen bspw. um Java-Klassen, könnten diese durch die in Tabelle 9.1 aufgelisteten Variablen beschrieben werden.

id	Name	Typ	Wertebereich
a	Loc	intervall	$\mathbb{N}$
b	Anzahl globale Variablen	intervall	$\mathbb{N}$
c	Sichtbarkeit	nominal	{public, protected, private}
d	Subklasse von *JComponent*	asymmetrisch binär	{1=ja, 0=nein}

Tabelle 9.1: Variablen zur Beschreibung einer Java-Klasse

Eine einheitliche Behandlung dieser Variablentypen als intervall-skalierte oder als nominale Variablen würde wohl keine zufrieden stellende Einteilung der Java-Klassen in Cluster ergeben.

Anforderungen

Es ist es also wünschenswert, jeden Variablentyp mit den dafür vorgesehenen Funktionen zur Berechnung von Distanzen- bzw. Ähnlichkeiten zu behandeln.

Die zu erstellende Clusteranalyse-Umgebung soll als experimentelle Plattform genutzt werden. Um dem Benutzer vielfältiges Experimentieren zu gestatten, sollten zahlreiche Möglichkeiten zur Konfiguration einer Clusteranalyse offen gehalten werden. Ein hohes Maß an Freiheit und Flexibilität zur Wahl einer Funktion zur Berechnung von Distanzen bzw. Ähnlichkeiten wird dadurch erreicht, dass man dem Benutzer die freie Eingabe einer Funktion ermöglicht. Dabei soll der Benutzer innerhalb dieser Funktion auf alle Variablen der Datentabelle zugreifen können. Nimmt man als Schema bspw. Tabelle 9.1 und weist den Variablen - aus Gründen der Übersichtlichkeit - in der Reihenfolge ihres Auftretens in der Tabelle (von oben nach unten) die Buchstaben a, b, c, d des Alphabets als *id* zu, könnte die vom Benutzer anzugebende Funktion zur Berechnung der Distanz zwischen zwei Objekten o_1 und o_2 folgende Gestalt haben:

$$d(o_1, o_2) = ed(o_1.a, o_2.a, o_1.b, o_2.b)/2 + nom(o_1.c, o_2.c) + jd(o_1.d, o_2.d) * 4 \tag{9.1}$$

Dabei stehen *ed*, *nom* und *jd* für die Formeln der Euklidischen Distanz, des Simple Matching Koeffizientes für nominale Variablen und der Jaccard Distanz (siehe Kapitel 2.2). Die Variablenbezeichnung $o_1.a$ beschreibt den Platzhalter für den Wert der Variable a des Objektes o_1: Eine Distanz- bzw. Ähnlichkeitsfunktion erhält als Eingabe immer die Datenvektoren zweier Objekte.

Es soll dem Benutzer jedoch auch möglich sein, eine völlig freie Eingabe einer Funktion, ohne Bezug auf existierende Distanz- bzw. Ähnlichkeitsfunktionen, wie z.B. die Euklidische Distanz, durchzuführen. So könnte eine Distanzfunktion für die Variable „Loc“ (a) und „Subklasse von JComponent“ (d) folgendermaßen aussehen:

$$d(o_1, o_2) = sqrt(o_1.a + o_2.a) - 2 * (o_1.d * o_2.d) \tag{9.2}$$

Die Eingabe einer Funktion durch den Benutzer bringt den Vorteil mit sich, dass der Benutzer sein Anwendungswissen gezielt einsetzen kann, um für jede Variable eine geeignete Funktion zur Berechnung einer Distanz bzw. Ähnlichkeit zu spezifizieren. In Beispielfunktion 9.1 wurde das Ergebnis der Euklidischen Distanz der Variablen „Loc“ (a) und „Anzahl globale Variablen“ (b) halbiert und das Ergebnis der Variable „Subklasse von JComponent“ (d) entsprechend der Wichtigkeit der Variable in den Augen des Benutzers mit vier multipliziert (Gewichtung). Beispielfunktion 9.2 zeigt eine benutzerdefinierte Funktion zur Berechnung der Distanz von „Loc“ (a), wobei die Wurzel ($sqrt()$) der Summe der beiden Werte gebildet wird und vom Resultat anschließend zwei abgezogen wird, falls es sich bei beiden Objekten um Subklassen von JComponent handelt (hier wird *true* als 1.0 ausgewertet). Im Falle von binären Variablen kann der Benutzer selbst spezifizieren, ob die Variable als symmetrisch oder asymmetrisch binäre Variable behandelt wird.

Zusätzlich zu den bisher aufgeführten Anforderungen soll es dem Benutzer möglich sein, quantitative Variablen zu normieren (siehe Kapitel 2.2.1). Nicht normierte quantitative Variablen können zu verzerrenden Ergebnissen führen. Nimmt man bspw. zwei Softwareele-

mente mit den Variablen aus Tabelle 9.1 und eine Belegung von (500, 10) bzw. (20, 5) für die Variablen (Loc, Anzahl globaler Variablen) der beiden Softwareelemente, ist der Einfluss der Variablen „Anzahl globaler Variablen“ auf die errechnete Distanz bzw. Ähnlichkeit verschwindend gering. Die Werte dieser Variablen - und damit auch die Differenz der Werte zweier Objekte - sind meist inhärent geringer als die der Variablen „Loc“. Die Normierung soll jedoch nur nach Bedarf durchgeführt werden, da eine Normierung nicht immer erwünscht ist (siehe [KR90], S.5-9).

Im Folgenden soll geprüft werden, inwieweit diese Anforderungen in `Yale` - unter überschaubarem Aufwand - umgesetzt werden können.

Entwurfsentscheidungen

Zur Realisierung der Anforderungen sind folgende Erweiterungen erforderlich:

1. Liegen quantitative Variablen vor, muss der Benutzer die Möglichkeit haben, die Daten vor der Distanz- bzw. Ähnlichkeitsberechnung zu normieren.
2. Die Eingabe einer mathematischen Funktion zur Berechnung der Distanz bzw. Ähnlichkeit muss vom Benutzer durchgeführt werden können. Dabei muss innerhalb der Funktion ein Bezug zu den Variablen(-namen) der Datentabelle hergestellt werden können. Aus Gründen der Benutzerfreundlichkeit sollen bekannte Distanz- bzw. Ähnlichkeitsfunktionen bereits über ein Schlüsselwort aufrufbar sein (siehe Formel 9.1, *ed*, *nom* und *jd*).
3. Variablen, die in der Datentabelle vorkommen, vom Benutzer in der spezifizierten Funktion jedoch nicht explizit angegeben werden, müssen angemessen behandelt werden.

zu Punkt 1: Zur Normierung quantitativer Variablen wird Formel 2.13 aus Kapitel 2.2.1 verwendet.

zu Punkt 2: Zur Spezifikation einer mathematischen Formel in `Yale` steht lediglich der ASCII- Zeichenvorrat zur Verfügung. Die Umsetzung von Punkt 2 erfordert daher einen Parser, der die Formel einliest und die spezifizierten Symbole in mathematische Konstrukte umsetzt. Zusätzlich wird eine formale Sprache benötigt, die die Syntax und Semantik der einzugebenden Funktion definiert. So müssen z.B. für das Wurzelsymbol Symbole aus dem ASCII-Zeichenvorrat definiert werden, die stellvertretend für die Wurzel stehen. In Kapitel 1 wurde der in dieser Arbeit verfolgte Ansatz der Wiederverwendung und Kopplung bestehender Software kurz beschrieben (Stichwort „*commercial off-the-shelf*“). Da die Eigenentwicklung einer zuvor beschriebenen formalen Sprache und des zugehörigen Parsers sich als sehr zeitaufwendig gestalten würde, soll dieser Ansatz auch hier weiterverfolgt werden.

Als Alternativen bieten sich Lösungen in einer funktionalen Programmiersprache (z.B. Haskell) an, die sich vor allem für die leichte Implementierung der in Punkt 2 angesprochenen

vordefinierten Funktionen eignen, und eine Lösung in Java, da `Yale` selbst in Java erstellt wurde und dadurch die Software in einer einheitlichen Programmiersprache gehalten werden würde. Als Nachteil der zuerst genannten Alternativen ist anzusehen, dass dadurch, für die Benutzung des Plugin „Softwareclustering“, ein Interpreter der jeweiligen Sprache notwendig wäre. Trotzdem wurden bei der Entscheidung beide Alternativen in Betracht gezogen und bestehende Bibliotheken auf Eignung überprüft. Der Auswahlprozess und die betrachteten Alternativen sollen hier nicht detailliert vorgestellt werden, da dies nicht Inhalt des Kapitels ist.

Als besonders geeignet stellt sich der in Java realisierte und freie[1] Parser `JEP` (Java Math Expression Parser)[2] heraus. Der Parser erlaubt die Verarbeitung benutzerdefinierter Variablen, Konstanten und Funktionen und stellt zusätzlich bereits die gängigsten mathematischen Funktionen und Konstanten zur Verfügung. `JEP` definiert eine intuitive formale Sprache zur Spezifikation einer mathematischen Funktion und stellt eine einfache und klar dokumentierte Schnittstelle bereit. Außerdem unterstützt `JEP` die Verarbeitung boolscher Variablen mit boolschen Operatoren, was für die Distanz- bzw. Ähnlichkeitsberechnung von binären und nominalen Variablen sehr von Vorteil ist. Die genannten Aspekte qualifizieren `JEP` zum Einsatz in der Clusteranalyse-Umgebung, weshalb die Klassenbibliothek verwendet wird. Eine kurze Einführung in `JEP` und die Einbindung in das Plugin „Softwareclustering“ werden im Abschnitt 9.1.3 beschrieben.

zu Punkt 3: Für die Umsetzung der in Punkt 3 angesprochenen Behandlung der Variablen der Datentabelle, die der Benutzer nicht in der Funktion spezifiziert, bieten sich zwei Alternativen. Zum Einen könnte man diese Variablen bei der Berechnung von Distanzen bzw. Ähnlichkeiten und der anschließenden Gruppierung ignorieren. Zum Anderen könnte man den Benutzer eine Default-Funktion aus einer Menge vorgegebener Funktionen auswählen lassen, mit der die nicht spezifizierten Variablen berechnet werden. Letztgenannte Alternative ist aus Gründen des Verstoß gegen das softwaretechnische Prinzip der Überraschungsminimierung als problematisch anzusehen. Der Benutzer erwartet, dass genau die Variablen bei der Distanz- bzw. Ähnlichkeitsberechnung berücksichtigt werden, die er in der Funktion spezifiziert hat. Verwendet man zusätzlich eine Default-Funktion, sind die Berechnungen für den Benutzer kaum noch nachvollziehbar. Deshalb sollen die vom Benutzer nicht berücksichtigten Variablen bei der Berechnung der Distanzen bzw. Ähnlichkeiten und der anschließenden Gruppierung weggelassen werden.

Es ist Aufgabe des Benutzers, sich zu merken, welche Variablen der Datentabelle zur Gruppierung verwendet wurden und welche nicht. Liegt eine Datentabelle mit n ($n > 2$) Variablen vor, wurden zur Gruppierung aber nur 2 Variablen verwendet, spiegelt die resultierende Gruppierung nicht die Gruppierung der Objekte nach allen Eigenschaften (Variablen) wider. Dies muss vom Benutzer berücksichtigt werden.

[1] GNU General Public License
[2] http://www.singularsys.com/jep/

Entwurf

Nachdem zunächst die Anforderungen abgesteckt und grundlegende Entwurfsentscheidungen getroffen wurden, wird im Folgenden die konzeptionelle Umsetzung der Anforderungen behandelt. Dabei wird sich zunächst nur auf den *Entwurf eines eigenständigen Operators* zur Berechnung einer Distanz- bzw. Ähnlichkeitsmatrix konzentriert und im Anschluss eine Umsetzung dieses Konzeptes auf andere Ansätze der Distanz- bzw. Ähnlichkeitsberechnung (Distanz- bzw. Ähnlichkeitsfunktion als Parameter des Clusterverfahren-Operators) geprüft.

Eine direkte Erweiterung des bestehenden Quellcodes wurde in Kapitel 7.4 ausgeschlossen. Es muss also eine neue Operatorklasse erstellt werden, die in das „Softwareclustering“-Plugin gepackt wird. Da viele grundsätzlich neue Konzepte eingeführt und bestehende Konzepte verändert werden, ist eine Erweiterung des existierenden Operators *ExampleSet2Similarity* - der ebenfalls eine Distanz- bzw. Ähnlichkeitsmatrix aus einer Datentabelle berechnet - nicht sinnvoll. Die neue Operatorklasse wird *DistanceOrSimilarityCalculator* genannt.

Abbildung 9.2 (A) und (B) zeigt den aus den folgenden Ausführungen resultierenden Aufbau und die Datentypen der Schnittstelle des Operators *DistanceOrSimilarityCalculator*.

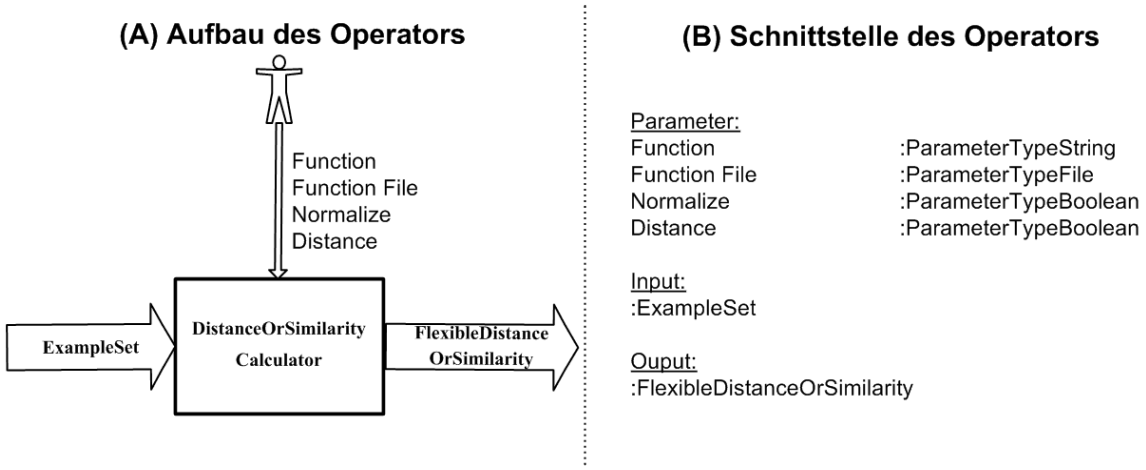


Abbildung 9.2: Aufbau (A) und Schnittstelle (B) des Operators *DistanceOrSimilarityCalculator*

Parameter

Der Operator *DistanceOrSimilarityCalculator* erlaubt die freie Eingabe einer Funktion als String („Function“). Die Funktion kann auch in einer Textdatei gespeichert und vom Operator geladen werden („Function File“). Die direkte Eingabe der Funktion als String hat Priorität gegenüber der Angabe in einer Datei. Wurde weder eine Funktion als String noch als Datei angegeben, erzeugt der Operator eine Fehlermeldung mit dem Hinweis, einen der beiden Parameter zu belegen. Außerdem kann über einen boolschen Parameter angegeben

werden, ob eine Normierung quantitativer Variablen durchgeführt werden soll („Normalize“). Ein weiterer boolscher Parameter gibt an, ob es sich bei der in „Function“ oder „Function File“ spezifizierten Funktion um eine Distanz- (true) oder Ähnlichkeitsfunktion (false) handelt. Die korrekte Angabe dieses Parameter ist von sehr hoher Bedeutung, da es sonst zu fehlerhaften Berechnungen kommt.

Output
Damit die bestehenden Clusterverfahren-Operatoren auf dem Ergebnis (berechnete Distanz- bzw. Ähnlichkeitsmatrix) der neuen Operatorklasse *DistanceOrSimilarityCalculator* arbeiten können, muss dieses in einem Objekt, das Subtyp von *SimilarityMeasure* ist, gespeichert werden. Diese Klasse wird *FlexibleDistanceOrSimilarity* genannt und im weiteren Verlauf dieses Abschnittes näher erläutert.

Zum besseren Verständnis der neuen Konzepte werden in Abbildung 9.3 zunächst die in `Yale` bestehende Klassenhierarchie des Interface *SimilarityMeasure* und deren Beziehungen zu der Operatorklasse *ExampleSet2Similarity* und den Clusterverfahren-Operator, die ein *SimilarityMeasure* als Input erwarten (hier als abstrakte Klasse *Clusterer* dargestellt), aufgezeigt. Die abstrakte Klasse *BasicExampleBasedSimilarity* ist Oberklasse aller Klassen die Distanz- bzw. Ähnlichkeitsfunktionen realisieren und stellt gemeinsame Methoden für diese bereit.

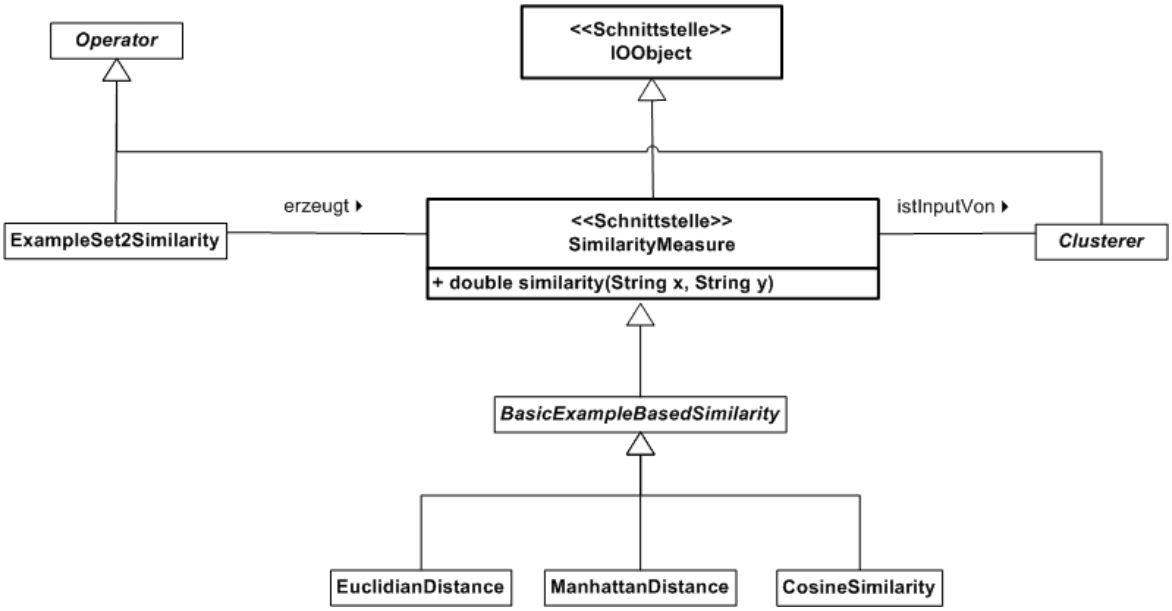


Abbildung 9.3: Klassenhierarchie des Interface *SimilarityMeasure* in `Yale`

Die Operatorklasse *ExampleSet2Similarity* berechnet nicht selbst die Distanz- bzw. Ähnlichkeitsmatrix, sie instanziert lediglich eine konkrete Subklasse vom Typ *SimilarityMeasure* entsprechend der Distanz- bzw. Ähnlichkeitsfunktion, die der Operator als Parameter erhalten hat - bspw. die Klasse *EuclidianDistance* - und stellt die Instanz als Output bereit.

Der folgende Clusterverfahren-Operator erhält die erzeugte Instanz als Input und baut durch wiederholtes Aufrufen der Methode

public double similarity(String x, String y)

dieser Instanz schrittweise (Iteration über Objekte der Datentabelle) die Distanz- bzw. Ähnlichkeitsmatrix auf. Die Variablen *x* und *y* repräsentieren dabei die Objekt-*id*s zweier Objekte, zwischen denen eine Distanz- bzw. Ähnlichkeit berechnet werden soll. Die Methode implementiert - je nach Klasse - die jeweilige Distanz- bzw. Ähnlichkeitsfunktion und gibt den berechneten Wert als *double* zurück.

Abbildung 9.4 zeigt die Eingliederung der neuen Konzepte in die Klassenhierarchie aus Abbildung 9.3. Der Operator *DistanceOrSimilarityCalculator* erzeugt ein Objekt vom Typ *FlexibleDistanceOrSimilarity* und übergibt diesem die zu berechnete Funktion als String. Die übergebene Funktion wird von der Klasse *FlexibleDistanceOrSimilarity* auf das Vorhandensein der Variablen aus der Datentabelle überprüft und die in der Funktion spezifizierten Variablen beim Parser `JEP` registriert. Der Parser wird dann jeweils für die Variablenbelegungen zweier Objekte aufgerufen und die Funktion von diesem ausgewertet. Die Kommunikation zwischen Parser und „Softwareclustering“-Plugin wird in Kapitel 9.1.3 detailliert behandelt.

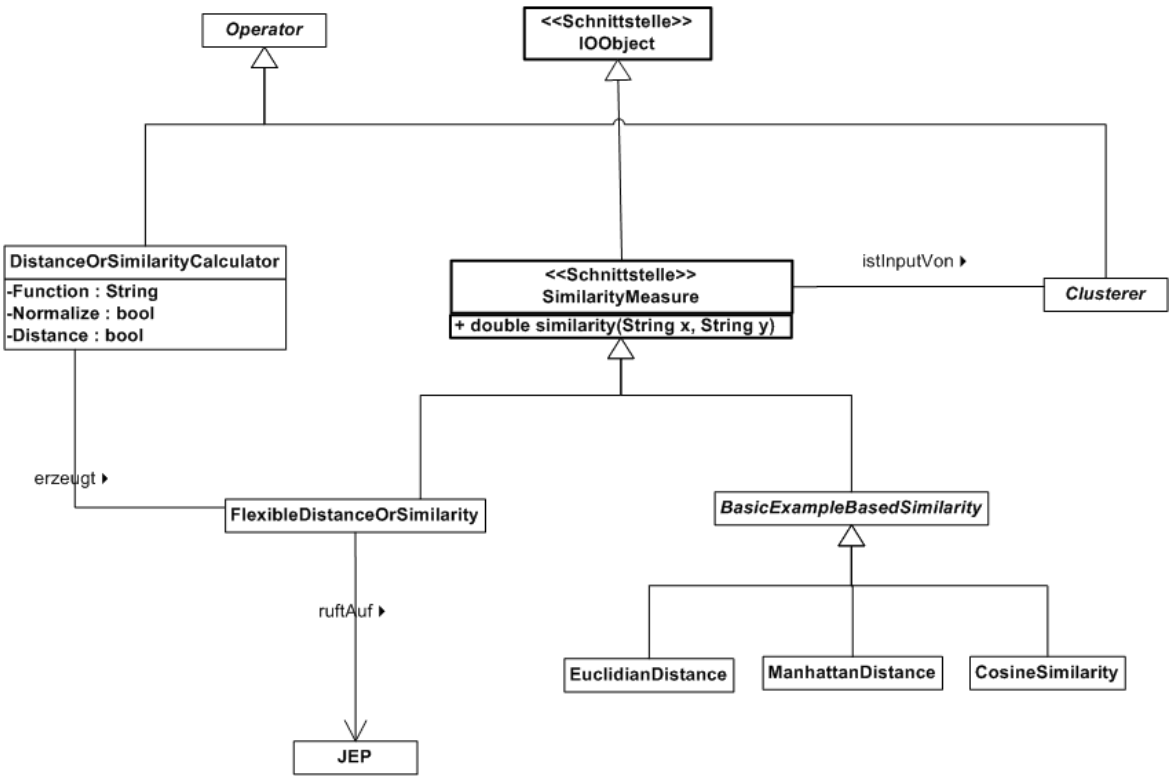


Abbildung 9.4: Klassenhierarchie des Interface *SimilarityMeasure* nach Erweiterung

Wie in Abschnitt 9.1.1 bereits kurz angesprochen, existieren in Yale Clusterverfahren-Operatoren, die kein *SimilarityMeasure* als Input erwarten. Um die neu eingeführten Konzepte für diese Operatoren anwendbar zu machen, müssten die Operatoren selbst modifiziert werden. Für Operatoren, die die Wahl einer zu verwendenden Distanz- bzw. Ähnlichkeitsfunktion selbst implementieren, müsste diese Funktionalität ausgelagert werden. Bei Operatoren, die die Berechnungen von Distanzen bzw. Ähnlichkeiten mit einer festgelegten Funktion implementieren, müsste der implementierte Clusteranalyse-Algorithmus modifiziert werden. Es sollte vorher jedoch für jeden Operator überprüft werden, ob die freie Wahl einer Distanz- bzw. Ähnlichkeitsfunktion überhaupt wünschenswert ist, da einige Clusterverfahren auf eine bestimmte Funktion abgestimmt sind.

