REIHE INFORMATIK
6/99

Eine neue objektorientierte Analysetechnik für die Entwicklung von Audio-Inhaltsanalyse-Algorithmen

P. Tomczyk, S. Pfeiffer und W. Effelsberg
Universität Mannheim
Praktische Informatik IV
L 15, 16
D-68131 Mannheim

Eine neue objektorientierte Analysetechnik für die Entwicklung von Audio-Inhaltsanalyse-Algorithmen

Peter Tomczyk, Silvia Pfeiffer und Wolfgang Effelsberg
Universität Mannheim
{tomczyk,pfeiffer,effelsberg}@pi4.informatik.uni-mannheim.de

Zusammenfassung

Die Entwicklung von Audioanalysealgorithmen ist eine klassische, funktionsorientierte Aufgabe. Trotzdem ist es sinnvoll, solche Algorithmen mit einer objektorientierten Programmiersprache wie C++ zu implementieren, insbesondere um die Wiederverwendung von Routinen durch andere Projektteilnehmer zu erleichtern. Eine klassische objektorientierte Analyse z.B. nach UMT nähert sich dem Problem jedoch aus der falschen Sicht: nämlich aus Datensicht, obwohl die Aufgabe in einer Modellierung von Funktionen besteht. Wir stellen hier eine geeignetere Modellierungstechnik vor, mit der wir sehr gute Resultate erzielt haben.

1 Einleitung

Das Projekt Movie Content Analysis (MoCA) an der Universität Mannheim verfolgt das Ziel, Inhalte aus Ton- und Bildspur von Videomaterial computergestützt zu extrahieren. Dazu werden Algorithmen entwickelt, die auf Audio-, Bild- oder Videodaten arbeiten und Eigenschaften dieser Daten bestimmen. Eigenschaften der Audiospur sind beispielsweise Tonhöhe, Lautstärke, Schärfe, Rauhigkeit oder Klangfarbe, aber auch signalanalytische Eigenschaften wie Nulldurchgangsrate oder Leistung des Signals. Solche Eigenschaften werden eingesetzt, um interessierende Ereignisse zu charakterisieren. Beispielsweise kann man Explosionen erkennen oder Musik- und Sprachsignale. Ähnliches gilt für den Bildbereich. Letztlich kann man die aus Bild- und Tonbereich gewonnenen Erkenntnisse auch kombinieren, um zu umfassenderen Aussagen zu gelangen.

Im Rahmen des MoCA Projekts wurde eine Audio-Bibliothek entwickelt. Sie enthält Algorithmen sowohl zur Handhabung verschiedener Audiodateiformate als auch zur Analyse von Audioeigenschaften. Diese Bibliothek wurde zunächst klassisch in C entwickelt - schließlich ging es um eine Erstellung von Algorithmen

und nicht um eine datenorientierte Anwendung. Es hat sich jedoch herausgestellt, daß auch in einem solchen Fall objektorientiertes Denken bessere Algorithmen hervorbringt.

Dies ist einesteils in der Aufgabe selbst begründet. Beispielsweise müssen Inhaltsanalyse-Algorithmen oft Zustandsinformationen verwalten, um eine Art menschliches Erinnerungsvermögen zu modellieren. Oft kommt es auch vor, daß ein Inhaltsanalyse-Algorithmus von einem anderen abgeleitet werden kann, weil seine Aufgabe der des anderen entspricht, aber enger begrenzt ist. Außerdem gibt es für verschiedene "konkrete" Größen (z.B. Lautstärke, Tonhöhe) unterschiedliche Verfahren, sie zu ermitteln, wofür Polymorphismus am besten geeignet ist. Andernteils gibt es aber auch organisatorische Gründe: in C++ geschriebene Programme sind leichter wiederzuverwenden, leichter zu modifizieren und zu erweitern, und verursachen weniger Namenskonflikte.

Damit war die Entscheidung für den Einsatz einer objektorientierten Programmiersprache gefallen. Beim Entwurf und der Entwicklung neuer Analyse-Algorithmen ist damit aber auch der Einsatz einer objektorientierte Methode sinnvoll. Der Sinn einer puren objektorientierten Analyse (OOA) ist für dieses Problem jedoch fraglich.

Coad und Yourdon [CY91] nennen eine Reihe von Eigenschaften, die eine Analysemethode aufweisen sollte:

- Das Analysemodell muß einen direkten Rückschluß auf das Anwendungsgebiet ermöglichen.
- Das Modell muß stabil gegenüber Veränderungen des Leistungsspektrums des Systems sein.
- Das Verfahren muß beim Umgang mit der Komplexität eine der grundlegenden Methoden des menschlichen Denkens unterstützen.
- Das Verfahren darf keinen Aspekt des Systems vernachlässigen.
- Das Verfahren muß einen reibungslosen Übergang von der Analyse zum Design bieten, insbesondere durch die Verwendung gleicher Darstellungsformen.

Das Ziel der Analyse eines Inhaltsanalyseprozesses ist zunächst die Erfassung seiner Funktion. Liegt eine solche Beschreibung vor, so sollte es möglich sein, auf dieser Basis die Klassen, Beziehungen, Attribute und Methoden zu bestimmen, mit deren Hilfe Algorithmen im Rahmen eines Inhaltsanalyse-Werkzeugs zu implementieren sind. Die OOA verteilt von Anfang an die Systemfunktionalität auf mehrere Klassen. Sie eignet sich damit nicht zur Beschreibung eines komplexen Verarbeitungsprozesses. Andererseits führt die Verwendung einer klassischen Methode, wie der Datenflußmodellierung aus der Strukturierten Analyse, zu einem

Bruch zwischen