9.1.3 Der Parser JEP

In diesem Abschnitt werden zunächst die Grammatik des Parsers JEP und die bereitgestellten Operatoren und Funktionen behandelt. Im Anschluss werden kurz die speziellen Funktionen zur Berechnung von Distanzen und Ähnlichkeiten aufgeführt, um die der Parser erweitert wurde. Danach werden anhand einer beispielhaften Datentabelle einige Beispielfunktionen gezeigt, die vom Parser ausgewertet werden können. Am Ende des Abschnitts wird die Kommunikation zwischen dem Plugin „Softwareclustering“ und dem Parser behandelt.

Dieser Abschnitt ist insbesondere an Leser gerichtet, die den Parser zur Berechnung von Distanzen bzw. Ähnlichkeiten nutzen wollen und an Leser, die eine Erweiterung von JEP um zusätzliche Funktionen anstreben.

Um dem Leser das Verständnis der Grammatik zu erleichtern und ihm einen ersten Eindruck der Fähigkeiten von JEP zu vermitteln, wird an dieser Stelle zunächst eine kleine Beispielfunktion aufgezeigt. Die Funktion $V = f(r) = \frac{4}{3}\pi r^3$ berechnet das Volumen einer Kugel. Die Eingabefunktion für JEP würde lauten:

*(4/3) * PI * r^3*

Dabei ist PI zunächst genauso eine Variable wie r. Über einen Methodenaufruf kann man PI, durch Zuweisen eines konstanten Wertes, als Konstante deklarieren. Für r kann man bei jedem (Auswertungs-)Aufruf der Funktion einen neuen Wert übergeben, weshalb r eine Variable ist. Als Ergebnis der Auswertung der Funktion erhält man den Funktionswert in Abhängigkeit der Belegung der Variablen, also in diesem Fall $f(r)$, zurück. Das in Formeln oftmals weggelassene Multiplikationszeichen muss explizit angegeben werden. Das Symbol ^ steht für eine Potenz (siehe Tabelle 9.2).

Grammatik, Operatoren und Funktionen

Grammatik

Listing 9.1 zeigt die Grammatik des Parsers in einer EBNF-ähnlichen Notation. Die Gram-

matikbeschreibung wurde von den JEP-Autoren mit dem Tool jjdoc des von JEP genutzten Parsergenerators JavaCC automatisch generiert. Die Darstellung im Listing ist übernommen von der JEP-Homepage[3].

Operatoren
Tabelle 9.2 zeigt die vollständige Liste von Operatoren die von JEP unterstützt werden. Alle gängigen arithmetischen und boolschen Operatoren können verarbeitet werden. Boolsche Ausdrücke werden zu 1 (true) oder 0 (false) ausgewertet. Die Tabelle ist abgeleitet aus der Operatortabelle der JEP-Homepage[4]. Spalte eins zeigt den Operatornamen, Spalte zwei das Symbol des Operators, das in der Funktion spezifiziert werden muss.

Name	Symbol
Potenz	^
Boolsches Nicht	!
Unär Plus, Unär Minus	+x, -x
Mod	%
Division	/
Multiplikation	*
Addition, Substraktion	+, −
Kleiner Gleich, Größer Gleich	<=, >=
Kleiner, Größer	<, >
Ungleich, Gleich	!=, ==
Boolsches Und	&&
Boolsches Oder	||

Tabelle 9.2: Unterstütze Operatoren in JEP

Funktionen
Zusätzlich zu den aufgeführten Operatoren bietet JEP einige vordefinierte Funktionen an. Diese werden in Tabelle 9.3 aufgelistet. Spalte eins gibt eine umgangssprachliche Beschreibung der Funktion an, Spalte zwei das Kürzel der Funktion mit den zugehörigen Parametern, das in der Eingabefunktion spezifiziert werden muss. In Spalte drei findet sich, falls notwendig, eine kurze Erklärung der Funktion bzw. deren Parametern.

Erweiterung des Funktionsumfanges

Wie in Abschnitt ENTWURFSENTSCHEIDUNGEN bereits kurz erwähnt, erlaubt JEP die Erweiterung um benutzerdefinierte Funktionen. Um dem Benutzer die Eingabe einer Formel zur Distanz- bzw. Ähnlichkeitsberechnung zu erleichtern, sollte dieser auf gängige Funktionen zurückgreifen können.

[3] http://www.singularsys.com/jep/doc/html/grammar.html Stand: 18.10.2006
[4] http://www.singularsys.com/jep/doc/html/operators.html Stand: 18.10.2006

```
Start ::= ( Expression ( <EOF> | <SEMI> ) |
          ( <EOF> | <SEMI> ) )

Expression ::= AssignExpression | OrExpression

AssignExpression ::= ( Variable <ASSIGN> Expression )

OrExpression ::= AndExpression ( ( <OR> AndExpression ) )*

AndExpression ::= EqualExpression
                  ( ( <AND> EqualExpression ) )*

EqualExpression ::= RelationalExpression
           ( ( <NE> RelationalExpression ) |
             ( <EQ> RelationalExpression ) )*

RelationalExpression ::= AdditiveExpression
                 ( ( <LT> AdditiveExpression ) |
                   ( <GT> AdditiveExpression ) |
                   ( <LE> AdditiveExpression ) |
                   ( <GE> AdditiveExpression ) )*

AdditiveExpression ::= MultiplicativeExpression
            ( ( <PLUS> MultiplicativeExpression ) |
              ( <MINUS> MultiplicativeExpression ) )*

MultiplicativeExpression ::= UnaryExpression
                         ( ( PowerExpression ) |
                           ( <MUL> UnaryExpression ) |
                           ( <DOT> UnaryExpression ) |
                           ( <CROSS> UnaryExpression ) |
                           ( <DIV> UnaryExpression ) |
                           ( <MOD> UnaryExpression ) )*

UnaryExpression ::= ( <PLUS> UnaryExpression ) |
                    ( <MINUS> UnaryExpression ) |
                    ( <NOT> UnaryExpression ) |
                    PowerExpression

PowerExpression ::= UnaryExpressionNotPlusMinus
                   ( ( <POWER> UnaryExpression ) )
```

```
UnaryExpressionNotPlusMinus ::= AnyConstant |
                                ( Function | Variable ) |
                                <LRND> Expression <RRND> |
                                ListExpression

ListExpression ::= ( <LSQ> Expression
                   ( <COMMA> Expression )* <RSQ> )

Variable ::= ( Identifier )

Function ::= ( Identifier <LRND> ArgumentList <RRND> )

ArgumentList ::= ( Expression ( <COMMA> Expression )* )

Identifier ::= ( <INDENTIFIER1> | <INDENTIFIER2> )

AnyConstant ::= ( <STRING_LITERAL> | RealConstant )

RealConstant ::= ( <INTEGER_LITERAL> |
                   <FLOATING_POINT_LITERAL> )
```

Listing 9.1: Grammatik des JEP-Parsers

Beschreibung	Kürzel	Erklärung
Sinus-Funktionen	sin(x), asin(x)	
Kosinus-Funktionen	cos(x), acos(x)	
Tangens-Funktionen	tan(x), atan(x)	
Natürlicher Logarithmus	ln(x)	
Logarithmus Basis 10	log(x)	
Exponentialfunktion	exp(x)	
Betrag	abs(x)	
Zufallszahl in [0,1]	rand()	
Modulo	mod(x,y)	= x % y
Wurzel	sqrt(x)	
Summe	sum(f(x),x,y,z)	$\sum_{x=y}^{z} f(x)$
If	if(x,y,z)	x=condition, y=truevalue, z=falsevalue
Binomialkoeffizient	binom(n,i)	

Tabelle 9.3: Vordefinierte Funktionen in JEP

Problematisch gestaltet sich dabei die Tatsache, dass Funktionen in JEP auf einer im Voraus fest definierten Anzahl von Parametern arbeiten, eine Funktion zur Berechnung der Distanz bzw. Ähnlichkeit jedoch auf beliebig viele Variablen, also Parameter, zurückgreifen kann. Die Euklidische Distanz kann bspw. für $1...n$ Variablen mit $n \epsilon \mathbb{N}$ berechnet werden. Um dies zu ermöglichen, muss JEP um die Eigenschaft beliebig vieler Parameter für Funktionen erweitert werden.

Abbildung 9.5 zeigt die Erweiterung von JEP um Funktionen zur Distanz- bzw. Ähnlichkeitsberechnung. Bei diesem Entwurf wurde darauf geachtet, dass die Klassenbibliothek von JEP nicht selbst modifiziert werden muss.

Die abstrakten Klassen

Die Klasse *PostfixMathCommand* ist Teil der JEP-Klassenbibliothek. Von dieser Klasse müssen alle Klassen abgeleitet werden, die eine Funktion implementieren. Die abstrakte Klasse *ArbitraryParametersFunction* realisiert die Eigenschaft, dass eine Funktion beliebig viele Parameter besitzen kann, die Funktion muss jedoch mindestens eine Variable als Parameter besitzen. Die drei direkten Subklassen von *ArbitraryParametersFunction* behandeln jeweils unterschiedliche Variablentypen. Diese Klassen wurden eingeführt, da Funktionen für unterschiedliche Variablentypen, verschiedene Typen von Parametern erwarten können. Die Parameterliste einer Funktion wird in diesen Klassen so vorverarbeitet, dass sie für die jeweiligen konkreten Subklassen in einer einfach zu verarbeitenden Weise vorliegen. Damit ist eine einfache Erweiterung der Klassenbibliothek um weitere Distanz- bzw. Ähnlichkeitsfunktionen garantiert.

Die abstrakte Klasse *AbstractIntervalFunction* ist Oberklasse aller Klassen, die Funktionen für intervall-skalierte Variablen implementieren. Als Parameter sind ausschließlich Varia-

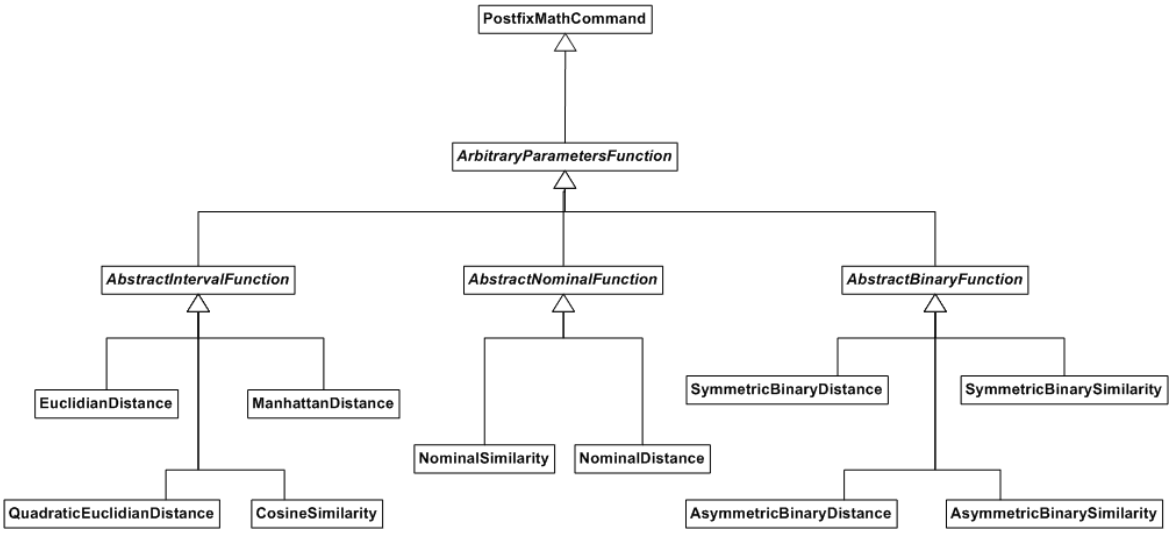


Abbildung 9.5: Klassenhierarchie der Funktionen zur Distanz- bzw. Ähnlichkeitsberechnung

blen erlaubt. Konkrete Subklassen müssen lediglich die von *AbstractIntervalFunction* definierte, abstrakte Methode

public abstract Double evaluate(double[] values);

implementieren. Der Übergabeparameter *values* enthält die Werte der beiden Objekte, für die eine Distanz bzw. Ähnlichkeit berechnet werden soll. Die Länge des Arrays ist gleich der Anzahl der Variablen, die der Funktion als Parameter übergeben wurden. Die Anzahl der Variablen muss gerade sein, da Distanz- bzw. Ähnlichkeitsfunktionen einen Abstand zwischen zwei Punkten im $\mathbb{R}^n$ berechnen und die Anzahl der Koordinaten der beiden Punkte gleich sein muss. Dies gilt auch für die im Folgenden vorgestellten evaluate-Methoden.

Die abstrakte Klasse *AbstractNominalFunction* ist Oberklasse aller Klassen, die Funktionen für nominale Variablen implementieren. Solche Funktionen können neben einer beliebigen (geraden) Anzahl von Variablen, als letzten Parameter, einen *double*-Wert enthalten, der als Gewichtungsfaktor genutzt wird. Wie dieser Parameter in der Berechnung verwendet wird, hängt von der konkreten Implementierung der Subklasse ab (siehe „Die konkreten Funktionsklassen“). Der Parameter ist optional. Wird er nicht explizit angegeben, wird der Defaultwert 1.0 zur Berechnung verwendet. Konkrete Subklassen müssen lediglich die von *AbstractNominalFunction* definierte, abstrakte Methode

public abstract Double evaluate(double[] values, double weighting);

implementieren.

Die abstrakte Klasse *AbstractBinaryFunction* ist Oberklasse aller Klassen, die Funktionen für binäre Variablen implementieren. Diese Funktionen können neben einer beliebigen Anzahl von Variablen, als letzte Parameter zwei *double*-Werte enthalten. Der erste der beiden *double*-Werte wird als Gewichtungsfaktor für Übereinstimmungen, der zweite für Nicht-Übereinstimmungen genutzt. Die Verwendung der Parameter hängt wieder von der konkreten Implementierung in der Subklasse ab (siehe „Die konkreten Funktionsklassen“). Die Parameter sind optional, es müssen jedoch entweder beide oder keiner dieser Parameter angegeben werden. Wird explizit keiner dieser Parameter spezifiziert, wird für beide ein Defaultwert von 1.0 verwendet. Konkrete Subklassen müssen lediglich die von *AbstractBinaryFunction* definierte, abstrakte Methode

public abstract Double evaluate(Integer[] elements, double weightingAgree, double weightingDisagree);

implementieren. Der Array *elements* hat die Länge vier und speichert die Werte für a, b, c und d, die die Anzahl der Übereinstimmungen bzw. Nicht-Übereinstimmungen der zwei Objekte, entsprechend der Tabelle 2.2 aus Kapitel 2.2.2, angeben.

Die konkreten Funktions-Klassen

Nachdem die abstrakten Klassen eingeführt wurden, werden in Tabelle 9.4 die konkreten Distanz- bzw. Ähnlichkeitsfunktionen vorgestellt. Die erste Spalte beinhaltet den Namen der jeweiligen Klasse und die zweite Spalte das Kürzel der Funktion, das in der Eingabefunktion spezifiziert werden muss. **Dabei steht x jeweils für eine Liste $\{x_1, ..., x_i, ..., x_n\}$ von Variablen** mit $n \epsilon \mathbb{N}$ und $i \epsilon \{1, ..., n\}$. Die Anzahl n von Variablen muss - aus oben genannten Gründen - gerade sein. Die dritte Spalte beschreibt, welche mathematische Formel von der Funktion ausgeführt wird. Die mathematischen Formeln für intervall-skalierte Variablen (die ersten 4 Einträge der Tabelle) können in Kapitel 2.2.1 nachgelesen werden. Bei der Beschreibung der nominalen Funktionen steht m für die Gesamtanzahl der Variablen und a für die Anzahl der Übereinstimmungen zweier Objekte in den Variablen. Bei der Beschreibungen der binären Variablen entspricht die Bedeutung der Bezeichner a, b, c und d der Bedeutung in Tabelle 2.2 aus Kapitel 2.2.2.

Die vier Funktionen für binäre Variablen decken - über die entsprechende Belegung der Gewichtungs-Parameter - alle in Tabelle 2.3 und 2.4 aus Kapitel 2.2.2 angegebenen Funktionen ab. Werden für die Gewichtungs-Parameter keine Werte angegeben, ergibt sich - durch die Default-Belegung der Gewichtungs-Parameter mit dem Wert 1.0 - für asymmetrisch binäre Variablen der *Jaccard Koeffizient* und für symmetrisch binäre Variablen der *Simple Matching Koeffizient*. Außerdem kann durch Variation der Gewichtungs-Parameter - genauso wie bei den Funktionen für nominale Variablen - mit einer Fülle zusätzlicher Distanz- bzw. Ähnlichkeitsfunktionen experimentiert werden.

Name	Kürzel	Beschreibung
EuclidianDistance	ed(x)	Euklidische Distanz
QuadraticEuclidianDistance	qed(x)	Quadratische Euklidische Distanz
ManhattanDistance	md(x)	Manhattan Distanz
CosineSimilarity	cs(x)	Cosine Similarity
NominalDistance	nomd(x,w)	$w * \frac{(m-a)}{m}$
NominalSimilarity	noms(x,w)	$w * \frac{a}{m}$
AsymmetricBinaryDistance	abind(x,$w1$,$w2$)	$\frac{w2*(b+c)}{w1*a+w2(b+c)}$
AsymmetricBinarySimilarity	abins(x,$w1$,$w2$)	$\frac{w1*a}{w1*a+w2(b+c)}$
SymmetricBinaryDistance	sbind(x,$w1$,$w2$)	$\frac{w2*(b+c)}{w1*(a+d)+w2(b+c)}$
SymmetricBinarySimilarity	sbins(x,$w1$,$w2$)	$\frac{w1*(a+d)}{w1*(a+d)+w2(b+c)}$

Tabelle 9.4: Erweiterte Funktionenen

Referenzieren einzelner Variablenwerte

In diesem Abschnitt soll auf die Referenzierung einzelner Variablenwerte eingegangen werden. Dies soll anhand zweier Objekte (Personen) mit der Variable Gewicht und der konkreten Belegungen 69 für Gabi und 86 für Klaus erläutert werden (siehe Abbildung 9.6).

Abbildung 9.6: Objektdiagramm zweier Personen

Die in Tabelle 9.4 spezifizierten Funktionen erhalten Variablen als Parameter und berechnen die Distanz zweier Objekte o_1 und o_2. Jedes dieser Objekte hat einen Wert für eine Variable. Nimmt man als Beispiel die Funktion der Euklidischen Distanz $ed(x_1, ..., x_i, ..., x_n)$ und als Variable das Gewicht, ergibt sich für oben aufgeführtes Beispiel der Funktionsaufruf $ed(Gabi.Gewicht, Klaus.Gewicht)$. Die Variable $Gabi.Gewicht$ speichert den Wert 69 für Gabi und die Variable $Klaus.Gewicht$ den Wert 86 für Klaus.

Um genau einen Variablenwert zu referenzieren, muss man also das jeweilige Objekt ansprechen, z.B. *Gabi.Gewicht*. Da der Name (die *id*) eines Objektes in der Regel nicht verfügbar ist und man außerdem eine Formel für beliebige Objekte erstellen möchte, muss man die Referenz auf ein Objekt allgemeiner formulieren. Das Ansprechen eines Variablenwertes eines konkreten Objektes wurde mit dem Präfix *o1.* bzw. *o2.* realisiert. Eine konkrete Variable eines Objektes kann also mit *o1.<Variablenname>* bzw. *o2.<Variablenname>* angesprochen werden.

Ein verallgemeinerter Funktionsaufruf der Euklidischen Distanz für jeweils zwei Objekte $o1$ und $o2$ einer Datentabelle lautet folgendermaßen:

ed(o1.Gewicht, o2.Gewicht)

Dies gilt auch bei der freien Eingabe einer Formel zur Distanz- bzw. Ähnlichkeitsberechnung, ohne Rückgriff auf bekannte Funktionen. Möchte man z.B. eine Distanz zwischen Gabi und Klaus durch Aufsummieren der Werte der Variable Gewicht erhalten, muss man sich auf einzelne Objekte beziehen können. Die gewünschte Eingabefunktion zur Aufsummierung der Variable Gewicht lautet folgendermaßen:

o1.Gewicht + o2.Gewicht

Beispiele von Eingabefunktionen

Um dem Leser einen Einblick in mögliche Eingabefunktionen zu gewähren, werden in diesem Abschnitt drei Beispielfunktionen spezifiziert, die der Parser verarbeiten kann. Alle angegebenen Beispielfunktionen beruhen auf dem Schema aus Tabelle 9.5, die nur die Variablen der Personen, aber keine konkreten Werte definiert. Als Eingabe erhält eine Funktion immer die Variablenwerte zweier Objekte $o1$ und $o2$, die - wie oben bereits eingeführt - mit *o1.* und *o2.* angesprochen werden können.

Name	Gewicht	Größe	Haarfarbe	Geschlecht	Raucher

Tabelle 9.5: Schema einer Datentabelle

Folgende Funktion berechnet die *Quadratische Euklidische Distanz* der Variablen Gewicht und Größe und teilt diese durch die Anzahl der Variablen zwei. Anschließend wird die nominale Distanz (ohne Gewichtung) der Variable Haarfarbe, die symmetrisch binäre Distanz der Variable Geschlecht (*Simple Matching Koeffizient*) und die asymmetrisch binäre Distanz der Variable Raucher (*Jaccard Koeffizient*) hinzuaddiert:

qed(o1.Gewicht, o2.Gewicht, o1.Größe, o2.Größe) / 2 +
nomd(o1.Haarfarbe, o2.Haarfarbe) +
sbind(o1.Geschlecht, o2.Geschlecht) +
abind(o1.Raucher, o2.Raucher)

Die nächste Funktion ist eine Modifikation der vorherigen Funktion. Für die Variable Haarfarbe werden die Nicht-Übereinstimmung doppelt gewichtet und für die Variable Geschlecht wird der Koeffizient von *Sokal and Sneath* verwendet (doppelte Gewichtung der Übereinstimmung):

qed(o1.Gewicht, o2.Gewicht, o1.Größe, o2.Größe) / 2 +
nomd(o1.Haarfarbe, o2.Haarfarbe, 2.0) +
sbind(o1.Geschlecht, o2.Geschlecht, 2.0, 1.0) +
abind(o1.Raucher, o2.Raucher)

Folgende Funktion berechnet die *Euklidische Distanz* für die Variable Größe, die Differenz zwischen den Werten der Variable Gewicht geteilt durch zwei, die Distanz des *Simple Matching Koeffizienten* der Variable Geschlecht multipliziert mit vier und die nominale Distanz der Variable Haarfarbe. Diese Werte werden aufsummiert. Anschließend wird der Wert fünf abgezogen, falls beide Objekte Raucher sind (true wird als 1 ausgewertet):

```
 ed(o1.Größe, o2.Größe) +
abs(o1.Gewicht-o2.Gewicht) / 2 +
sbind(o1.Geschlecht, o2.Geschlecht) * 4 +
nomd(o1.Haarfarbe, o2.Haarfarbe) -
(o1.Raucher*o2.Raucher) * 5
```

Diese Funktion ist mit Vorsicht einzusetzen, da die Distanz hier negativ werden könnte. Es findet zwar auch mit negativen Distanzen eine korrekte Gruppierung statt, einige weiterverarbeitende Verfahren erwarten jedoch ausschließlich Distanz- bzw. Ähnlichkeitswerte größer Null. Es ist Aufgabe des Anwenders dies sicherzustellen.

Anbindung an das „Softwareclustering“-Plugin

Die Klasse *FlexibleDistanceOrSimilarity* ist die einzige Klasse des „Softwareclustering“-Plugin, die mit dem `JEP`-Parser kommuniziert und diesen kennt (siehe auch Abbildung 9.5). Die Aufruf-Schnittstelle des Parsers bildet die Klasse *JEP*. Alle im Folgenden aufgeführten Methoden befinden sich in dieser Klasse. Um das Ergebnis einer Funktionsberechnung zu erhalten, müssen folgende Schritte durchgeführt werden:

1. Zuerst wird ein Objekt der Klasse *JEP* erzeugt.
2. Über die Methode

   public void addFunction(String name, PostFixCommand function);

   müssen neue Funktionen registriert werden. Der Übergabeparameter *name* definiert das Kürzel der Funktion, das in der Eingabefunktion angegeben werden muss, um die Funktion zu aktivieren. *PostFixCommand* ist Oberklasse aller Klassen, die Funktionen implementieren. Hier muss die konkrete Klasse der Funktion angegeben werden.
3. Über die Methode

   public void addVariable(String name, Object object);

   müssen alle Variablen, die in der Eingabefunktion vorkommen, registriert werden. Der Parameter *name* spezifiziert den Namen der Variable, *object* gibt den initialen Wert der Variable an.
4. Durch Aufrufen der Methode

public void parseExpression(String expression_in);

wird die Eingabefunktion geparst und auf Fehler geprüft. Über entsprechende Methoden kann abgerufen werden, ob und wo in der Funktion sich Fehler, wie bspw. nicht registrierte Variablen, befinden.

5. Über die Methode

   public void setVarValue(String name, Object val);

   kann der Wert der Variable *name* auf *val* gesetzt werden.

6. Über die Methode

   public double getValue();

   kann das Ergebnis der Funktionsberechnung abgefragt werden. Das Ergebnis wird immer für die aktuelle Variablenbelegung ausgewertet.

Die Schritte eins bis vier können als Initialisierung des Parsers angesehen und dementsprechend einmalig durchgeführt werden. Die Schritte fünf und sechs werden für jeweils zwei Objekte der Datentabelle wiederholt durchgeführt, bis die Distanzen bzw. Ähnlichkeiten für alle Objektpaare berechnet wurden.

9.1.4 Der Operator *ExtendedExampleSet2Similarity*

Anforderungen

Der in Abschnitt 9.1.2 vorgestellte Operator *DistanceOrSimilarityCalculator* erlaubt die freie Eingabe einer Funktion zur Distanz- bzw. Ähnlichkeitsberechnung. Liegen eine Vielzahl von Variablen vor, müssen diese alle einzeln referenziert werden, was eine mühevolle Aufgabe sein kann. Möchte man zudem sowieso alle Variablen mit einer einheitlichen Distanz- bzw. Ähnlichkeitsfunktion behandeln, wünscht sich der Benutzer die Auswahl einer bekannten Funktion aus einer Menge vorgegebener Funktionen. Wie in Abschnitt 9.1.1 beschrieben, wird genau dies durch den bestehenden `Yale`-Operator *ExampleSet2Similarity* realisiert. Dieser Operator bietet allerdings nur die vordefinierten Funktionen *EuclidianDistance*, *ManhattanDistance* und *CosineSimilarity* an.

Die Clusteranalyse-Umgebung wird für das Softwareclustering entwickelt. In dieser Teildisziplin der Clusteranalyse treten häufig binäre Variablentypen auf. Es soll ein Operator entwickelt werden, der die Berechnung einer Distanzmatrix mit dem *Jaccard Koeffizienten* und dem *Simple Matching Koeffizienten* (siehe Abschnitt 2.2.2, Tabelle 2.3 bzw. Tabelle 2.4) ermöglicht. Zusätzlich sollen die bereits vom Operator *ExampleSet2Similarity* angebotenen Distanz- bzw. Ähnlichkeitsfunktionen ausgewählt werden können.

Entwurf

Da der neue Operator *ExtendedExampleSet2Similarity* dieselben Berechnungen ausführt wie der bestehende Operator *ExampleSet2Similarity*, wird der neue Operator als Subklasse des bestehenden Operators realisiert. Abbildung 9.7 zeigt die Datentypen und die Schnittstellen des Operators *ExtendedExampleSet2Similarity*, die mit den Datentypen und Schnittstellen des Operators *ExampleSet2Similarity* übereinstimmen.

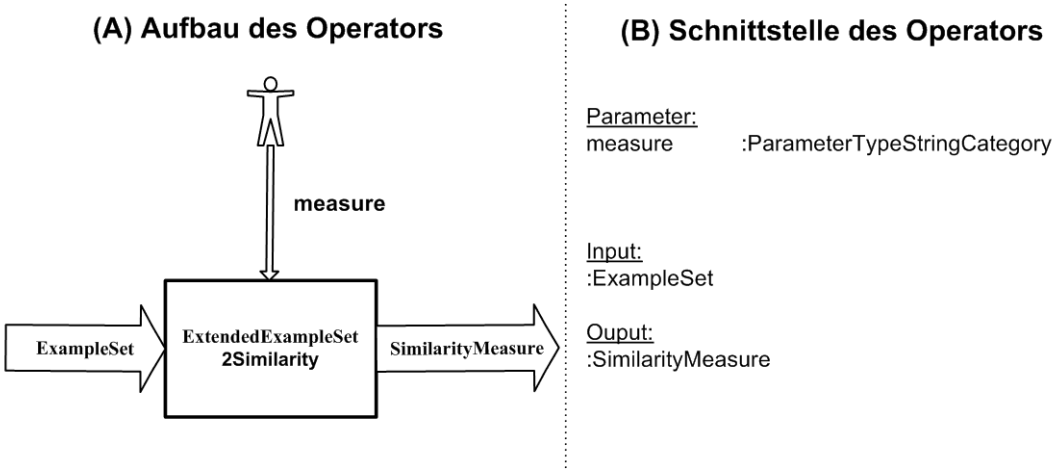


Abbildung 9.7: Aufbau (A) und Schnittstelle (B) des Operators *ExtendedExampleSet2Similarity*

Parameter
Der Parameter „measure“ vom Typ *ParameterTypeStringCategory* erlaubt die Auswahl eines Strings aus einer Liste vorgegebener Strings. Bei diesen Strings handelt es sich um die Namen der Klassen, die die jeweilige Distanz- bzw. Ähnlichkeitsfunktion realisieren. Es müssen also zwei neue Klassen implementiert werden, die die Berechnung der *Jaccard Distanz* und der *Simple Matching Distanz* realisieren. Diese Klassen werden *JaccardDistance* und *SimpleMatchingDistance* genannt und von der Klasse *BasicExampleBasedSimilarity* (siehe Abschnitt 9.1.2, Abbildung 9.3) abgeleitet. Von dieser Klasse werden alle bestehenden Klassen zur Distanz- bzw. Ähnlichkeitsberechnung, wie z.B. die Klasse *EuclidianDistance*, abgeleitet.

Die neuen Klassen *JaccardDistance* und *SimpleMatchingDistance* müssen der Liste des Parameters „measure“ hinzugefügt werden, damit diese vom Benutzer ausgewählt werden können.

9.2 Linkage-Kriterien

Wie in Kapitel 2.3.2 bereits beschrieben, zeichnen sich hierarchische Verfahren zur Clusteranalyse vor allem dadurch aus, dass eine bestimmte Anzahl an Clustern im Voraus nicht festgelegt werden muss, sondern diese auf Basis des Ergebnisses des hierarchischen Verfahrens flexibel bestimmt werden können. Um dies zu erreichen verlangen hierarchische Verfahren gegenüber anderen Clusterverfahren einen zusätzlichen Parameter, das so genannte Linkage-Kriterium, das angibt, zu welchem Distanz- / Ähnlichkeitswert zwei Cluster vereint werden (agglomerativ) bzw. ein Cluster in zwei Cluster aufgeteilt wird (divisiv). Die Umsetzung der Linkage-Kriterien in `Yale` soll in diesem Kapitel untersucht und notwendige Modifikationen bzw. Erweiterungen beschrieben werden.

9.2.1 Unzulänglichkeiten

Als Linkage-Kriterien stehen Single-Linkage (Nearest Neighbor) und Complete-Linkage (Furthest Neighbor) zur Verfügung, welche wohl die bekanntesten Kriterien darstellen. Allerdings sind beide als ein Extrem unter den Linkage-Kriterien anzusehen, da im ersten Fall die geringste Distanz zweier Objekte der Cluster als Kriterium herangezogen wird und im zweiten Fall die größte Distanz. Das führt dazu, dass die Anwendung der beiden Kriterien auf denselben Datensatz zu extrem unterschiedlichen Ergebnissen führen kann. Wünschenswert, vor allem zur experimentellen Anwendung hierarchischer Clusterverfahren, wäre ein Linkage-Kriterium, das die durchschnittliche Distanz aller Objekte des ersten und des zweiten Cluster berechnet, also das Average-Linkage-Kriterium (siehe Kapitel 2.3.2).

Die Berechnungen der Linkage-Kriterien basieren auf den zuvor berechneten Distanz- / Ähnlichkeitswerten der einzelnen Objekte der Datentabelle. Liegt der Berechnung des Single-Linkage z.B. ein Ähnlichkeitsmaß zugrunde, müssen die zwei Objekte der zwei Cluster mit der **größten** Ähnlichkeit gefunden werden. Liegt ein Distanzmaß vor, müssen die zwei Objekte der zwei Cluster mit der **geringsten** Distanz gefunden werden. In beiden Fällen werden also, aufgrund des Single-Linkage-Kriteriums, die Objekte gesucht, die sich am „nächsten“ sind. Die Berechnungen eines hierarchischen Clusterverfahrens hängen ebenso davon ab, ob ein Distanz- oder Ähnlichkeitsmaß vorliegt. Da Transformationen existieren, um aus Distanzmaßen Ähnlichkeitsmaße zu konstruieren und umgekehrt (siehe Kapitel 2.2), müssen zu Beginn der Berechnungen eines hierarchischen Clusterverfahrens lediglich die Art des vorliegenden Maßes bestimmt werden und dies, falls notwendig, in ein festgelegtes Maß transformiert werden, auf dem die Berechnungen der implementierten Linkage-Kriterien und Clusterverfahren basieren.

Dies wurde vom Autor des Clusteranalyse-Plugin mithilfe der Klasse *DistanceSimilarityConverter*, die ein übergebenes Ähnlichkeitsmaß in ein Distanzmaß transformiert und umgekehrt, auch so praktiziert und die Berechnungen der Linkage-Kriterien und Clusterverfahren auf ein **Ähnlichkeitsmaß** abgestimmt. Allerdings ist die implementierte Transformationsfunktion falsch. Folgender Auszug aus dem Originalcode der Klasse *DistanceSimilarityConverter* zeigt die Methode *similarity(String x, String y)*, die die Transformation eines

Distanzmaßes in ein Ähnlichkeitsmaß und umgekehrt realisieren soll. Auch die Auskommentierung des mittleren Teils findet sich so im Quellcode. Zunächst wird der Distanz- bzw. Ähnlichkeitswert (hier noch unbekannt) der Objekte *x* und *y* aus dem Objekt *sim* ausgelesen und in der Variable *v* gespeichert. Vor der Rückgabe wird lediglich das Vorzeichen der Variable *v* geändert.

```
    public double similarity(String x, String y) {

        double v = sim.similarity(x,y);

//        if(sim.isDistance())
//            return 1.0 - (v /( 1 + v ));
//        else
//            return 1.0 - v;

        return -v;
  }
```

D.h. der Autor legt hier eine Transformationsfunktion von

$$s_{i,j} = -d_{i,j} \tag{9.3}$$

und

$$d_{i,j} = -s_{i,j} \tag{9.4}$$

für ein Ähnlichkeitsmaß s und ein Distanzmaß d zweier Objekte i und j zugrunde. Infolgedessen sind einige das Distanz- und Ähnlichkeitsmaß betreffende Axiome nicht mehr erfüllt (siehe Axiome 2.1-2.6 in Kapitel 2.2). Trotzdem ergeben sich dadurch keine Fehleinteilungen der Objekte in Cluster[5], da die relativen Abstände der Werte zueinander entscheidend sind und diese bei Vorzeichenwechsel erhalten bleiben. Außerdem sind die im `Yale`-Plugin implementierten Linkage-Kriterien und Clusterverfahren auf diese Transformationsfunktion abgestimmt.

9.2.2 Modifikationen bzw. Erweiterungen

Die oben angesprochenen Unzulänglichkeiten der Linkage-Kriterien sollen behoben werden. Obwohl sich, wie bereits angesprochen, durch die falsche Transformationsfunktion

[5] Dies wurde allerdings nur durch einen Vergleich der Ergebnisse des in `Yale` und `SPSS 14.0` (vgl. Kapitel 4.3.2) implementierten agglomerativen Clusterverfahrens mit gleicher Konfiguration und auf demselben Datensatz überprüft.

keine Fehleinteilung in Cluster ergeben, soll diese aus folgenden Gründen geändert werden:

1. Per Definition gelten für Distanz- / Ähnlichkeitsmaße die Axiome 2.1-2.6 (siehe Kapitel 2.2).

   Die Axiome sind bei gegebener Transformationsfunktion nicht mehr erfüllt. Allein dadurch, dass ein Wert negativ werden kann, sind 2.1 und 2.4 nicht mehr erfüllt. Da die Berechnungen von Clusterverfahren sich auf diese Axiome stützen, ist bei Nichteinhaltung die korrekte Berechnung der Clusterverfahren nicht mehr garantiert.

2. Bei hierarchischen Clusterverfahren wurde jeder Elternknoten des entstehenden Baumes bei einem bestimmten Distanz- / Ähnlichkeitswert erstellt. Einige weiterverarbeitende Verfahren der Ergebnisse einer hierarchischen Clusteranalyse führen Berechnungen auf diesen Distanz- / Ähnlichkeitswerten aus (wie z.B. das im folgenden Kapitel 9.3 vorgestellte Dendrogramm) und erwarten dabei einen nicht-negativen Distanz- / Ähnlichkeitswert. Diese Voraussetzung ist bei gegebener Implementation nicht zwangsläufig gegeben.

Entwurfsentscheidungen

Da es bei Distanz- und Ähnlichkeitswerten nicht auf die absoluten Werte, sondern auf die relativen Abstände der Werte zueinander ankommt, existieren mehrere Transformationsfunktionen (vgl. [SL77]). Für die Implementierung der Transformationsfunktion in `Yale` wird folgende Funktion aus [SL77] verwendet:

Für ein Ähnlichkeitsmaß s, ein Distanzmaß d und Objekte i und j gilt:

$$s = 1 - d_{i,j} / a, \quad mit\ a = Max\{d_{i,j}\} \tag{9.5}$$

und

$$d = 1 - s \tag{9.6}$$

Wie bereits angesprochen, sollten die zugrunde liegenden Distanz- oder Ähnlichkeitswerte vor der Berechnung durch ein Clusterverfahren und ein Linkage-Kriterium auf ein festgelegtes Distanz- oder Ähnlichkeitsmaß normiert werden. Da es sich bei fast allen Funktionen zur Berechnung von Distanzen bzw. Ähnlichkeiten von intervall-skalierten Variablen um Distanzfunktionen handelt und außerdem die Visualisierung der Ergebnisse von hierarchischen Clusterverfahren als Dendrogramm (siehe Kapitel 9.3) auf Distanzwerten beruhen, wird, im Gegensatz zu der bisherigen Implementierung, das Distanzmaß gewählt. Die Implementierung des hierarchischen Clusterverfahrens und der Linkage-Kriterien müssen also dementsprechend angepasst werden.

Entwurf

In Kapitel 7.4 wurde entschieden, ein direktes Eingreifen in den Quellcode von `Yale` zu vermeiden und jegliche Modifikationen bzw. Erweiterungen in einem Plugin zusammenzufassen. Deswegen müssen statt der Modifikation bestehender Klassen neue Klassen implementiert werden. Jedoch sollen Eigenschaften und Fähigkeiten der bestehenden Klassen, soweit möglich, übernommen werden. Abbildung 9.8 zeigt den Entwurf der Linkage-Kriterien und zugehörigen Klassen, der im Folgenden beschrieben wird. Um den Unterschied zum bisherigen Aufbau und auch den Bezug der neuen zu den bestehenden Klassen zu verdeutlichen, wird der bisherige Aufbau ebenfalls dargestellt.

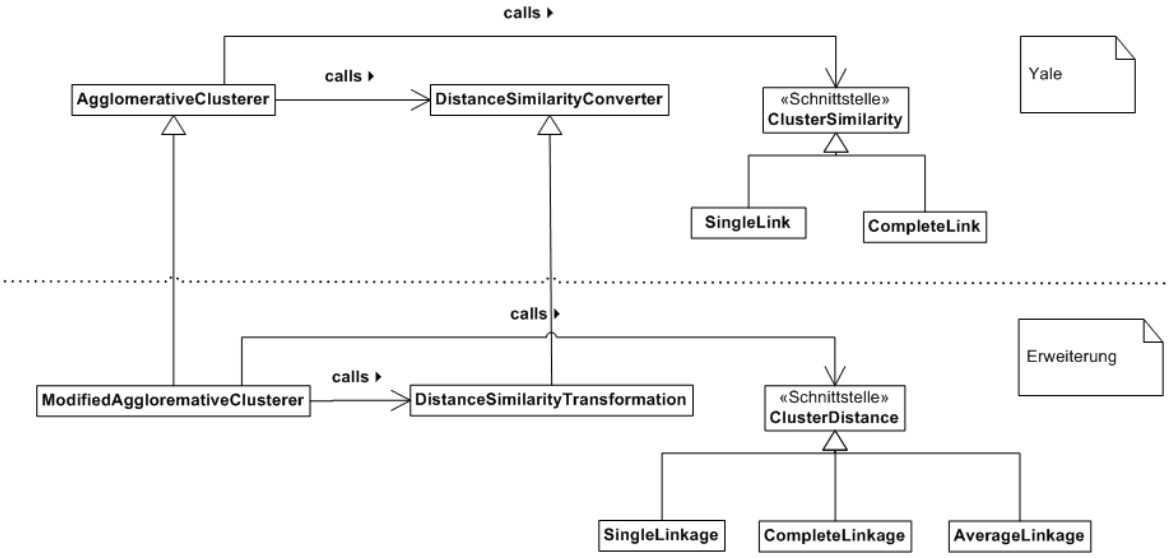


Abbildung 9.8: Entwurf Linkage-Kriterien als Klassendiagramm

Eine Klasse *DistanceSimilarityTransformation* wird als Subklasse der bestehenden Klasse *DistanceSimilarityConverter* erstellt. Die neue Klasse überschreibt lediglich die oben aufgeführte Methode *similarity(String x, String y)* und implementiert die angegebenen Transformationsfunktionen aus 9.5 und 9.6.

Die bestehende Implementation der Linkage-Kriterien beruht auf Ähnlichkeitswerten. Eine neue Implementierung, die auf Distanzwerten beruht, ist notwendig. Eine Erweiterung (im Sinne von Vererbung) der bestehenden Konzepte macht hier - aufgrund des Umsteigens von Ähnlichkeiten auf Distanzen - keinen Sinn. Es wird ein Interface *ClusterDistance* erstellt, das von jeder Klasse, die ein Linkage-Kriterium realisiert, implementiert werden muss. Dieses Interface dient in Klassen, die Linkage-Kriterien verwenden, als Stellvertreter der konkreten Linkage-Kriterien (Polymorphie). Die Klassen *SingleLinkage*, *CompleteLinkage* und *AverageLinkage* implementieren die gleichnamigen Linkage-Kriterien und führen ihre Berechnungen auf Basis von Distanzwerten aus. Der neu zu implementierende Algorithmus, der das Average-Linkage Kriterium realisiert, benötigt zur Berechnung wesentlich mehr Informationen als Single- oder Complete-Linkage (siehe Kapitel 2.3.2). Der Aufwand für eine

Distanzbestimmung zwischen zwei Clustern beläuft sich beim implementierten Algorithmus von Single- und Complete-Linkage auf $O(1)$ und beim Average-Linkage-Algorithmus auf $O(m * n)$ ($\approx O(n^2)$) für m bzw. n Objekte in Cluster 1 bzw. Cluster 2.

Es wird eine neue Operatorklasse *ModifiedAgglomerativeClusterer* als Subklasse der bestehenden Operatorklasse *AgglomerativeClusterer* erstellt. Der Algorithmus des neuen Operators wird an die Berechnungen auf Distanzwerten statt Ähnlichkeitswerten angepasst. Liegt ein Ähnlichkeitsmaß vor, werden die Werte mittels der Klasse *DistanceSimilarityTransformation* in Distanzwerte transformiert. Über einen Parameter „linkage criterion“ wird dem Operartor das zu verwendende Linkage-Kriterium übergeben und der entsprechende Subtyp von *ClusterDistance* geladen.

9.3 Visualisierung

Zur Visualisierung der Ergebnisse einer Clusteranalyse stehen in `Yale` zwei Visualisierungsmethoden zur Verfügung (siehe Kapitel 5.5), die „folder view“ (Ordneransicht) und die „graph view“ (Graphansicht).

9.3.1 Unzulänglichkeiten

Ordneransicht

Abbildung 9.9 zeigt die Ordneransicht einer partitionierenden (A) und hierarchischen (B) Clusteranalyse mit sechs Objekten. In der Ordneransicht werden - ähnlich der Visualisierung eines Dateisystems - die Cluster als Ordnersymbol und die Objekte als Dateisymbol dargestellt. Für partitionierende Verfahren, die als Ergebnis k Cluster produzieren ($k = 2$ in Abbildung 9.9 (A)), die selbst nur Objekte und damit keine weiteren Cluster enthalten, stellt sich diese Visualisierungsmethode als geeignet heraus. Die Tiefe des Ordnerbaumes ist bei solchen Verfahren konstant auf drei Ebenen (inklusive des Wurzelordners, der das Ergebnis als Ganzes darstellt) beschränkt. Abbildung 9.9 (B) lässt jedoch bereits erahnen, dass die Ordneransicht als Visualisierungsmethode für hierarchische Clusteranalysen eher ungeeignet ist. Da bei hierarchischen Verfahren Cluster wiederum Cluster als Elemente beinhalten können, steigt die Tiefe des Baumes mit der Anzahl zu gruppierender Objekte rapide an[6]. Im Beispiel wurde eine Clusteranalyse auf sechs Objekten ausgeführt, in der Praxis sind Objekte in einer Größenordnung von mehreren Hundert durchaus realistisch. Damit ist eine Übersichtlichkeit der Ordneransicht nicht mehr gegeben.

Graphansicht

Abbildung 9.10 zeigt die Graphansicht des Ergebnisses der hierarchischen Clusteranalyse aus Abbildung 9.9. In einer Baumdarstellung werden Cluster als Knoten dargestellt, klickt man auf einen Knoten, werden die im Subbaum des Knoten enthaltenen Objekte in einem Fenster rechts neben dem Baum angezeigt. Die Knoten sind über Kanten verbunden, die symbolisieren, welche Cluster zu einem neuen Cluster vereint wurden (die Richtung der Pfeile ist irreführend, die Pfeilspitzen müssten sich auf der anderen Seite des Pfeils befinden). Man kann den Abstand der Knoten skalieren und den Baum heran- und herauszoomen. Doch auch diese Darstellung wird bei steigender Anzahl von Objekten schnell unübersichtlich. Ein entscheidender Nachteil ist, dass die Lage der Knoten zueinander, nichts über den Distanzwert aussagen, bei dem zwei Cluster zu einem neuen Cluster vereint wurden. Zudem gibt die Notierung „no description available“ an jedem Knoten (wie auch an jedem Ordner der Ordneransicht) keine neuen Informationen und macht die Visualisierung nur unübersichtlich.

[6]Die Anzahl c der Cluster beträgt $c = 2n - 1$ bei n Objekten.

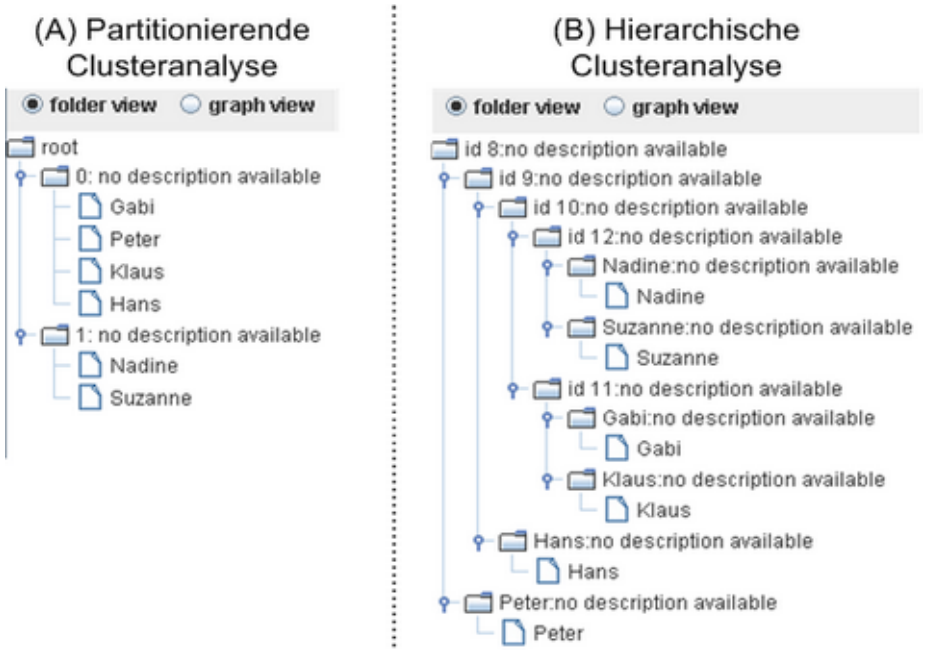


Abbildung 9.9: Ordneransicht einer partitionierenden (A) und hierarchischen (B) Clusteranalyse

Die Graphansicht einer partitionierenden Clusteranalyse besteht aus einem Wurzelknoten mit k Kindknoten, die jeweils die ihnen zugehörigen Objekte beinhalten. Die Baumdarstellung der Ergebnisse dieser Verfahren hat also eine konstante Tiefe von zwei Ebenen und bleibt damit übersichtlich. Allerdings stört auch hier die angesprochene Notierung.

Zusammenfassend lässt sich festhalten, dass keine geeignete Visualisierungsmethode für hierarchische Clusterverfahren vorhanden ist. Für eine angemessene Visualisierung von Ergebnissen partitionierender Clusterverfahren sind die zur Verfügung stehenden Visualisierungsmethoden ausreichend.

9.3.2 Modifikationen bzw. Erweiterungen

Die Visualisierung von Ergebnissen hierarchischer Clusterverfahren gestaltet sich aufgrund der Vielzahl von entstehenden Clustern als schwierig. Eine wichtige Information die man der Darstellung des Ergebnisses hierarchischer Clusterverfahren entnehmen möchte, ist die Rückverfolgung des Algorithmus, d.h. konkret wann (in welcher Reihenfolge und zu welchem Distanz- / Ähnlichkeitswert) welche Cluster zu einem Cluster vereint (agglomerativ) bzw. ein Cluster in zwei Cluster geteilt (divisiv) wurde.

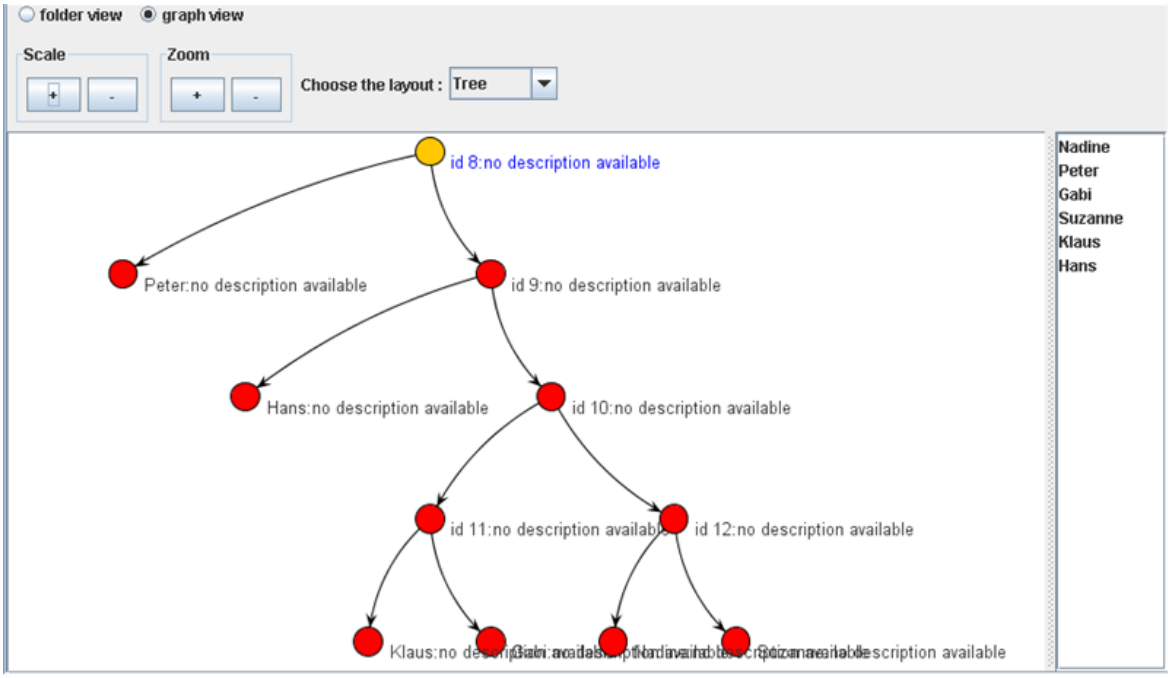


Abbildung 9.10: Graphansicht einer hierarchischen Clusteranalyse

Entwurfsentscheidungen

In der Praxis hat sich das Dendrogramm (siehe Kapitel 2.3.2) als Visualisierungsmethode für hierarchische Clusterverfahren durchgesetzt, gerade weil es die eben beschriebenen Anforderungen erfüllt und selbst bei einer Vielzahl von Objekten die Struktur des entstehenden Clusterbaumes und damit die Reihenfolge der Vereinigung der Cluster und deren relativen Abstand noch erkennen lässt. Das Dendrogramm soll also als zusätzliche Visualisierungsmethode für hierarchische Clusterverfahren eingeführt werden.

Entwurf

Wie in Kapitel 7.4 eingangs kurz beschrieben, werden von `Yale` nur die Operatorklassen eines Plugin direkt geladen. Deshalb muss ein Operator implementiert werden, der die neue Visualisierungsmethode ermöglicht. Die Visualisierung in `Yale` funktioniert so, dass alle *IOObject*s, die sich nach Ausführung eines Experimentes im *IOContainer* befinden, von der Klasse *ResultDisplay* visualisiert werden. Dazu implementiert jedes konkrete *IOObject* eine Methode

public Component getVisualisationComponent();

die von *ResultDisplay* aufgerufen wird und die zu visualisierende GUI-Komponente zurückgibt. Abbildung 9.11 zeigt diesen Ablauf als Sequenzdiagramm am Beispiel der Visua-

lisierung des Ergebnisses einer hierarchischen Clusteranalyse. Die beteiligten Objekte bzw. deren Klassen werden im folgenden Absatz erläutert.

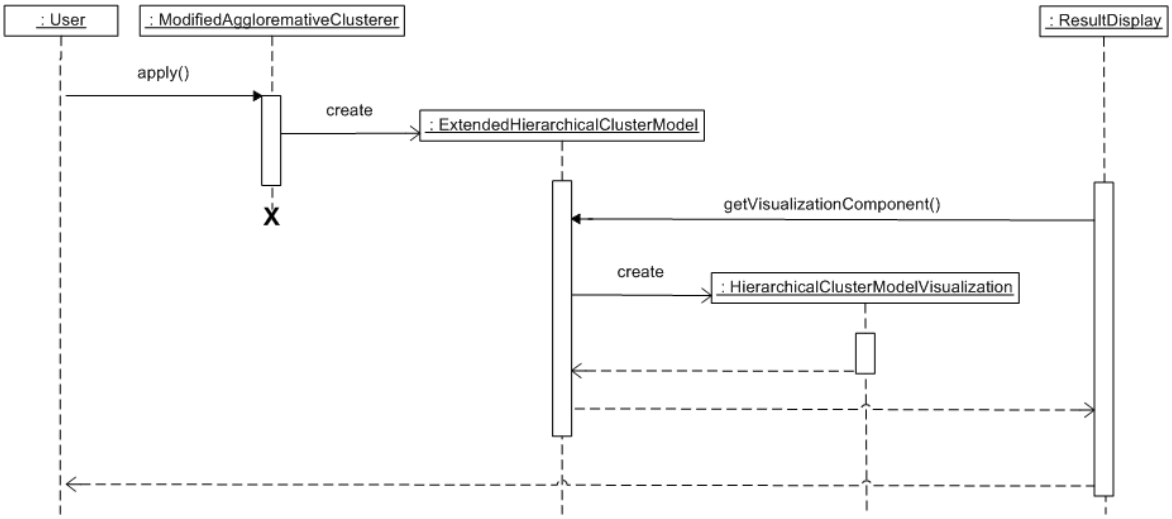


Abbildung 9.11: Sequenzdiagramm der Visualisierung von *IOObject*s am Beispiel der modifizierten hierarchischen Visualisierungsmethoden

Der neu zu implementierende Operator wird *ModifiedAgglomerativeClusterer* genannt und erbt von dem bestehenden Operator *AgglomerativeClusterer* (siehe Abschnitt 9.2.2, Abbildung 9.8), der das agglomerative hierarchische Clusterverfahren implementiert. Es wird also der bereits in Kapitel 9.2 eingeführte Operator *ModifiedAgglomerativeClusterer* genutzt. Er führt, bis auf wenige Ausnahmen, denselben Algorithmus wie seine Oberklasse aus. Allerdings wird ein anderes *IOObject* als Output erstellt, ein Objekt vom Typ *ExtendedHierarchicalClusterModel*. Diese Klasse erbt von der bestehenden Klasse *HierarchicalClusterModel* und verwaltet dieselben Informationen wie diese, sie speichert den durch ein hierarchisches Clusterverfahren entstehenden vollen Binärbaum. Lediglich die angesprochene Methode *getVisualisationComponent()* wird überschrieben und eine neu implementierte GUI-Komponente der Klasse *HierarchicalClusterModelVisualization*, die als Visualisierungsmethode ein Dendrogramm enthält, zurückgegeben. Diese wird dann von der Klasse *ResultDisplay* visualisiert.

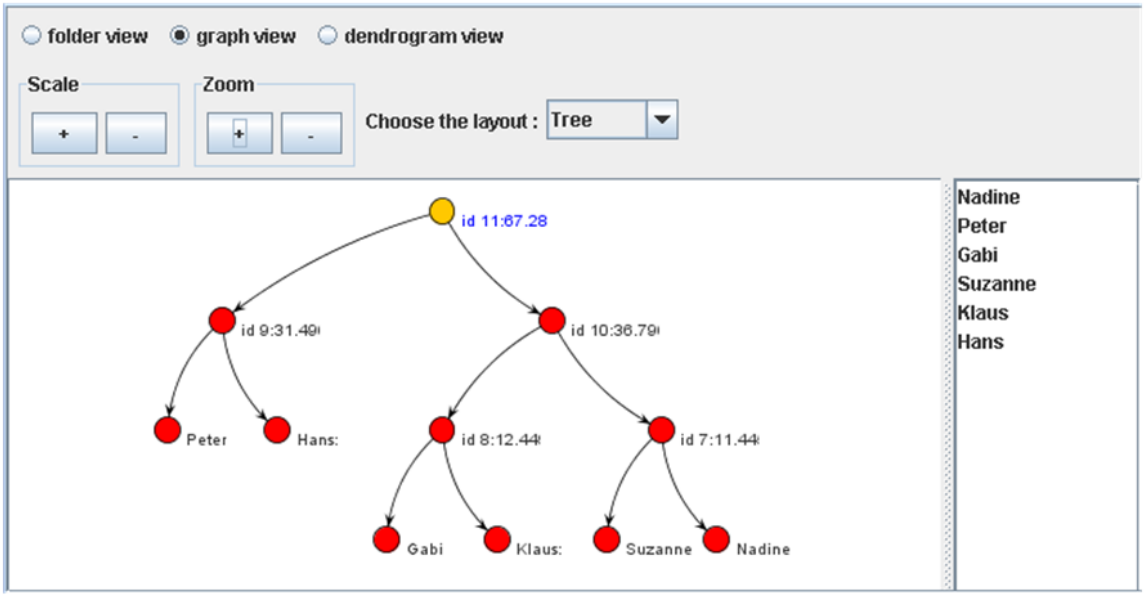
Zusätzlich zu der neu implementierten „dendrogram view“ (Dendrogrammansicht) werden die bestehende Ordneransicht und Graphansicht beibehalten. Allerdings wird hier die Notierung „no description available“ bei Clustern, die aus der Vereinigung zweier Cluster entstehen, durch den Distanzwert, bei dem diese Vereinigung bzw. Teilung stattfand, ersetzt. Blattknoten erhalten außer ihrem Namen keine zusätzliche Notierung. Abbildung 9.12 zeigt die entstehende Graph- (A) und Dendrogrammansicht (B).

Das Dendrogramm ist horizontal ausgerichtet. Über eine Zoom-Funktion kann das Dendrogramm heran- und herausgezoomt werden. Das Kontrollkästchen „zoom only y-axis“ gibt an, ob nur in y-Richtung gezoomed werden soll (Kontrollkästchen aktiviert, Defaulteinstel-

lung) oder in x- und y-Richtung (Kontrollkästchen nicht aktiviert). Jeder dargestellte Name repräsentiert ein Cluster, klickt man ein Cluster an, werden im rechten Teilfenster der Name des Cluster, die Distanz bei der das Cluster entstand bzw. geteilt wurde, die Anzahl der Objekte im Subbaum und eine Liste der Namen der Objekte im Subbaum angezeigt. Der Abstand der Cluster in horizontaler Ausrichtung (also die horizontalen Linien) repräsentiert den relativen Abstand der Cluster zueinander. D.h. das Cluster mit dem Namen „id 10“ wurde ca. bei dreifacher Distanz erstellt wie das Cluster „id 7“. So lässt sich die Anzahl der optimalen Cluster eines Ergebnisses leichter bestimmen, indem man nach großen Distanzen zwischen zwei Clustervereinigungen bzw. -teilungen sucht (siehe dazu Kapitel 2.3.2). Die Eigenschaft der relativen Distanzen bleibt auch bei sehr großen Dendrogrammen erhalten. Zwar werden die Clusterbezeichnung unleserlich (beim Herauszoomen), doch bleibt die Struktur des Dendrogramm sichtbar. Unter dem Dendrogram befindet sich eine Anzeige „Distance-Interval“, die das Intervall der Distanzwerte von null bis zur maximalen Distanz angibt.

Betätigt man in der Dendrogrammansicht die rechte Maustaste, öffnet sich ein Popup-Menü, das das Ausdrucken des Dendrogramm und Abspeichern als png-Datei ermöglicht.

(A) Modifizierte Graphansicht einer hierarchischen Clusteranalyse



(B) Dendrogrammansicht einer hierarchischen Clusteranalyse

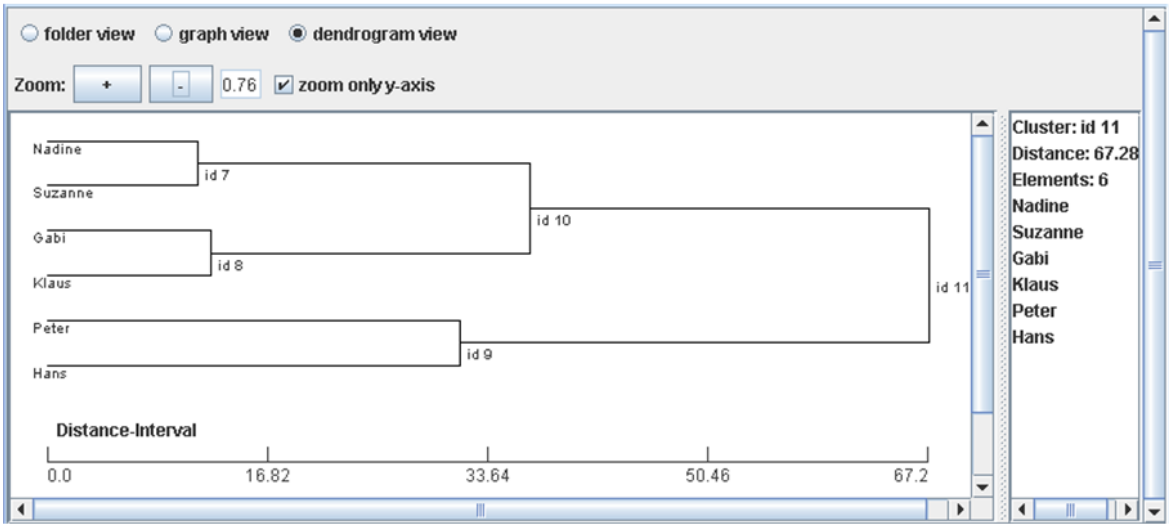


Abbildung 9.12: Modifizierte Graphansicht (A) und Dendrogrammansicht (B) einer hierarchischen Clusteranalyse

9.4 Clusterverfahren

Yale bietet eine Vielzahl von Clusterverfahren an. Tabelle 9.6 zeigt die Operatoren des Yale-Plugin „Clustering“, die ein Clusterverfahren implementieren. Dabei werden die Operatoren entsprechend der in Kapitel 2.3 vorgenommenen Systematisierung in Gruppen eingeteilt. Für jeden Operator wird angegeben, welchen Input der Operator erwartet. Als Output generieren alle hierarchischen Verfahren ein *HierarchicalClusterModel* und alle anderen Verfahren ein *FlatClusterModel*. Eine Beschreibung der Input- und Output-Objekte, der Parameter und der Implementation der Operatoren findet sich in [Wur06].

Systematisierung	**Operator**	**Input**
Partitionierende Verfahren	*KMeans*	*ExampleSet*
	SimpleKMeans	*Exampleset*
	KernelKMeans	*ExampleSet*
	XMeans	*ExampleSet*
	KMedoids	*ExampleSet*, *SimilarityMeasure*
	EM	*ExampleSet*
	FarthestFirst	*ExampleSet*
	SupportVectorClustering	*ExampleSet*
Hierarchische Verfahren	*AgglomerativeClustering*	*ExampleSet*, *SimilarityMeasure*
	UPGMAClustering	*ExampleSet*
	TopDownClustering	*ExampleSet*
Dichtebasierte Verfahren	*DBScan*	*ExampleSet*
	DBScanClustering	*ExampleSet, SimilarityMeasure*
	OPTICS	*ExampleSet*
Modellbasierte Verfahren	*Cobweb*	*ExampleSet*

Tabelle 9.6: Clusterverfahren-Operatoren in Yale

Black-Box-Test

Jeder der hier aufgelisteten Operatoren wurde nach der Methode des Black-Box-Testens auf funktionale Korrektheit geprüft. Dazu wurde ein einfacher Datensatz mit Testdaten erstellt, der die Normalfälle abdeckt. Er wurde so einfach gehalten, dass das durch Anwendung eines Clusterverfahrens entstehende Ergebnis, schnell manuell nachgerechnet und so mit dem Ergebnis eines Operators verglichen werden konnte. Datensätze, die Rand- bzw. Fehlerfälle abdecken wurden nicht verwendet.

Der *KMedoids*-Operator berechnete keine korrekte Gruppierung der Testobjekte in Cluster. Nach einigen Testläufen mit weiteren Testdaten und unterschiedlicher Konfiguration des Operators fiel auf, dass der Operator, bei n Objekten und k Ziel-Clustern, $n-(k-1)$ Objekte in ein Cluster und die verbleibenden k-1 Objekte jeweils in ein noch leeres Cluster einteilte. Diese Fehleinteilung der Objekte findet allerdings nur statt, wenn ein Distanzmaß vorliegt.

Liegt ein Ähnlichkeitsmaß vor, sind die Berechnungen des Operators korrekt. Die Ursache dafür ist eine falsche Initialisierung einer lokalen Variablen im Quellcode des Operators. Diese Variable dient als Vergleichsvariable und speichert jeweils die geringste Distanz bzw. die größte Ähnlichkeit eines Objektes zu einem Clusterzentrum. Der Initialisierungswert der Variablen wurde so hoch angesetzt, dass, im Falle von Distanzen, kein Vergleichswert kleiner als der Initialisierungswert ist. Somit werden alle Objekte einem Default-Cluster zugeordnet. Bei den Objekten, die allein in ein Cluster eingeteilt werden, handelt es sich um die *Medoids* (siehe Kapitel 2.3).

Da in Kapitel 7.4 ein direktes Eingreifen in den Quellcode von `Yale` ausgeschlossen wurde, muss zur Behebung des Fehlers ein neuer Operator erstellt werden. Dieser Operator wird *ModifiedKMedoids* genannt und in das „Softwareclustering“-Plugin gepackt. Bis auf die Behebung des angesprochenen Fehlers wurde der Quellcode, im Vergleich zur Operatorklasse *KMedoids*, nicht verändert. Nach Behebung des Fehlers wurden zusätzliche Tests mit dem neuen Operator durchgeführt. Dabei konnten keine Fehler festgestellt werden.

Operatoren im „Softwareclustering“-Plugin

Das Plugin „Softwareclustering“ enthält zwei Clusterverfahren-Operatoren.

In Kapitel 9.2 und Kapitel 9.3 wurde bereits der neu implementierte Operator *ModifiedAgglomerativeClusterer* vorgestellt, der den bestehenden Operator *AgglomerativeClustering* um Visualisierungsmethoden und zusätzliche Linkage-Kriterien erweitert. Außerdem wurde die Performanz des Algorithmus optimiert.

Im vorangehenden Abschnitt wurde der Operator *ModifiedKMedoids* vorgestellt. Die Operatorklasse ist von der Klasse *KMedoids* abgeleitet.

Ansonsten werden keine neuen Clusterverfahren-Operatoren realisiert. Das `Yale`-Plugin „Clustering“ deckt über die angebotenen Operatoren neben einigen experimentellen Clusterverfahren alle gängigen Clusterverfahren ab. Der Autor könnte sich vorstellen, einige Operatoren so zu modifizieren, dass sie als Input ein *SimilarityMeasure* erwarten, um mit dem Ergebnis des Operators *DistanceOrSimilarityCalculator* arbeiten zu können (siehe Kapitel 2.2).

9.5 Sonstige Operatoren

9.5.1 Der Operator *Hierachical2FlatClusterModel*

Anforderungen

Eine hierarchische Clusteranalyse wird oftmals durchgeführt, wenn noch keine geeignete Anzahl k von Clustern bekannt ist, in die die Objekte der Eingabetabelle eingeteilt werden sollen. Eine hierarchische Clusteranalyse teilt Objekte nicht eindeutig in ein Cluster ein, sondern die Objekte werden im Laufe des Verfahrens mehreren Clustern zugeordnet. Es wird keine feste Anzahl k an Clustern berechnet, sondern Lösungen für $1 \leq k \leq n$ (siehe Kapitel 2.3.2). Liegt das Ergebnis einer hierarchischen Clusteranalyse vor, kann man am enstehenden Baum (als Dendrogramm) geeignete Werte für k ablesen, wobei die weitere Unterteilung der Cluster unter der Schwelle k meist nicht mehr von Interesse ist.

Um dem Benutzer das Betrachten der Ergebnisse auf einem höheren Abstraktionsniveau zu ermöglichen, soll ein *HierarchicalClusterModel* in ein *FlatClusterModel* konvertiert werden, wobei der Benutzer die Anzahl k von Clustern frei wählen kann. Zusätzlich werden durch die Konvertierung eines *HierarchicalClusterModel* in ein *FlatClusterModel* die Ergebnisse von hierarchischen und nicht-hierarchischen Clusterverfahren vergleichbar gemacht.

Entwurf

Abbildung 9.13 (A) und (B) zeigt den aus den folgenden Ausführungen resultierenden Aufbau und die Datentypen der Schnittstelle des Operators *Hierachical2FlatClusterModel*.

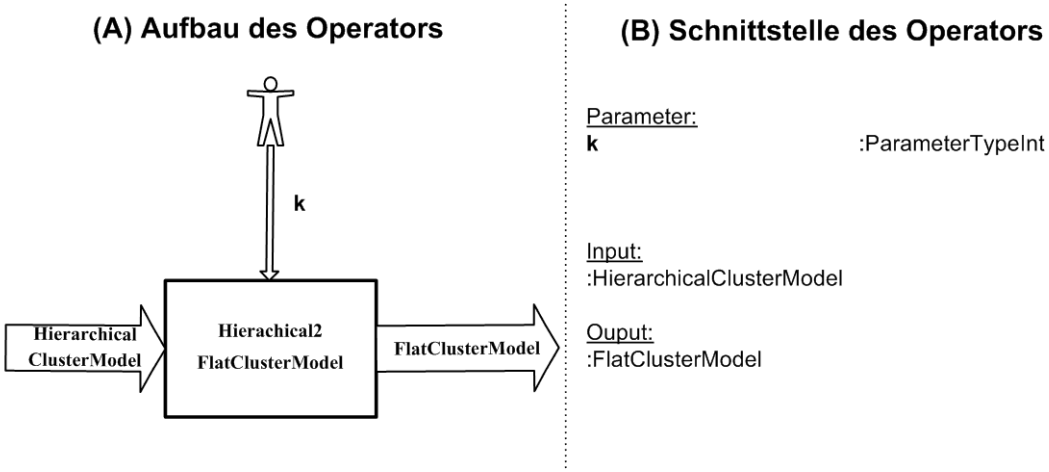


Abbildung 9.13: Aufbau (A) und Schnittstelle (B) des Operators *Hierachical2FlatClusterModel*

Parameter
Über einen Parameter mit dem Namen „k“ muss dem Operator die Anzahl k an Clustern mitgeteilt werden, die das zu erstellende Ergebnis enthalten soll. Der Parameter ist verpflichtend. Als Defaultwert wird der Parameter mit dem Wert $k = 2$ versehen.

Input
Der Operator benötigt ein *HierarchicalClusterModel* als Input. Das *HierarchicalClusterModel* speichert den Ergebnisbaum einer hierarchischen Clusteranalyse, wobei jedes Cluster durch einen Knoten repräsentiert wird, das alle Objekte, die sich in seinem Subbaum befinden, beinhaltet. Der Wurzelknoten beinhaltet somit alle Objekte der Datentabelle. Im Ergebnisbaum müssen also die k Cluster unterhalb der Wurzel ausfindig gemacht werden, die, im Falle eines agglomerativen Verfahrens, in den letzten k Iterationsschritten vereinigt wurden und, im Falle eines divisiven Verfahrens, in den ersten k Iterationsschritten geteilt wurden. Die Iterationsschritte können anhand der Distanzen, bei denen die Cluster vereinigt bzw. geteilt wurden, nachvollzogen werden. Es müssen von der Wurzel aus die k Cluster gesucht werden, die die höchsten Distanzen aufweisen. Die Distanzen bei denen ein Cluster vereinigt bzw. geteilt wurde sind im *HierarchicalClusterModel* gespeichert.

Output
Als Output erstellt der Operator ein *FlatClusterModel* mit k Clustern, wobei jedes Cluster die Objekte beinhaltet, die im *HierarchicalClusterModel* im Subbaum des jeweiligen Cluster enthalten sind.

Abbildung 9.14 (A) zeigt das Ergebnis einer hierarchischen Clusteranalyse als Dendrogramm und Abbildung 9.14 (B) die zugehörige flache Ergebnishierarchie für $k = 4$, die von *Hierachical2FlatClusterModel* aus dem gegebenen *HierarchicalClusterModel* erzeugt wurde.

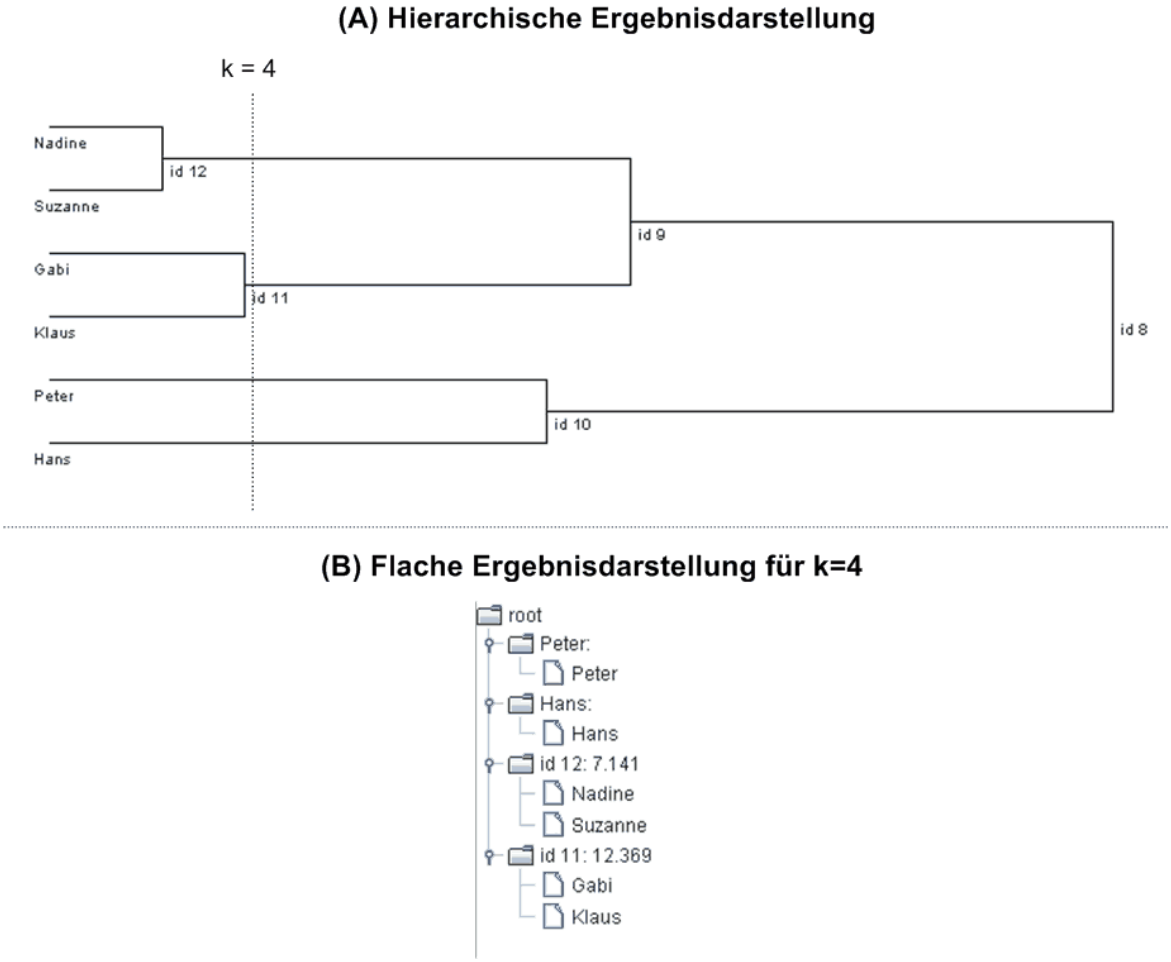


Abbildung 9.14: Ergebnisdarstellung einer hierarchischen Clusteranalyse als Baumdarstellung (A) und als flache Hierarchie mit $k = 4$ (B)

10 Beispielanwendung

In diesem Kapitel wird eine Clusteranalyse auf Softwareelementen mithilfe der erstellten Clusteranalyse-Umgebung durchgeführt. Das Kapitel zeigt die Einsatztauglichkeit der Umgebung und dient gleichzeitig als Tutorial für Benutzer der Umgebung. Zusätzlich wird dem Leser eine typische Vorgehensweise bei der Durchführung einer Clusteranalyse näher gebracht.

Der Clusteranalyse soll ein strukturiertes Vorgehen zugrunde liegen, das dem Benutzer der Clusteranalyse-Umgebung als Leitfaden dienen kann. Folgende aufeinander aufbauende Schritte werden durchgeführt:

1. Festlegen der zu untersuchenden Software (Untersuchungsgegenstand)
2. Planen der Clusteranalyse
    a) Definieren des Ziels der Clusteranalyse
    b) Auswahl der Daten
    c) Auswahl einer Distanz- bzw. Ähnlichkeitsfunktion
    d) Auswahl eines Clusterverfahrens
    e) Auswahl einer Visualisierungsmethode
3. Durchführen der Clusteranalyse
    a) Erstellen eines TGraphen aus dem Quellcode der zu untersuchenden Software und Einbinden des TGraphen in die Clusteranalyse-Umgebung
    b) Installieren und Starten der Clusteranalyse-Umgebung
    c) Einlesen der Daten in die Clusteranalyse-Umgebung
    d) Konfigurieren und Durchführen der Clusteranalyse
    e) Auswerten der Ergebnisse

Da dieses Kapitel schwerpunktmäßig das Ziel verfolgt, dem Leser den Umgang mit der Clusteranalyse-Umgebung näher zu bringen, wird im Folgenden vor allem Schritt 3 in Abschnitt 10.3 ausführlich behandelt. Die Schritte 1 und 2, die in den Abschnitten 10.1 und 10.2 behandelt werden, sollen dem Leser das typische Vorgehen beim Durchführen einer Clusteranalyse verdeutlichen und ihm die Grundlagen der in diesem Kapitel durchgeführten Clusteranalyse aufzeigen.

10.1 Der Untersuchungsgegenstand

Als Untersuchungsgegenstand der Clusteranalyse wird die C-Implementation der Software **`Cosmos-Client`** verwendet. Die Software wurde von der Arbeitsgruppe Rechnernetze der Universität Koblenz-Landau im Rahmen des Projektes `Cosmos`[1] entwickelt und wird zum Monitoring von verteilten Applikationen mit Schwerpunkt auf Client / Server-Anwendungen eingesetzt.

Der Code der Software verteilt sich auf die fünf Übersetzungseinheiten (Module):

- client.c
- server.c
- comm.c
- debug.c
- ident.c

Die Software ist dem Autor dieser Arbeit nicht bekannt. Es wurde bewusst eine dem Autor unbekannte Software gewählt, da untersucht werden soll, welche Erkenntnisse über die Software aus einer Clusteranalyse gewonnen werden können.

10.2 Planen der Clusteranalyse

Dieser Abschnitt behandelt die Planung der Clusteranalyse, worunter das Definieren des Untersuchungsziels und die Auswahl der zu verwendeten Verfahren für die Clusteranalyse (Konfiguration der Clusteranalyse), gemäß der eingangs in Kapitel 2 vorgestellten Vorgehensweise, fällt.

10.2.1 Definieren des Ziels der Clusteranalyse

Das Ziel der Clusteranalyse ist die Untersuchung der Struktur der Software `Cosmos-Client`. Dabei wird untersucht, **inwieweit die Ergebnisse der Clusteranalyse mit der tatsächlichen Modularisierung der Software übereinstimmen**.

10.2.2 Auswahl der Daten

Zur Identifikation der Struktur von Software müssen die strukturgebenden Elemente untersucht werden. Bei der vorliegenden - in der Programmiersprache C implementierten -

[1] http://www.uni-koblenz.de/∼steigner/cosmos/

Software handelt es sich um ein prozedurales System, sodass Übersetzungseinheiten, Funktionen und globale Variablen die entscheidenden strukturgebenden Elemente darstellen. Es muss also versucht werden, einen dieser Elementtypen so in Gruppen zusammenzufassen, dass sich semantisch zusammengehörende Elemente im selben Cluster wiederfinden.

Für die im Folgenden durchgeführte Clusteranalyse werden Funktionen auf Basis der Aufrufbeziehungen zu anderen Funktionen gruppiert. Es wird davon ausgegangen, dass Funktionen, die von vielen gemeinsamen Funktionen aufgerufen werden, in ein Modul eingeteilt werden sollten.

Die Aufrufbeziehungen zwischen Funktionen werden auf eine Datentabelle, genauer auf eine Objekt-Objekt-Struktur (siehe Abschnitt 2.4), abgebildet. Tabelle 10.1 zeigt die resultierende Datentabelle. In der ersten Spalte finden sich die Funktionen der Software, die in der Tabelle die aufgerufenen Funktionen (Callee) repräsentieren. Diesen Funktionen werden als Features - in der ersten Zeile der Tabelle - wiederum die Funktionen der Software zugeordnet, die aber die Aufrufer (Caller) der Funktionen der ersten Spalte repräsentieren (Matrix-Darstellung). Ein „x“ in einer Zelle der Tabelle zeigt an, dass die Funktion der jeweiligen Spalte die Funktion in der jeweiligen Zeile aufruft. Die Tabelle gibt also für alle Callee-Caller-Paare an, ob die Beziehung

Callee „wird aufgerufen von“ Caller

besteht oder nicht. In der Beispieltabelle wird die Funktion f1 von keiner (aufgeführten) Funktion aufgerufen, ruft aber selbst die Funktionen f2 und f3 auf.

Caller → **Callee** ↓	f1	f2	f3	...
f1				
f2	x			
f3	x	x		
...				

Tabelle 10.1: Input der Clusteranalyse

Die bisher beschriebene Datentabelle stellt alle Funktionen der Software `Cosmos-Client` gegenüber. Hutchens und Basili [HB85] haben die Erfahrung gemacht, dass die Reduktion der Datentabelle auf die wesentlichen Softwareelemenente und Features zu einer immensen Verbesserung und einfacheren Interpretation der Ergebnisse führt. Softwareelemente, die wenig zur Struktur der Software beitragen, können das Ergebnis in ungewollter Weise beeinflussen und eine eventuell vorhandene klare Struktur verwischen. Erste Erfahrungen des Autors mit dem Softwareclustering bestätigen diese Erkenntnisse.

Übertragen auf die vorliegenden Daten bedeutet dies, dass die Funktionen, die nur von wenigen anderen Funktionen aufgerufen werden, in der Datentabelle nicht berücksichtigt werden sollen. Es werden nur solche Funktionen als Callee in die Tabelle aufgenommen, die von **mindestens zwei verschiedenen Funktionen aufgerufen werden**. Ebenso werden nur

solche Funktionen als Caller aufgenommen, die **mindestens eine der in der Menge der Callee's enthaltenen Funktionen aufrufen**.

10.2.3 Auswahl einer Distanz- bzw. Ähnlichkeitsfunktion

Zur Distanzberechnung zwischen den Objekten der Tabelle (den Callee's) wird der ***Jaccard Koeffizient*** (siehe Abschnitt 2.2.2, Tabelle 2.4) herangezogen, weil es sich in der vorliegenden Datentabelle ausschließlich um asymmetrisch binäre Features handelt. Eine 0-0-Übereinstimmung zweier Callee's in einer Variable bedeutet im vorliegenden Fall, dass zwei Callee's vom jeweiligen Caller nicht aufgerufen werden. Dies sagt nichts über die Distanz bzw. Ähnlichkeit der beiden Callee´s aus und soll deshalb bei der Distanz- bzw. Ähnlichkeitsberechnung außen vor gelassen werden.

10.2.4 Auswahl eines Clusterverfahrens

Da eine feste Anzahl von Clustern nicht vorausgesagt werden kann und soll, sind partitionierende Verfahren ungeeignet. Außerdem soll eine hierarchische Struktur der Software aufgedeckt werden, wofür sich ein hierarchisches Clusterverfahren am Besten eignet. Es wird also das **agglomerative hierarchische Verfahren** verwendet.

Bei hierarchischen Verfahren stellt sich zusätzlich die Frage, welches Linkage-Kriterium als Join-Bedingung zweier Cluster verwendet wird. Anquetil und Lethbridge zeigten in [AL99], dass das Complete-Linkage-Kriterium im Softwareclustering am Besten geeignet ist. Dies ist dadurch zu begründen, dass sich bei Verwendung dieses Verfahrens schnell viele kleine Cluster bilden (siehe Abschnitt 2.3.2) und dadurch eine klarere Trennung verschiedener Zweige des entstehenden Clusterbaumes möglich ist. Letztgenannte Eigenschaft ist bei den vorliegenden Daten und dem definierten Anwendungsziel wünschenswert. Es wird also das **Complete-Linkage-Kriterium** verwendet.

10.2.5 Auswahl einer Visualisierungsmethode

Zur Interpretation der Ergebnisse des hierarchischen Clusterverfahrens wird das **Dendrogramm** verwendet (siehe Abschnitt 2.3.2).

10.3 Durchführen der Clusteranalyse

Nachdem die Planung der Clusteranalyse abgeschlossen ist, wird diese mithilfe der Clusteranalyse-Umgebung durchgeführt. Dieser Abschnitt beschreibt die dafür notwendigen Schritte.

10.3.1 Erstellen und Einbinden eines TGraphen

Da der Quellcode der Software nicht als Eingabe einer Clusteranalyse geeignet ist, muss dieser erst in eine geeignete Datenstruktur transformiert werden. Abbildung 10.1 zeigt vorgreifend die notwendigen Softwarekomponenten für die im Folgenden beschriebenen Arbeitsschritte und den Fluss der Dokumente zwischen diesen Komponenten.

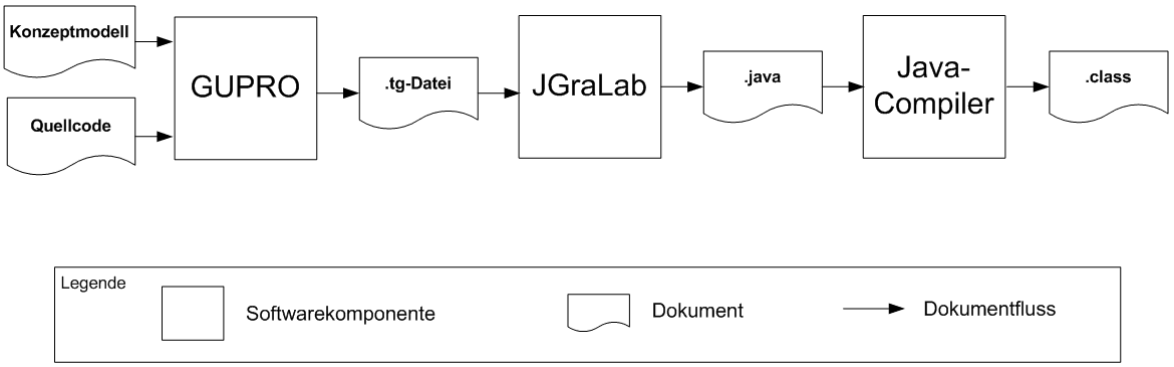


Abbildung 10.1: Erstellen eines TGraphen aus Quellcode

Zunächst muss die Software `Cosmos-Client` mit `GUPRO` geparst und aus dem Quellcode ein TGraph erstellt werden. `GUPRO` verlangt zusätzlich zum Quellcode ein Konzeptmodell (siehe Kapitel 6), das die aus dem Quellcode zu extrahierenden Informationen und das Schema des entstehenden TGraphen beschreibt, als Input. Am Institut für Softwaretechnik der Universität Koblenz-Landau wurde ein allgemeines Konzeptmodell für Programme, die in der Programmiersprache C implementiert sind, entworfen. Dieses Konzeptmodell wird zum Erstellen des TGraphen aus dem Quellcode der Software `Cosmos-Client` verwendet.

Der erstellte TGraph liegt in einer Textdatei mit der Endung .tg vor. Im konkreten Beispiel wird also die Datei „cosmos-client.tg“ erstellt. Die Datei enthält sowohl die Schemabeschreibung des Graphen, als auch die Beschreibung der extrahierten Programminformationen der Software `Cosmos-Client`, die eine konkrete Instanz der Schemabeschreibung darstellen.

Aus der Datei „cosmos-client.tg“ müssen mit der Software `JGraLab` Java-Klassen, die das Schema des Graphen beschreiben, generiert und anschließend die entstehenden .java-Dateien kompiliert werden. Dies ist notwendig, damit eine Software, die auf dem Graphen arbeiten soll, auf das Schema des Graphen in einer der Software verständlichen Sprache zugreifen kann. Eine Beschreibung dieses Vorganges kann in [Kah06] nachgelesen werden.

Damit die Clusteranalyse-Umgebung das Schema des Graphen kennt, muss eine Referenz auf die erstellten Java-Klassen hergestellt werden. Da nur das Plugin „GReQL-Interface“ mit dem TGraphen-Schema arbeitet, reicht es aus, dem Classpath-Eintrag dieses Plugins (realisiert als JAR-Datei) den Pfad zu den erstellten Java-Klassen hinzuzufügen.

10.3.2 Installieren und Starten der Clusteranalyse-Umgebung

Tabelle 10.2 zeigt die Softwarekomponenten und ihre zugehörigen Dateien, die notwendig sind, um ein Softwareclustering mit der Clusteranalyse-Umgebung durchführen zu können.

Softwarekomponente	Datei
`Yale 3.2`	yale-3.2-bin.tar.gz
Clustering-Plugin	clustering-plugin-2.2.jar
GReQL-Interface-Plugin	greql-interface.jar
Softwareclustering-Plugin	softwareclustering.jar
`GReQL2`	greql2.jar
`JEP`	jep.jar

Tabelle 10.2: Softwarekomponenten und zugehörige Dateien

Alle angegebenen Dateien sind auf der dieser Arbeit beiliegenden CD zu finden.

Zunächst muss die Datei „yale-3.2-bin.tar.gz“ entpackt werden. Die Komponenten `GReQL2` und `JEP` werden von den Plugins „GReQL-Interface“ und „Softwareclustering“ verwendet. Die Dateien greql2.jar und jep.jar müssen also dem Klassenpfad von Yale hinzugefügt werden. In dem Ordner *<yale_home>/bin* finden sich einige Start-Skripte. Diese enthalten bereits einen Eintrag *Class-Path*, dem der Pfad zu den Dateien greql2.jar und jep.jar hinzugefügt werden kann.

Beim Einbinden der Plugins „Clustering-Plugin“, „GReQL-Interface“ und „Softwareclustering“ ergab sich ein Problem, das auf eine fehlerhafte Implementation im Plugin-Konzept von `Yale` zurückzuführen ist. Eigentlich müssen die Plugins einfach in den Ordner *<yale_home>/lib/plugins* kopiert werden. Da die Plugins „GReQL-Interface“ und „Softwareclustering“ Klassen des Plugin „Clustering-Plugin“ verwenden (siehe Kapitel 8 und 9), müssen diesen Plugins die Klassen des „Clustering-Plugin“ bekannt sein. Dieses Problem wurde von den `Yale`-Autoren durch einen Eintrag *Plugin-Dependencies: pluginName [pluginVersion]* in der Manifest-Datei des JAR-Archives des abhängigen Plugins gelöst (siehe [MFK06], S.332f). Leider ist die Implementation dieses Lösungsansatzes fehlerhaft, da die Klassen des Plugin, von dem andere Plugins abhängig sind, mehrfach geladen werden[2]. Dies führt dazu, dass bei der Ausführung von `Yale` Operatoren verschiedener Plugins Objekte verschiedener Klassen verwenden (z.B. *IOObjects*) und diese Operatoren somit nicht gemeinsam in einer Operatorkette verwendet werden können. Den `Yale`-Autoren wurde das beschriebene Problem bereits mitgeteilt. Eine Behebung des Fehlers wird sich jedoch erst frühestens im nächsten Release von `Yale` befinden.

Um das Problem, das durch die *Plugin-Dependencies* entsteht, zu umgehen, werden alle drei (in der Tabelle aufgeführten) Plugins in **einer** JAR-Datei zusammengefasst. Die JAR-Datei wird **clustering.jar** genannt und muss in den Ordner *<yale_home>/lib/plugins* kopiert wer-

[2]Die Klassen werden einmal geladen, wenn das Plugin selbst geladen wird und nochmals, wenn die *Plugin-Dependencies* des / der abhängigen Plugin(s) aufgelöst werden. Die Ursache dafür liegt in einer fehlerhaften Behandlung des *ClassLoader*-Objektes in der Klasse *edu.udo.cs.yale.tools.plugin.Plugin*.

den. Dies stellt nur eine temporäre Lösung dar. Der Autor dieser Arbeit wird sich zusammen mit den `Yale`-Autoren darum bemühen, eine Lösung mit drei Plugins zu realisieren.

Eine detaillierte Anleitung zum Installieren und Ausführen der Clusteranalyse-Umgebung findet sich in der Datei „Installationshinweise.pdf“, die sich auf der dieser Arbeit beiliegenden CD befindet.

10.3.3 Einlesen der Daten in die Clusteranalyse-Umgebung

In diesem Abschnitt wird sich häufig auf GUI-Elemente bezogen. Abbildung 10.2 zeigt das Hauptfenster der Clusteranalyse-Umgebung in der Ansicht „Tree“, wobei einige Beschriftungen vorgenommen wurden, die dem Leser das Verstehen dieses Abschnittes erleichtern sollen. Die Abbildung zeigt die Oberfläche in dem Zustand, in dem bereits alle im Folgenden beschriebenen Konfigurationen vorgenommen wurden.

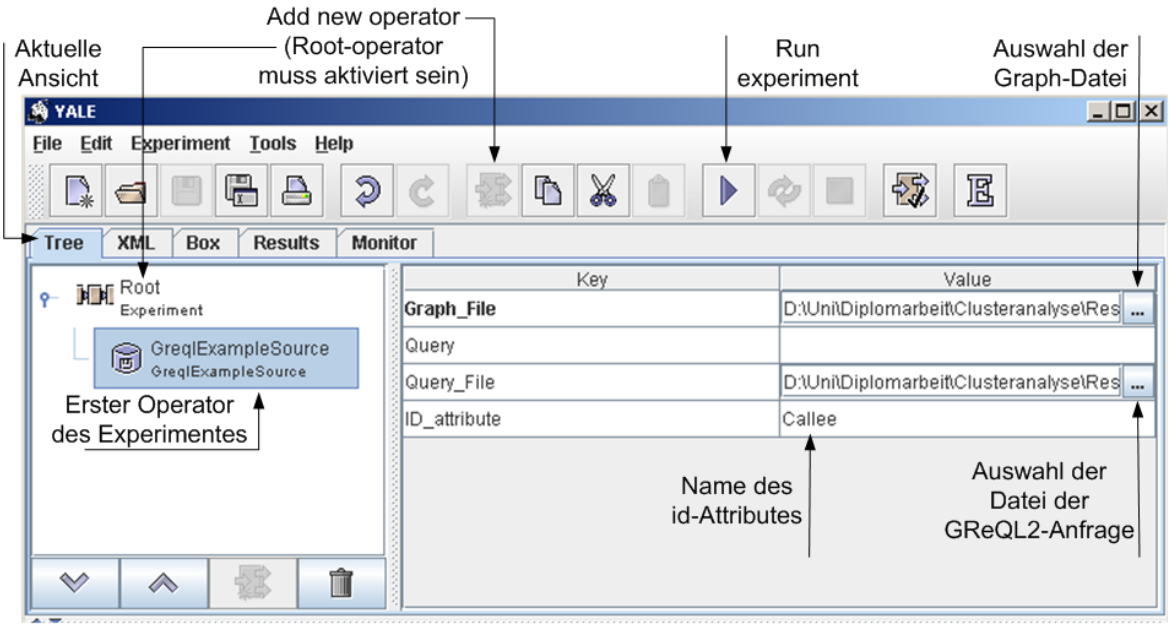


Abbildung 10.2: Konfiguration des Operators *GreqlExampleSource*

Der Operator *GreqlExampleSource*

Zum Einlesen der Daten aus dem erstellten TGraphen in die Clusteranalyse-Umgebung wird der Operator *GreqlExampleSource* (siehe Abschnitt 8.1) verwendet. Über den Menüpunkt „Add new operator“ wird eine Liste aller verfügbaren Operatoren angezeigt. Klickt man auf einen Operator, wird eine kurze Beschreibung der Funktionalität, die erwarteten Inputobjekte und die erstellten Outputobjekte des Operators angezeigt. Die Anzahl der Operatoren

in der Liste kann über verschiedene Filter reduziert werden. Der Operator *GreqlExampleSource*, der in der Operatorgruppe „IO“ zu finden ist, wird ausgewählt und somit als erster Operator des Experimentes hinzugefügt. *GreqlExampleSource* erwartet keine Inputobjekte und erstellt als Output ein Objekt vom Typ *ExampleSet*.

Der Parameter „Graph_File“

Wie in Abschnitt 8.1 beschrieben, erwartet der Operator die Angabe der .tg-Datei, die den zu ladenden Graph speichert. Über einen „File-Chooser“ wird die Datei „cosmos-client.tg“ ausgewählt und dem Operator über den Parameter „Graph_File“ übergeben.

Der Parameter „Query_File“

Außerdem erwartet der Operator eine `GReQL2`-Anfrage, die auf dem angegebenen Graph ausgeführt wird. Diese Anfrage muss so spezifiziert werden, dass sie die in Abschnitt 10.2.2 definierte Tabelle zurückgibt. Code-Listing 10.1 zeigt die `GReQL2`-Anfrage, die in der Datei „callee_caller_query.txt“ gespeichert wird. Diese Textdatei wird dem Operator *GreqlExampleSource* über den Parameter „Query_File“ übergeben. Eine Beschreibung der `GReQL2`-Syntax findet sich in [Mar06] und [Bil06].

Der Parameter „ID_attribute“

Als weiterer Parameter kann der Name eines Attributes angegeben werden, das als *id*-Attribut verwendet werden soll. Da im vorliegenden Beispiel die aufgerufenen Funktionen (Callee's) gruppiert werden sollen, wird der erste Spalteneintrag der Datentabelle, der die Namen der aufgerufenen Funktionen speichert, als *id*-Attribut deklariert. In der `GReQL2`-Anfrage wurde diesem Spalteneintrag der Name „Callee“ (siehe Code-Listing 10.1, letzter Parameter der **reportTable**-Klausel[3]) zugewiesen. Als Wert für den Parameter „ID_attribute“ wird also Callee angegeben.

Einlesen der Daten

Nun wird das Experiment über den Menüpunkt „Run experiment“ gestartet. Die Daten werden eingelesen und die Ansicht wechselt automatisch in den „Results“-Bildschirm. Abbildung 10.3 (A) zeigt einen Ausschnitt aus dieser Ansicht, wobei einige Beschriftungen zur Erläuterung der wichtigsten GUI-Elemente vorgenommen wurden.

Da der Operator *GreqlExampleSource* ein *ExampleSet* als Outputobjekt erstellt und kein weiterer Operator dieses Objekt konsumiert, wird es im „Results“-Bildschirm visualisiert.

[3] Die reportTable-Klausel wurde `GReQL2` erst nach Fertigstellung der Arbeiten [Mar06] und [Bil06] hinzugefügt. Eine Dokumentation der Klausel kann in diesen Arbeiten nicht nachgelesen werden.

```
/* record of all FunctionDefinitions and its Identifiers */
let functions :=
from funDef:V{FunctionDefinition},
     funId:V{Identifier}
with contains(funDef <--{IsFunctionDeclaratorIn}
              <--{IsDirectDeclaratorIn}+, funId)
report rec(def:funDef, identifier:funId)
end
,

/* determine functions which are called at least two times */
freqCalledFunctions :=
from fqcallee:functions
with outDegree{IsFunctionNameIn}(fqcallee.identifier) > 1
report fqcallee
end
,

/* determine the FunctionCalls of "freqCalledFunctions" */
freqFunctionCalls :=
from freqCallee:freqCalledFunctions, fc:V{FunctionCall}
with contains(freqCallee.identifier -->{IsFunctionNameIn} ,
              fc)
report fc
end
,

/* determine the callers of "freqFunctionCalls" */
freqCallers:=
from freqCaller:functions, functionCall:freqFunctionCalls
with contains(freqCaller.def <--{IsCompoundStatementIn}
              (( <--{IsStmtIn}*) | (<--{IsDeclarationIn}+
              <--{IsInitDeclaratorIn}<--{IsInitializationIn}))
              <--{IsExprIn}*, functionCall)
reportSet freqCaller
end
```

```
in

/* the query */
from caller : freqCallers, callee : freqCalledFunctions
reportTable
     /* caller's in column */
     caller.identifier.name,

     /* calle's in row */
     callee.identifier.name,

     /* the constraint for the cells */
     contains(caller.def <--{IsCompoundStatementIn}
             (( <--{IsStmtIn}*) | (<--{IsDeclarationIn}+
             <--{IsInitDeclaratorIn}<--{IsInitializationIn}))
             <--{IsExprIn}* <--{IsFunctionNameIn},
             callee.identifier),

     /* the name of the row description, here the calle's */
     "Callee"
end
```

Quellcode 10.1: GReQL2-Anfrage zum Erstellen der Tabelle aus Abschnitt 10.2.2

Hier können die eingelesenen Daten in den Sichten „meta data view“ (siehe Abbildung 10.3 (A)), „data view“ und „plot view“ betrachtet werden. Insbesondere können die eingelesenen Daten über einen „Save“-Button in einem XML-Format gespeichert werden. Dies ist in doppelter Hinsicht empfehlenswert, da die Daten bei erneutem Ausführen eines Experimentes nicht nochmals über `GReQL2` eingelesen werden müssen (was Berechnungszeit erfordert) und außerdem ein Editor existiert, der das Laden, Visualisieren und Bearbeiten von Datentabellen, die im angesprochenen XML-Format gespeichert sind, erlaubt. Die eingelesenen Daten werden in der Datei „Callee_Caller.xml“ gespeichert. Der Editor kann über das Menü „Tools“ unter dem Menüpunkt „Start Attribute Editor“ aufgerufen werden. Abbildung 10.3 (B) zeigt einen Ausschnitt der eingelesenen Datentabelle im Attribute Editor.

Betrachten der eingelesenen Daten

Die eingelesene Datentabelle enthält 17 Zeilen- und 18 Spalteneinträge. D.h. es existieren 17 Funktionen (Callee) in der Software `Cosmos-Client`, die von mindestens zwei verschiedenen Funktionen aufgerufen werden. Diese 17 Funktionen werden von insgesamt 18 verschiedenen Funktionen (Caller) aufgerufen. Über die „meta data view“ (siehe Abbildung 10.3 (A)), die z.B. anzeigt welche Attribute (die Caller) einen Wert wie häufig aufweisen (im Beispiel existieren nur die Werte *true* und *false*), kann schnell abgelesen werden, welche Caller die meisten Callee's aufrufen. Die Sicht „data view“ zeigt die vollständige Datentabelle (ähnlich der Visualisierung im Attribute Editor). In der „plot view“ können die Daten mit einigen zweidimensionalen Diagrammen visualisiert werden.

Da `GReQL2` selbst über keine Oberfläche verfügt, hat sich die Clusteranalyse-Umgebung als praktisches Visualisierungs-Tool von `GReQL2`-Anfrageergebnissen erwiesen.

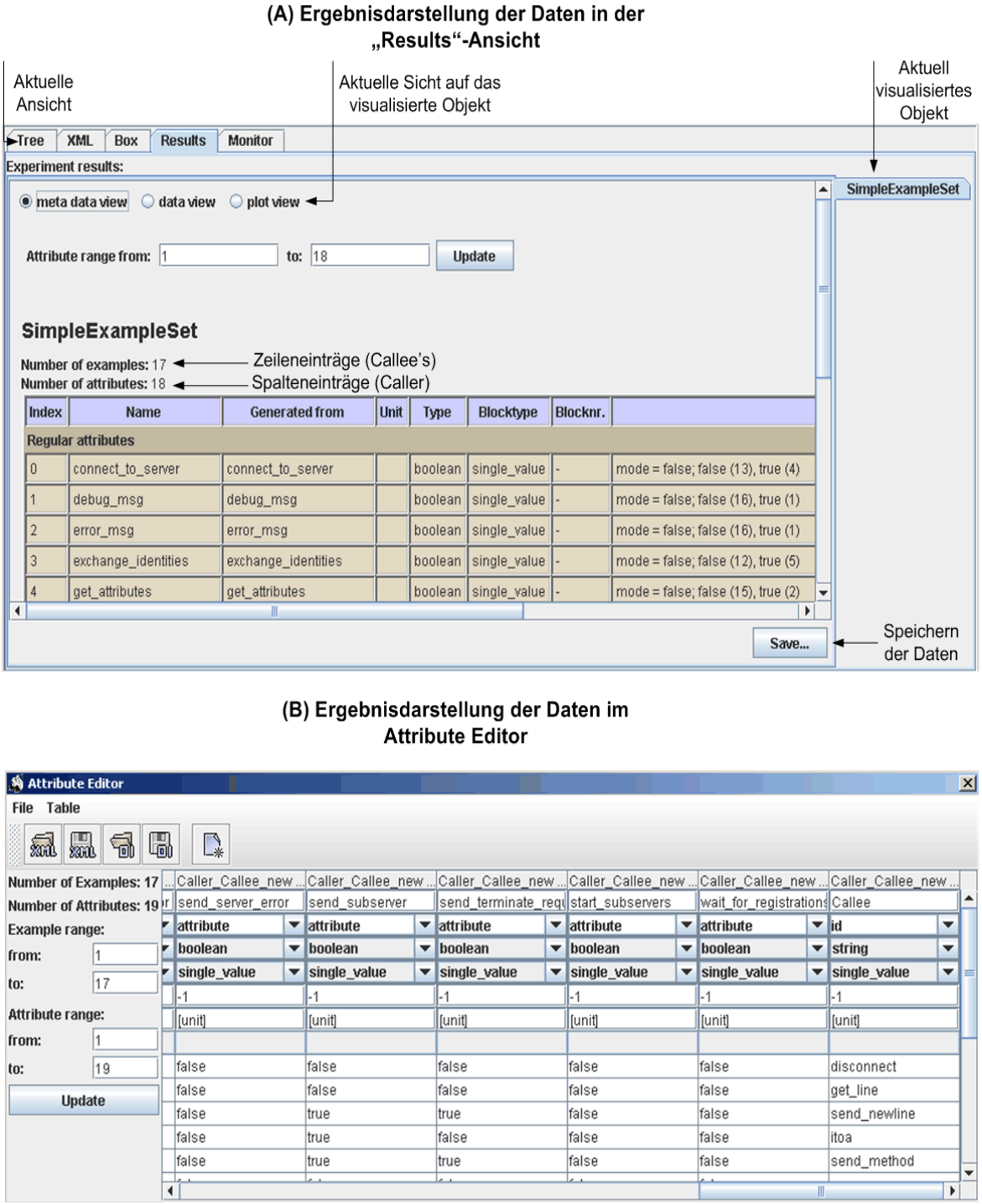


Abbildung 10.3: Ergebnisdarstellung der eingelesenen Daten in der „Results“-Ansicht (A) und im Attribute Editor (B)

10.3.4 Konfiguration und Durchführung der Clusteranalyse

Nachdem die Daten in die Clusteranalyse-Umgebung eingelesen wurden, sollen nun die in den Abschnitten 10.2.3 bis 10.2.5 definierten Konfigurationen in der Clusteranalyse-Umgebung umgesetzt und die Clusteranalyse durchgeführt werden. Hierfür könnte das im vorangegangenen Abschnitt 10.3.3 erstellte Experiment um die notwendigen Operatoren erweitert werden. Es soll jedoch, aus den im Folgenden aufgeführten Gründen, ein neues Experiment erstellt werden. Abbildung 10.4 (A) zeigt den aus folgenden Ausführungen resultierenden Aufbau des Experimentes. Bei der Aneinanderreihung von Operatoren zu einem Experiment muss stets darauf geachtet werden, dass alle Operatoren ihre erwarteten Inputobjekte erhalten. Die Input- und Outputobjekte werden in der `Yale`-Ansicht eines Experimentes nicht visualisiert. Abbildung 10.4 (B) zeigt die Operatorkette des Experimentes aus Abbildung 10.4 (A) mit den von den einzelnen Operatoren erwarteten Inputobjekten und erstellten Outputobjekten.

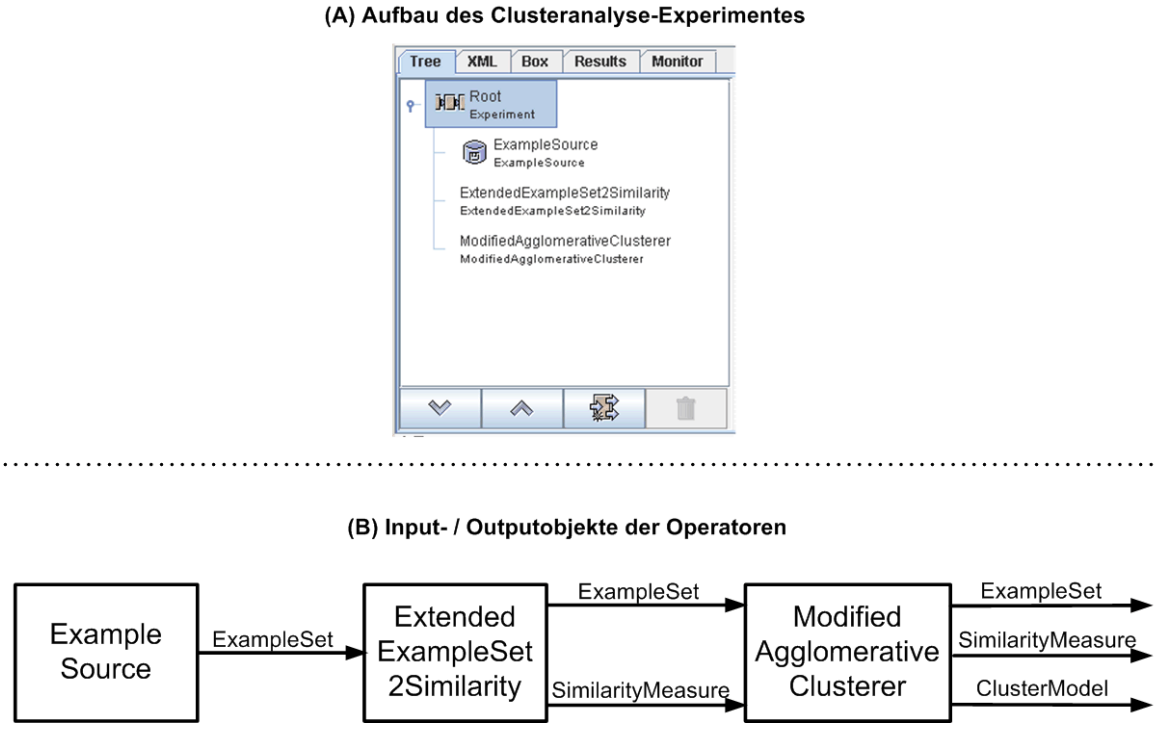


Abbildung 10.4: Aufbau des Clusteranalyse-Experimentes (A) und Input- / Outputobjekte der Operatoren (B)

Der Operator *ExampleSource*

Im vorangegangen Abschnitt wurden die eingelesenen Daten in der Datei „Callee_Caller.xml“ gespeichert. Um die Berechnungszeit einzusparen, die der Operator *Gre-*

qlExampleSource beim Auswerten der `GReQL2`-Anfrage benötigt, werden die Daten direkt aus der Datei „Callee_Caller.xml“ geladen. Dies ist mit dem `Yale`-Operator *ExampleSource*, dem diese Datei als Parameter übergeben wird, möglich. Der Operator findet sich in der Operatorgruppe „IO“ und wird dem Experiment als erster Operator hinzufügt. Als Outputobjekt erzeugt der Operator ein *ExampleSet*.

Der Operator *ExtendedExampleSet2Similarity*

Zur Distanzberechnung wurde in Abschnitt 10.2.3 der *Jaccard Koeffizient* ausgewählt. Da eine einheitliche Behandlung aller Variablen stattfinden soll, eignet sich der Operator *ExtendedExampleSet2Similarity* (siehe Abschnitt 9.1.4) am Besten. Dieser Operator befindet sich in der Operatorgruppe „Clustering.Similarity“ und wird dem Experiment als zweiter Operator hinzugefügt. Der Operator erwartet ein *ExampleSet* als Inputobjekt und erstellt ein *SimilarityMeasure* als Outputobjekt. Da der Operator das *ExampleSet* konsumiert, muss über den boolschen Parameter „keep_example_set“ explizit angegeben werden, dass dieses Objekt ebenfalls als Output weitergegeben werden soll. Abbildung 10.5 zeigt die Parameter-Konfiguration des Operators.

Key	Value
keep_example_set	☑
measure	clustering.similarity.attributebased.JaccardDistance

Abbildung 10.5: Parameter-Konfiguration des Operators *ExtendedExampleSet2Similarity*

Der Operator *ModifiedAgglomerativeClusterer*

Als Clusterverfahren wurde in Abschnitt 10.2.4 das agglomerative hierarchische Verfahren und als Visualisierungsmethode in Abschnitt 10.2.5 das Dendrogramm bestimmt. Diese Visualisierungsmethode wird nur von dem Operator *ModifiedAgglomerativeClusterer* (siehe Abschnitt 9.3) unterstützt. Der Operator befindet sich in der Operatorgruppe „Clustering.Clusterers“ und wird dem Experiment als dritter Operator hinzugefügt. Über den Parameter „linkage criterion“ wird dem Operator mitgeteilt, dass das Complete-Linkage-Kriterium verwendet werden soll. Als Inputobjekte erwartet der Operator ein *ExampleSet* und ein *SimilarityMeasure*, die er von dem vorangegangenen Operator zur Verfügung gestellt bekommt. Als Outputobjekt erstellt der Operator ein *HierarchicalClusterModel*. Sollen auch die Inputobjekte als Output weitergegeben werden, muss dies explizit über die Parameter „keep_example_set“ und „keep_similarity_measure“ angegeben werden. Abbildung 10.6 zeigt die Parameter-Konfiguration des Operators.

Key	Value
keep_example_set	☑
keep_similarity_measure	☑
k	1
linkage criterion	**clustering.similarity.CompleteLinkage**

Abbildung 10.6: Parameter-Konfiguration des Operators *ModofiedAgglomerativeClusterer*

Starten des Experimentes

Das Experiment wird über den Menüpunkt „Run experiment“ gestartet und die Gruppierung der Funktionen berechnet. Die Ansicht wechselt automatisch in den „Results“-Bildschirm.

10.3.5 Auswerten der Ergebnisse

Im „Results“-Bildschirm werden alle Objekte visualisiert, die sich nach Ablauf des Experimentes im *IOContainer* befinden. Im Beispiel sind dies das *ExampleSet* (Datentabelle), das *SimilarityMeasure* (Distanzmatrix) und das *HierarchicalClusterModel* (Clusterbaum). Von besonderem Interesse für die Auswertung der Ergebnisse ist die Visualisierung des Clusterbaumes, die im Folgenden untersucht wird.

Wie in Abschnitt 9.3 bereits beschrieben, existieren als mögliche Ansichten für die Visualisierung eines *HierarchicalClusterModel* die „folder view“, die „graph view“ und die „dendrogram view“. Im angesprochenen Abschnitt wurde erläutert und begründet, dass die „folder view“ und die „graph view“ zur Auswertung der Ergebnisse von hierarchischen Clusterverfahren eher ungeeignet sind. Es wird sich also auf die Auswertung der Ergebnisse anhand der „dendrogram view“ konzentriert. Abbildung 10.7 zeigt das resultierende Dendrogramm. Die Abbildung wurde mit der „Save as image“-Funktion der „dendrogram view“ aus der Clusteranalyse-Umgebung exportiert. Vorher wurde das Dendrogramm aus Übersichtlichkeitsgründen herausgezoomt, weshalb sich die Softwareelemente in y-Richtung sehr nah aneinander befinden.

Zunächst ist festzustellen, dass eine relative flache Hierarchie entsteht. Die Vereinigung der letzten sechs Cluster (*id 28 - id 33*) wurde bei gleicher Distanz vollzogen, wobei es sich um den Distanzwert 1.0 handelt. Dieser Distanzwert kann in der GUI durch Anklicken eines Clusterknotens ermittelt werden. Der Distanzwert 1.0 ist bei Verwendung des *Jaccard Koeffizienten* der höchstmögliche Wert und signalisiert, dass zwischen den zwei betrachteten Objekten bzw. Softwareelementen keinerlei positive Übereinstimmung existiert. Dies lässt schlussfolgern, dass sehr viele Funktionen existieren, die keinen gemeinsamen Aufrufer haben.

Trotz der flachen Hierarchie lassen sich einige strukturelle Informationen aus dem Dendrogramm gewinnen. Die oberen fünf Funktionen des Dendrogramms, die sich zum Cluster *id 27* bei einer Distanz von 0.8 vereinen, weisen untereinander eine hohe Ähnlichkeit auf und

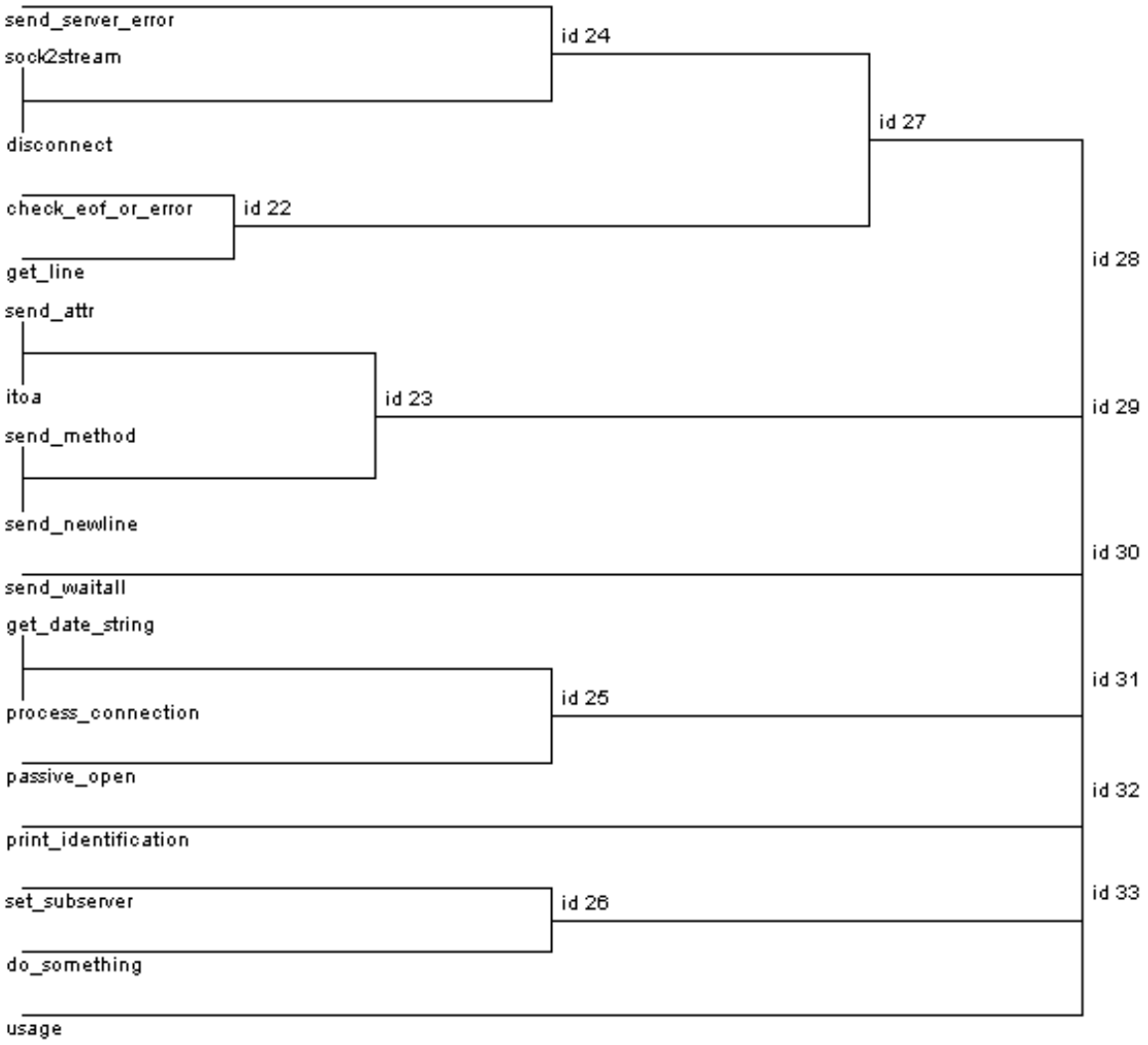


Abbildung 10.7: Ergebnis der Clusteranalyse als Dendrogramm

sind Kandidaten, um in einem gemeinsamen Modul zusammengefasst zu werden. Dies trifft ebenfalls auf die vier folgenden Funktionen, die sich bei einer Distanz von 0.2 zu Cluster *id 23* vereinen, zu. Auch die drei Funktionen die sich zu Cluster *id 25* und die zwei Funktionen, die sich zu Cluster *id 26* vereinen, sind Kandidaten für die jeweilige Platzierung in einem eigenen Modul. Die Funktionen „send_waitall“, „print_identification“ und „usage“ weisen jeweils eine große Distanz zu allen anderen Funktionen auf.

Nachdem diese Erkenntnisse aus dem Dendrogramm gezogen wurden, wurde über eine weitere `GReQL2`-Anfrage ermittelt, welche Funktion zu welchem Modul gehört. Als Ziel dieser Clusteranalyse wurde in Abschnitt 10.2.1 ein Vergleich der durch die Clusteranalyse gewonnenen Strukturierung der Software `Cosmos-Client` mit der Modularisierung der Software durch dessen Autor festgelegt. Das Resultat ist sehr viel versprechend. Abbildung

10.8 zeigt nochmals das aus der Clusteranalyse resultierende Dendrogramm, wobei jede Funktion dem Modul zugewiesen wurde, in welchem sie sich im Quellcode befindet. Die Ergebnisse der Clusteranalyse stimmen weitestgehend mit der Einteilung der Funktionen in Module durch den Autor der Software überein.

Lediglich die Funktion „send_waitall“, die von der Clusteranalyse als isolierte Funktion berechnet wurde, weicht von dem sonst stimmigen Ergebnis ab. Die Funktionen der Cluster *id 25* und *id 26*, die alle zum Modul „server.c“ gehören, weisen keine geringere Distanz zueinander auf als zu Funktionen anderer Module. Gleiches gilt für die Funktionen der Cluster *id 23* und *id 27*, die alle zum Modul „comm.c“ gehören. Soll eine Restrukturierung der Software vorgenommen werden, bieten letztgenannte Funktionen einen Ansatzpunkt.

10.4 Anmerkung

Die in diesem Kapitel vorgestellte Clusteranalyse nutzt nur einen geringen Teil der Funktionalität der Clusteranalyse-Umgebung. Dem Leser wurde jedoch das grundlegende Vorgehen einer Clusteranalyse und die Bedienung der Clusteranalyse-Umgebung näher gebracht.

Aufbauend auf den in Abschnitt 10.3.5 gewonnenen Erkenntnissen, könnten weiterführende Clusteranalysen durchgeführt werden. Durch schrittweise Modifikation einiger oder mehrerer Verfahren bzw. Konfigurationen, kann in einem iterativen Prozess eine Optimierung der Clusterergebnisse und ein besseres Verstehen der zu untersuchenden Software erreicht werden. Zwei mögliche Ansatzpunkte für das vorliegende Beispiel sind:

- Verwenden von intervall-skalierten Features anstelle von binären Features, die statt einer Beziehung „wird aufgerufen von“ die Häufigkeit angeben, in der ein Callee von einem Caller aufgerufen wird.
- Verwenden einer freien Funktion zur Berechnung der Distanz. Dabei können die Caller, die eine Utility-Funktion darstellen (bspw. Fehlerbehandlung, Logging-Funktion, etc.), mit einer geringeren Gewichtung versehen werden.

Anhang B zeigt eine Übersicht der im Rahmen dieser Arbeit entwickelten Operatoren. Diese Übersicht beschreibt die wichtigsten Merkmale der Operatoren, wie Input- und Outputobjekte, Parameter und eine kurze Beschreibung der Funktionalität. Durch das Studieren dieses Kapitels und des Manuals [Wur06] erhält der Leser einen Eindruck über die Konfigurationsmöglichkeiten der Clusteranalyse-Umgebung.

Das Erreichen viel versprechender Ergebnisse mit Clusteranalysen verlangt, gerade aufgrund der vielfältigen Konfigurationsmöglichkeiten einer Clusteranalyse, ein gewisses Maß an Erfahrung des Benutzers mit dieser Disziplin. Diese Erfahrung gilt es sich durch vielfältiges Experimentieren anzueignen.

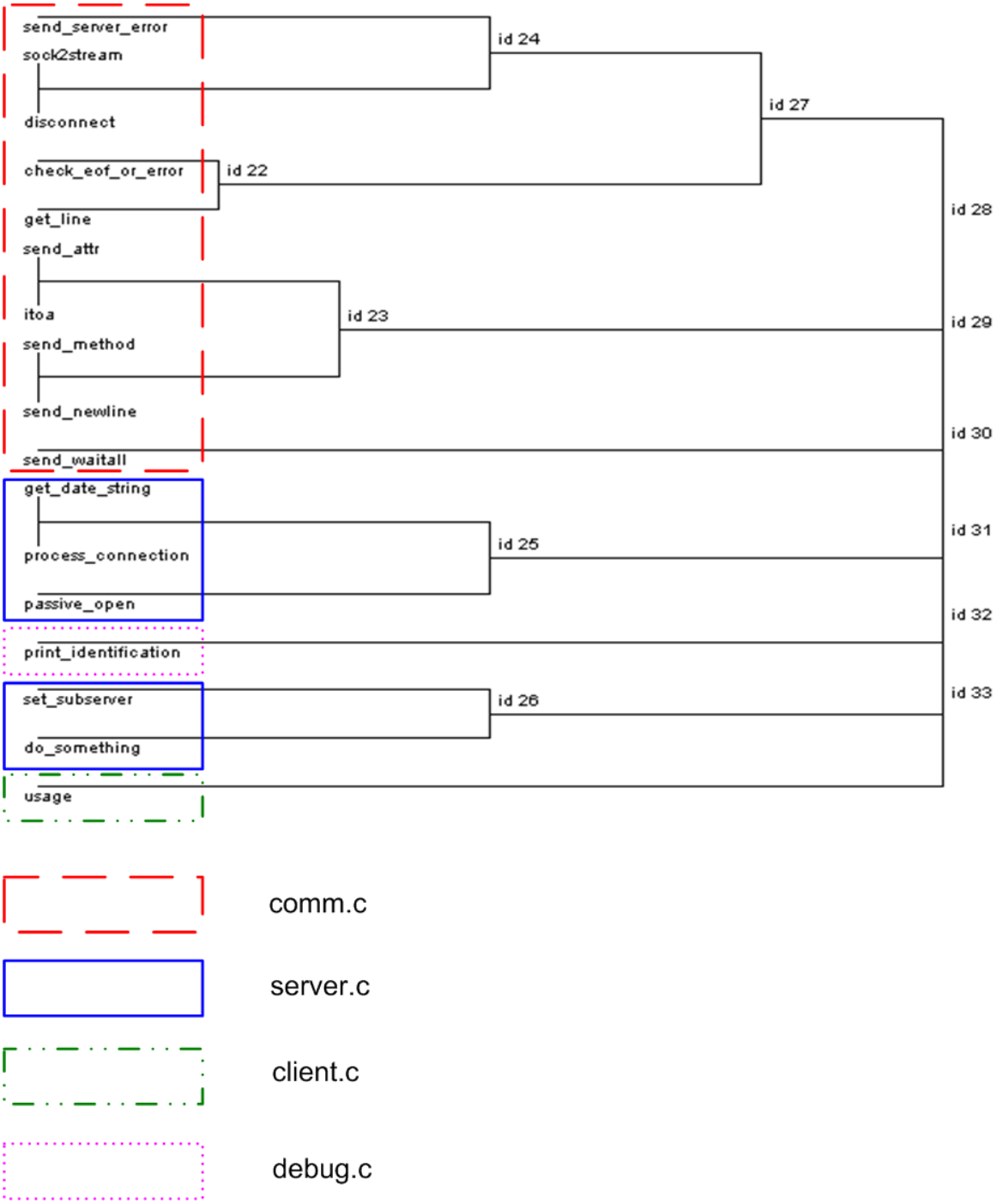


Abbildung 10.8: Zuweisung der Funktionen im Dendrogramm zu den Modulen im Quellcode

11 Fazit

In diesem Kapitel wird die vorliegende Arbeit unter kritischen Gesichtspunkten bewertet und eine Zusammenfassung der Tätigkeiten im Rahmen dieser Diplomarbeit abgegeben.

Abschließend wird in einem Ausblick darüber diskutiert, welche Erweiterungen der Clusteranalyse-Umgebung noch vollzogen werden könnten und wie die Umgebung im praktischen Umfeld eingesetzt werden kann.

11.1 Bewertung und Zusammenfassung

In diesem Abschnitt wird die entwickelte Clusteranalyse-Umgebung bewertet. Die Bewertung der Umgebung findet auf Grundlage der in Anhang A definierten Anforderungsliste statt. Für jeden Abschnitt der Anforderungsliste wird hier, in übereinstimmender Reihenfolge mit den Anforderungen aus Anhang A, geprüft, ob die jeweiligen Anforderungen erfüllt wurden.

Zusätzlich kann dieser Abschnitt als eine Zusammenfassung der Tätigkeiten und Resultate dieser Diplomarbeit angesehen werden. Die Bewertung der Arbeit anhand der definierten Anforderung beschreibt hier zusätzlich, wie diese Anforderungen im Zeitverlauf erfüllt wurden.

11.1.1 Architektur

Die Clusteranalyse-Umgebung setzt sich aus den Softwarekomponenten

- `GUPRO` (siehe Kapitel 6),
- `Yale` (siehe Kapitel 5) und
- `JEP` (siehe Abschnitt 9.1.3)

zusammen, wobei `GUPRO` und `Yale` die wesentlichen Bestandteile der Umgebung ausmachen. Die Software `Yale` wurde in einem strukturierten Software-Evaluations-Prozess (siehe Kapitel 4) ausgewählt.

Die Kopplung der Komponenten `GUPRO` und `Yale` wurde durch den Operator *GreqlExampleSource* (siehe Abschnitt 8.1) realisiert. Es handelt sich dabei um eine lose Kopplung, sodass beide Softwarekomponenten unabhängig voneinander verwendet werden können und

neue Releases der Komponenten in die Clusteranalyse-Umgebung integriert werden können (vorausgesetzt die Schnittstellen der Komponenten wurden nicht verändert). Auch die Anbindung der Software `JEP` an `Yale` wurde über eine einheitliche und kompakte Schnittstelle durch eine lose Kopplung realisiert (siehe Abschnitt 9.1.3).

Es wurde bei keiner der verwendeten Softwarekomponenten eine direkte Modifikation des Quellcodes vorgenommen und alle Erweiterungen in Plugins (siehe Kapitel 8 und 9) zusammengefasst. Der Datenaustausch zwischen `GUPRO` und `Yale` wurde für beide Richtungen umgesetzt (siehe Kapitel 8). Die Software `Yale` wurde als einheitliche Benutzerschnittstelle ausgewählt und notwendige Erweiterungen vorgenommen (siehe Abschnitt 7.2).

Somit wurden alle unter Abschnitt A.1 definierten Anforderungen erfüllt.

11.1.2 Reverse Engineering Tool

In Kapitel 6 wurde die Software `GUPRO` eingeführt und bereits gezeigt, dass `GUPRO` alle unter Abschnitt A.2 definierten Anforderungen erfüllt.

11.1.3 Clusteranalyse-Funktionalität

Bereits im Auswahl-Prozess einer Software zur Clusteranalyse in Kapitel 4 wurde darauf geachtet, dass die ausgewählte Software möglichst viele Anforderungen aus Abschnitt A.3 erfüllt.

Eine entscheidende Anforderung an die Clusteranalyse-Umgebung war, dass diese sich vor allem als **experimentelle** Plattform für Softwareclustering eignet, was eine breite Palette an Konfigurationsmöglichkeiten voraussetzt. Zusätzlich zu den in `Yale` vorhandenen Möglichkeiten wurden deshalb im Plugin „Softwareclustering“ (siehe Kapitel 9) weitere Features implementiert.

Dazu gehört die - bisher noch in keiner dem Autor bekannten Clusteranalyse-Software vorhandenen - freie Eingabe einer Funktion zur Distanz- bzw. Ähnlichkeitsberechnung (Abschnitt 9.1), das Hinzufügen des Average-Linkage-Kriteriums (Abschnitt 9.2) und die Realisierung des Dendrogramms als Visualisierungsmethode für hierarchische Clusterverfahren (Abschnitt 9.3). Die Fülle von Clusterverfahren, die in `Yale` bereits zur Verfügung stehen, wurden auf Korrektheit überprüft und falls notwendig korrigiert (siehe Abschnitt 9.4).

Sicherlich kann die Clusteranalyse-Umgebung bzgl. der Clusteranalyse-Funktionalität noch in vielerlei Hinsicht erweitert werden (z.B. um weitere Linkage-Kriterien, Visualisierungsmethoden, etc.). Es wurden jedoch alle unter Abschnitt A.3 definierten Anforderungen erfüllt und dem Benutzer damit zahlreiche Konfigurationsmöglichkeiten einer Clusteranalyse offen gehalten.

11.1.4 Benutzerschnittstelle

Die Software `Yale`, die als einheitliche Benutzerschnittstelle der Clusteranalyse-Umgebung fungiert, verfügt über eine graphische Benutzeroberfläche (siehe Abschnitt 5.3).

Die Bedienung der Oberfläche verlangt etwas Einarbeitungszeit, stellt sich aber nach ersten Erfahrungen als benutzerfreundlich heraus. `Yale` verfolgt ein modulares Operatorenkonzept (siehe Abschnitt 5.2.1), was eine modulare und damit schrittweise Konfiguration eines Experimentes erlaubt. Ebenfalls können einzelne Operatoren eines Experimentes ausgetauscht werden, sodass dessen Einfluss auf das Ergebnis einer Clusteranalyse gut nachvollzogen werden kann.

`Yale` verfügt über zahlreiche Visualisierungsmethoden, z.B. für Datentabellen, Distanz- bzw. Ähnlichkeitswerte und Ergebnisse von Clusterverfahren. Während die Visualisierung von Datentabellen z.B. in Form von einigen Plots und tabellarischen Statistiken sehr gut gelöst ist, könnte die Visualisierung von Clusteranalyseergebnissen noch um einige Features erweitert werden, die dem Benutzer ein Verstehen der Ergebnisse erleichtern und den Ablauf des verwendeten Clusterverfahrens transparent machen.

11.1.5 Programmierschnittstelle

Da Yale die zentrale Komponente der Clusteranalyse-Umgebung darstellt, ist die Programmierschnittstelle der Clusteranalyse-Umgebung mit der API von `Yale` gleichzusetzen. Eine Dokumentation der `Yale`-API findet sich in [MFK06].

Sowohl der Quellcode der Software `Yale`, als auch der Quellcode der im Rahmen dieser Arbeit entwickelten Plugins, sind auf einer dieser Diplomarbeit beigelegten CD zu finden. Eine Dokumentation der im Plugin befindlichen Implementationen findet sich in dieser Arbeit (siehe Kapitel 8 und 9).

Alle in Abschnitt A.5 geforderten Erweiterungsmöglichkeiten können mithilfe der angegebenen API realisiert werden. Zur Anbindung eines weiteren Tools zur Extraktion von Informationen aus Softwareartefakten, kann die Umsetzung des Operators *GreqlExampleSource* (siehe Abschnitt 8.1) als Vorbild genommen werden.

11.1.6 Allgemeines

Fehlerbehandlung

Bei der Realisierung der neuen Funktionalität wurde jederzeit auf eine geeignete Fehlerbehandlung geachtet. Nach dem softwaretechnischen Prinzip der Vollständigkeit wurden stets alle möglichen Fälle von Benutzereingaben erfasst und für fehlerhafte und unangemessene Konfigurationen möglichst aussagekräftige Fehlermeldungen erzeugt. Die erzeugten Fehlermeldungen geben den Operator und den Parameter an, der den Fehler verursacht hat.

Tests

Entwicklungsbegleitend wurden für die Erweiterungen Black-Box-Tests durchgeführt. Als Eingabedaten für die Clusteranalysen wurden Datentabellen mit verschiedenen Variablentypen (symmetrisch und asymmetrisch binär, nominal, intervall-skaliert) erstellt. Die Daten der Tabelle decken einige Normalfälle, Randfälle und Fehlerfälle ab. Die verwendeten Datentabellen können der Klasse *GenerateTestData*, die sich in der Paketstruktur der beiden Plugins „GReQL-Interface“ und „Softwareclustering“ befindet, nachgelesen werden.

Außerdem wurden die Implementation anhand von konkreten TGraphen getestet, die sich auf der beiliegenden CD befinden. Kapitel 10 zeigt die Anwendung der Clusteranalyse-Umgebung auf den konkreten TGraphen „cosmos-client“.

Zusätzlich wurden die bereits in `Yale` implementierten Clusterverfahren mit Black-Box-Tests auf Korrektheit überprüft (siehe Abschnitt 9.4).

Dokumentation

Die Dokumentation des gesamten Projektablaufes wurde in Form dieser Arbeit erstellt. Die Dokumentation der Architektur der Clusteranalyse-Umgebung findet sich ebenfalls in dieser Arbeit (siehe Kapitel 7).

Eine Dokumentation von `GUPRO` findet sich auf der `GUPRO`-Homepage[1], eine Dokumentation von `GReQL2` in den Diplomarbeiten [Mar06] und [Bil06] und eine Dokumentation von Yale in [MFK06].

Eine Dokumentation der Erweiterungen der Software `Yale` von Seiten des Autors findet sich in dieser Diplomarbeit in den Kapiteln 8 und 9. Insbesondere zeigt Anhang B eine Übersicht über die entwickelten Plugins und Operatoren. Die Dokumentation des Quellcodes und eine Spezifikation der Methoden wurde mit dem Tool `JavaDoc` erstellt und findet sich auf der beiliegenden CD.

11.2 Ausblick

Ziel der Arbeit war die Entwicklung einer Clusteranalyse-Umgebung durch Kopplung und Erweiterung bestehender Lösungen. Die entstehende Umgebung soll dem Benutzer vielfältige Möglichkeiten zur Konfiguration und zum Experimentieren offen halten.

Da die Möglichkeiten der Konfiguration einer Clusteranalyse - wie bereits mehrmals erwähnt - extrem vielfältig sind, kann natürlich auch die entwickelte Clusteranalyse-Umgebung nicht alle Verfahren der Clusteranalyse abdecken. Hier gilt es im praktischen Einsatz der Umgebung eventuelle Unzulänglichkeiten und fehlende Funktionalität zu ermitteln und bei Bedarf zu integrieren. Da der Quellcode der Umgebung zur Verfügung steht

[1] http://www.gupro.de

und saubere Schnittstellen zur Erweiterung derselben existieren, ist für die Möglichkeit der zukünftigen Erweiterung gesorgt.

Am Institut für Softwaretechnik der Universität Koblenz-Landau soll die Clusteranalyse-Umgebung eingesetzt werden, um das Potential von Clusteranalysen auf Softwareelementen zu erforschen. Wie bereits in Abschnitt 2.4 dargelegt, bietet die Disziplin des Softwareclustering, abgesehen von den bisherigen Erkenntnissen, noch viel Raum für neue Erkenntnisse und Ansätze. Durch Variation in der Konfiguration einer Clusteranalyse, wie z.B. der verwendeten Daten, der Distanz- bzw. Ähnlichkeitsfunktion oder der Clusterverfahren, soll ermittelt werden, wie sich dies auf die erzielten Clusterergebnisse und deren praktische Relevanz auswirkt.

Die entwickelte Clusteranalyse-Umgebung mit ihren zahlreichen Konfigurationsmöglichkeiten stellt für diese Aufgabe eine geeignete Plattform dar.

Obwohl die entwickelte Umgebung in dieser Arbeit als Clusteranalyse-Umgebung bezeichnet wurde, eröffnet sie noch zusätzliche Möglichkeiten: Wie in Kapitel 5 bereits angesprochen, handelt es sich bei `Yale` um eine Data Mining-Software, die über 300 Data Mining-Operatoren zur Verfügung stellt. Durch die Anbindung der Umgebung an `GUPRO` über das Plugin „GReQL-Interface“, bietet sich die Möglichkeit, das Potential der Methoden des Data Mining für das Reverse Engineering zu erforschen. Diese Untersuchungen können mithilfe der Clusterananalyse-Umgebung durchgeführt werden und Ziel einer an diese Arbeit anknüpfenden Forschungsarbeit sein.

A Anforderungsliste

Im Folgenden werden die detaillierten Anforderungen an die Clusteranalyse-Umgebung als Anforderungsliste aufgeführt.

Die Anforderungsliste orientiert sich an den als Fließtext formulierten Anforderungen aus Kapitel 3. Sowohl die Gliederung, als auch die Bedeutung wichtiger Begrifflichkeiten der folgenden Anforderungsliste stimmen mit der Beschreibung in Kapitel 3 überein und können den einleitenden Abschnitten dieses Kapitels entnommen werden.

A.1 Anforderungen an die Architektur

1. Die Clusteranalyse-Umgebung wird nicht von Grund auf neu entwickelt, sondern macht sich bestehende Lösungen zunutze.
2. Die Auswahl der Komponenten der Clusteranalyse-Umgebung ist wichtiger Bestandteil der Arbeit. Die ausgewählten Komponenten sollten bereits einen Großteil der Anforderung aus A.2 und A.3 abdecken, um zusätzlichen Entwicklungsaufwand zu minimieren.
3. Die Kopplung zwischen den einzelnen Komponenten soll möglichst lose sein.
4. Neue Releases der verwendeten Komponenten sollen problemlos in die Clusteranalyse-Umgebung integriert werden können.
5. Ein Import der vom Reverse Engineering Tool erstellten Daten in die Clusteranalyse-Software ist möglich.
6. Ein Rückschreiben der Clusteranalyse-Ergebnisse in das vom Reverse Engineering Tool erstellte Dateiformat ist möglich.
7. Der Benutzer kann die Clusteranalyse-Umgebung über eine einheitliche Benutzerschnittstelle bedienen.
8. Alle Erweiterungen einer verwendeten Software von Seiten des Autors werden als eigenständiges Modul im Sinne einer Softwarekomponente erstellt.

A.2 Anforderungen an das Reverse Engineering Tool

1. Eine Analyse von Programmcode ist möglich.

2. Neben Quellcode ist auch die Analyse von weiteren Softwareartefakten (Datenbankdefinitionen, Präprozessordefinitionen,...) möglich.

3. Softwareartefakte (wie Programmcode) sind Eingabe des Reverse Engineering Tools.

4. Der Benutzer kann festlegen, welche Informationen aus den Softwareartefakten extrahiert werden. Die zu extrahierenden Informationen, konkreter die Objekte der Softwareartefakte und deren Beziehungen, sind dabei frei wählbar.

5. Die Extraktion von Informationen aus Softwareartefakten (wie Programmcode) auf verschiedenen Abstraktionsgraden ist möglich.

6. Die Analyse von Softwareartefakten beliebiger Programmiersprachen ist möglich.

7. Analyseergebnisse von Softwareartefakten können persistent gespeichert werden.

8. Die Analyseergebnisse sind in einem programmiersprachenunabhängigen und offenen Format abgelegt.

9. Es existieren Schnittstellen zum Auslesen der Analyseergebnisse.

10. Aus den Analyseergebnissen können wiederum Informationen beliebiger Abstraktionsgrade und Komplexität extrahiert werden.

A.3 Anforderungen an die Clusteranalyse-Funktionalität

1. Die Clusteranalyse-Umgebung soll dem Benutzer möglichst großen Freiraum zur Konfiguration einer Clusteranalyse bieten und damit als experimentelle Plattform zur Durchführung von Softwareclustering geeignet sein. Die folgenden Anforderungen dieses Abschnittes konkretisieren diese Anforderung.

2. Ein Editor zum Betrachten und Editieren der eingelesenen Daten existiert.

3. Zum Erstellen einer Distanz- bzw. Ähnlichkeitsmatrix stehen die bekanntesten (siehe Kapitel 2.2) Distanz- bzw. Ähnlichkeitsfunktionen zur Verfügung. Außerdem hat der Benutzer die Möglichkeit der freien Eingabe einer Funktion zur Berechnung der Distanzen bzw. Ähnlichkeiten. Dabei kann sich der Benutzer auf einzelne Variablen der Datentabelle beziehen.

4. Es stehen eine Vielzahl von Clusterverfahren zur Auswahl. Darunter befinden sich mindestens folgende Clusterverfahren:

   - K-Means
   - K-Medoids
   - hierarchisch agglomeratives Verfahren

- DBScan

5. Handelt es sich um ein hierarchisches Verfahren, sind mindestens folgende Linkage-Kriterien wählbar:
   - Single-Linkage
   - Complete-Linkage
   - Average-Linkage
6. Die Ergebnisse von Clusterverfahren können in einer dem jeweiligen Verfahren angemessenen Visualisierungsmethode dargestellt werden.
7. Die Ergebnisse von Clusterverfahren können persistent gespeichert und jederzeit wieder in die Clusteranalyse-Umgebung geladen werden.

A.4 Anforderungen an die Benutzerschnittstelle

1. Die Clusteranalyse-Umgebung kann über eine graphische Benutzeroberfläche (GUI) bedient werden.
2. Die Bedienung der Clusteranalyse-Umgebung soll übersichtlich, intuitiv erlernbar und benutzerfreundlich sein.
3. Eine schrittweise Konfiguration einer Clusteranalyse ist möglich.
4. Einzelne Schritte einer Clusteranalyse sind austauschbar, d.h: Wurde eine Clusteranalyse durchgeführt, kann dieselbe Konfiguration - z.B. Datentabelle, Distanz- bzw. Ähnlichkeitsfunktion, Clusterverfahren, Visualisierungsmethode - unter Austausch eines Schrittes durchgeführt werden, ohne eine völlig neue Konfiguration zu erstellen.
5. Die Ergebnisse aller Schritte einer Clusteranalyse - z.B. Datentabelle, Distanz- bzw. Ähnlichkeitsmatrix, Clusterverfahren, Evaluation der Clusterergebnisse - sind einzeln visualisierbar.

A.5 Anforderungen an die Programmierschnittstelle

1. Die Clusteranalyse-Umgebung kann vom Benutzer erweitert werden. Zu den Erweiterungsmöglichkeiten zählen die Erweiterung von:
   - der Anbindung weiterer Tools zur Extraktion von Informationen aus Softwareartefakten
   - Distanz- bzw. Ähnlichkeitsfunktionen
   - Clusterverfahren

- Linkage-Kriterien
- Visualisierungsmethoden

A.6 Allgemeine Anforderungen

A.6.1 Fehlerbehandlung

1. Bei einer falschen Konfiguration der Clusteranalyse-Umgebung durch den Benutzer, wird dieser über die Fehlkonfiguration mit einer geeigneten Fehlermeldung informiert.
2. Die Fehlermeldung soll möglichst die genaue Stelle der Fehlkonfiguration aufdecken.

A.6.2 Tests

1. Die Erweiterungen der verwendeten Softwarekomponenten von Seiten des Autors werden entwicklungsbegleitend getestet.
2. Als primäres Testverfahren werden Black-Box-Tests verwendet.
3. Nach fertig stellen der Clusteranalyse-Umgebung werden konkrete Clusteranalysen auf Softwareelementen durchgeführt.

A.6.3 Dokumentation

1. Eine Dokumentation des gesamten Projektablaufes wird erstellt.
2. Eine Dokumentation der Architektur der Clusteranalyse-Umgebung wird erstellt.
3. Eine Dokumentation des verwendeten Reverse Engineering Tools ist vorhanden.
4. Eine Dokumentation der verwendeten Clusteranalyse-Software ist vorhanden.
5. Eine Dokumentation aller Erweiterungen von Seiten des Autors wird erstellt.

B Plugins und Operatoren

Dieses Kapitel zeigt eine Übersicht über die im Rahmen dieser Arbeit realisierten Plugins und den in diesen Plugins enthaltenen Operatoren. Für jeden realisierten Operator werden folgende Angaben gemacht:

- Name
- Operatorgruppe
- Klassenpfad im Quellcode
- Benötigte Inputobjekte
- Erstellte Outputobjekte
- Parameter
- Kurzbeschreibung der Funktionalität

Tabelle B.1 zeigt die erstellten Plugins „GReQL-Interface“ und „Softwareclustering“ und die Operatoren, die in den Plugins enthalten sind. Für jeden Operator ist der Abschnitt innerhalb dieser Arbeit angegeben, in dem der Operator näher beschrieben wird.

Plugin	**Operator**	**Abschnitt**
GReQL-Interface	Cluster2JValue	8.2
	GreqlExampleSource	8.1
Softwareclustering	DistanceOrSimilarityCalculator	9.1.2
	ExtendedExampleSet2Similarity	9.1.4
	Hierachical2FlatClusterModel	9.5.1
	ModifiedAgglomerativeClusterer	9.2 und 9.3
	ModifiedKMedoids	9.4

Tabelle B.1: Plugins und ihre Operatoren

B.1 GReQL-Interface

In diesem Abschnitt werden die Operatoren des Plugin „GReQL-Interface“ in alphabetischer Reihenfolge aufgelistet.

B.1.1 Cluster2JValue

Name: Cluster2JValue **Operatorgruppe:** Clustering.IO

Klassenpfad: clustering.operator.io.greql.Cluster2JValue

Benötigte Inputobjekte:

- *ExampleSet*
- *ClusterModel*

Erstellte Outputobjekte:

- *ExampleSet*

Parameter:

- Result_File (String): Die Datei, in die das JValue-Objekt geschrieben wird.
- ClusterToExampleSet (Boolean): Gibt an, ob das *ExampleSet* mit einem zusätzlichen Attribut versehen werden soll, das das dem jeweiligen Objekt zugeteilte Cluster spezifiziert (true).

Kurzbeschreibung:

Ordnet die Ergebnisse einer Clusteranalyse den Objekten der Datentabelle zu und speichert die Datentabelle als Objekt vom Typ *JValue* in einer XML-Datei.

B.1.2 GreqlExampleSource

Name: GreqlExampleSource **Operatorgruppe:** IO.Examples

Klassenpfad: clustering.operator.io.greql.GreqlExampleSource

Benötigte Inputobjekte:

- *keine*

Erstellte Outputobjekte:

- *ExampleSet*

Parameter:

- Graph_File (File): Die Datei, die den zu ladenden TGraphen enthält.
- Query (String): Eine `GReQL`-Anfrage als String. Dieser Parameter hat Priorität gegenüber dem Parameter „Query_File“.
- Query_File (File): Die Datei, die eine `GReQL`-Anfrage enthält.
- ID_attribut (String): Das Attribut, das als *id*-Attribut deklariert werden soll.

Kurzbeschreibung:

Liest Daten aus einem TGraphen durch Absetzen einer `GReQL`-Anfrage ein und konvertiert die Daten in ein *ExampleSet*. Vorsicht: Es sind ausschließlich Tabellen ohne verschachtelte Strukturen erlaubt.

B.2 Softwareclustering

In diesem Abschnitt werden die Operatoren des Plugin „Softwareclustering“ in alphabetischer Reihenfolge aufgelistet.

B.2.1 DistanceOrSimilarityCalculator

Name: DistanceOrSimilarityCalculator **Operatorgruppe:** Clustering.Similarity

Klassenpfad: clustering.similarity.attributebased.DistanceOrSimilarityCalculator

Benötigte Inputobjekte:

- *ExampleSet*

Erstellte Outputobjekte:

- *SimilarityMeasure*

Parameter:

- Function (String): Die Funktion zur Berechnung einer Distanz- bzw. Ähnlichkeitsmatrix als String. Dieser Parameter hat Priorität gegenüber dem Parameter „Function_File“.
- Function_File (File): Die Datei, die die Funktion zur Berechnung einer Distanz- bzw. Ähnlichkeitsmatrix enthält.
- Distance (Boolean): Gibt an, ob es sich bei der spezifizierten Funktion um eine Distanz- (true) oder Ähnlichkeitsfunktion (false) handelt.
- Normalize_Numerical_Attributes (Boolean): Gibt an, ob intervall-skalierte Werte des *ExampleSet* normalisiert werden sollen (true) oder nicht (false).

Kurzbeschreibung:

Berechnet eine Distanz- bzw. Ähnlichkeitsmatrix aus einem *ExampleSet*. Der Benutzer kann dabei eine freie Funktion zur Berechnung der Matrix angeben.

B.2.2 ExtendedExampleSet2Similarity

Name: ExtendedExampleSet2Similarity **Operatorgruppe:** Clustering.Similarity

Klassenpfad: clustering.similarity.attributebased.ExtendedExampleSet2Similarity

Benötigte Inputobjekte:

- *ExampleSet*

Erstellte Outputobjekte:

- *SimilarityMeasure*

Parameter:

- keep_example_set (Boolean): Gibt an, ob das *ExampleSet* als Output weitergegeben werden soll (true) oder nicht (false).
- measure (Category): Die Funktion zur Berechnung der Distanz- bzw. Ähnlichkeitsmatrix, die aus einer vorgegebenen Menge von Funktionen ausgewählt werden kann. Zur Auswahl stehen:
  - *EuclidianDistance*
  - *ManhattanDistance*
  - *CosineSimilarity*
  - *JaccardDistance*
  - *SimpleMatchingDistance*

Kurzbeschreibung:

Berechnet eine Distanz- bzw. Ähnlichkeitsmatrix aus einem *ExampleSet*. Der Benutzer kann dabei auf eine Menge vorgegebener Funktionen zurückgreifen.

B.2.3 Hierachical2FlatClusterModel

Name: Hierachical2FlatClusterModel **Operatorgruppe:** Clustering.Misc

Klassenpfad: clustering.operator.misc.Hierachical2FlatClusterModel

Benötigte Inputobjekte:

- *HierarchicalClusterModel*

Erstellte Outputobjekte:

- *FlatClusterModel*

Parameter:

- K (Integer): Gibt die Anzahl von Clustern an, die das *FlatClusterModel* enthalten soll.

Kurzbeschreibung:

Erstellt aus einem hierarchischen Clusterbaum eine flache Clusterhierarchie mit *K* Clustern. Dabei enthalten die *K* Cluster der flachen Hierarchie alle Objekte, die sich im Subbaum des jeweiligen Clusters im hierarchischen Clusterbaum befinden.

B.2.4 ModifiedAgglomerativeClusterer

Name: ModifiedAgglomerativeClusterer **Operatorgruppe:** Clustering.Clusterers

Klassenpfad: clustering.operator.clusterer.ModifiedAgglomerativeClusterer

Benötigte Inputobjekte:

- *ExampleSet*
- *SimilarityMeasure*

Erstellte Outputobjekte:

- *HierarchicalClusterModel*

Parameter:

- keep_example_set (Boolean): Gibt an, ob das *ExampleSet* als Output weitergegeben werden soll (true) oder nicht (false).
- keep_similarity_measure (Boolean): Gibt an, ob das *SimilarityMeasure* als Output weitergegeben werden soll (true) oder nicht (false).
- linkage criterion (Category): Das Linkage-Kriterium, das zur Berechnung der Gruppierung verwendet werden soll. Es stehen folgende Linkage-Kriterien zur Verfügung:
  - *SingleLinkage*
  - *CompleteLinkage*
  - *AverageLinkage*

Kurzbeschreibung:

Realisiert das hierarchisch agglomerative Clusterverfahren. Ermöglicht als Visualisierungsmethode das Dendrogramm.

B.2.5 ModifiedKMedoids

Name: ModifiedKMedoids **Operatorgruppe:** Clustering.Clusterers

Klassenpfad: clustering.operator.clusterer.ModifiedKMedoids

Benötigte Inputobjekte:

- *ExampleSet*
- *SimilarityMeasure*

Erstellte Outputobjekte:

- *FlatClusterModel*

Parameter:

- keep_example_set (Boolean): Gibt an, ob das *ExampleSet* als Output weitergegeben werden soll (true) oder nicht (false).
- keep_similarity_measure (Boolean): Gibt an, ob das *SimilarityMeasure* als Output weitergegeben werden soll (true) oder nicht (false).
- k (Integer): Die Anzahl der Cluster, die das *FlatClusterModel* enthalten soll.

Kurzbeschreibung:

Realisiert das K-Medoids-Clusterverfahren. Der Operator ist ein Bugfix des `Yale`-Operators *KMedoids*.

Literaturverzeichnis

[ABKS99] ANKERST, M. ; BREUNING, M. ; KRIEGEL, H.-P. ; SANDER, Joerg: OPTICS: Ordering Points To Identify the Clustering Structure. In: *CM SIGMOD International Conference on Management of Data* (1999), S. 49–60

[AL99] ANQUETIL, N. ; LETHBRIDGE, C.: Experiments with clustering as a software remodularization method. In: *6th Working Conference on Reverse Engineering* IEEE, 1999

[Bac96] BACHER, Johann: *Clusteranalyse: anwendungsorientierte Einführung*. München : Oldenbourg, 1996

[BE81] BELADY, L. ; EVANGELISTI, C.: System partitioning and its measures. In: *Journal of Systems and Software* (1981), S. 23–29

[Bil06] BILDHAUER, D.: *Ein Interpreter für GReQL 2 - Entwurf und prototypische Implementation*, Universität Koblenz-Landau, Institut für Softwaretechnik, Diplomarbeit, 2006

[BMR+96] BUSCHMANN, F. ; MEUNIER, R. ; ROHNERT, H. ; SOMMERLAD, P. ; STAL, M.: *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley, 1996

[Cac73] CACOULLOS, C.: *Discriminant Analysis and Application*. New York : John Wiley and sons, 1973

[CH01] COUNCILL, T. ; HEINEMANN, G.: *Component-Based Software Engineering*. Addison-Wesley, 2001

[CLR00] CORMEN, T. ; LEISERSON, C. ; RIVEST, R.: *Introduction to Algorithms*. McGraw-Hill, 2000

[DEF+98] DAHM, P. ; EBERT, J. ; FRANZKE, A. ; KAMP, M. ; WINTER, A.: TGraphen und EER-Schemata — Formale Grundlagen. In: *GUPRO — Generische Umgebung zum Programmverstehen*. 1998, S. 51–66

[EKSX96] ESTER, Martin ; KRIEGEL, Hans-Peter ; SANDER, Jörg ; XU, Xiaowei: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: *Second International Conference on Knowledge Discovery and Data Mining*, 1996. – http://ifsc.ualr.edu/xwxu/publications/kdd-96.pdf, S. 226–231

[EKW98] EBERT, J. ; KAMP, M. ; WINTER, A.: GUPRO: A Generic System to Support Multi-Language Understanding of Heterogeneous Software. In: *GUPRO — Generische Umgebung zum Programmverstehen*. 1998, S. 11–30

[ESPD98] EISEN, P. ; SPELLMANN, P. ; P., Brown ; D., Botstein: Cluster Analysis and Display of Genome-Wide Expression Patterns. In: *National Academy of Sciences*, 1998

[Fra97] FRANZKE, A.: Gral: A reference manual / Universität Koblenz-Landau, Fachbereich Informatik. 1997. – Fachbericht Informatik

[Gam97] GAMMA, Erich: *Design Patterns*. Addison-Wesley Professional, 1997

[Gor99] GORDON, A.: *Classification*. 2. London : Chapman and Hall, 1999

[HB85] HUTCHENS, D. ; BASILI, V.: System structure analysis: Clustering with data bindings. In: *IEEE Transactions on Software Engineering* (1985), S. 749–757

[HK01] HAN, Jiawei ; KAMBER, Micheline: *Data Mining, Concepts and Techniques*. Morgan Kaufmann, 2001

[ISO01a] ISO. *International Standard ISO/IEC 14598-1: Information Technology - Software Product Evaluation*. 07 2001

[ISO01b] ISO. *International Standard ISO/IEC 9126-1, Software engineering - Product quality - Part1: Quality model*. 06 2001

[Kah06] KAHLE, S.: *Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Universität Koblenz-Landau, Institut für Softwaretechnik, Diplomarbeit, 2006

[KK01] KAMP, M. ; KULLBACH, B.: GReQL - Eine Anfragesprache für das GUPRO-Repository - Sprachbeschreibung (Version 1.3) / Universität Koblenz-Landau, Institut für Informatik. 2001. – Fachbericht Informatik

[Kos00] KOSCHKE, R.: *Atomic Architectural Component Recovery for Program Understanding and Evolution*, University of Stuttgart, Diss., 2000

[KR90] KAUFMAN, L. ; ROUSSEEUW, P.: *Finding Groups in data: An Introduction to Cluster Analysis*. New York : John Wiley and sons, 1990

[Lun98] LUNG, C.: Software Architecture Recovering and Restructering through Clustering Techniques. In: *3rd International Softare Architecture Workshop* ISAW, 1998, S. 101–104

[Mar06] MARCHEWKA, K.: *Entwurf und Definition der Graphanfragesprache GReQL 2*, Universität Koblenz-Landau, Institut für Softwaretechnik, Diplomarbeit, 2006

[MF06] MIERSWA, Ingo ; FISCHER, Simon: *The Yale GUI Manual*. http://yale.sf.net/: University of Dortmund, 2006

[MFK06] MIERSWA, Ingo ; FISCHER, Simon ; KLINKENBERG, Ralf: *The Yale 3.2 Tutorial: User Guide, Operator Reference, Developer Guide*. http://yale.sf.net/: University of Dortmund, 2006

[Mit97] MITCHEL, Tom: *Machine Learning*. McGraw Hill, 1997

[MRW77] MCCALL, J. ; RICHARDS, P. ; WALTERS, F.: Factors in Software Quality / US Rome Air Development Center. Springfield, 1977. – Technical Report (RADC)

[MSPB00] MORISIO, M. ; SEAMAN, C. ; PARRA, A. ; BASILI, V.: Investigating and improving a COTS-based software development process. In: *International Conference on Software Engineering (ICSE)*, 2000, S. 32–41

[RKFM01] RITTHOFF, O. ; KLINGENBERG, R. ; FISCHER, S. ; MIERSWA, I.: YALE: Yet Another Learning Environment. In: *Proceedings GIWorkshop-Woche Lernen - Lehren - Wissen*, 2001, S. 84–92

[SL77] STEINHAUSEN ; LANGER: *Clusteranalyse: Einführung in Methoden und Verfahren der automatischen Klassifikation*. Berlin : De Gruyter, 1977

[Wig97] WIGGERTS, A.: Using clustering algorithms in legacy systems remodularization. In: *4th Working Conference on reverse Engineering* WCRE, 1997

[Wur06] WURST, Michael: *The Yale Clustering Plugin: User Guide, Operator Reference, Developer Tutorial*. http://yale.sf.net/: University of Dortmund, 2006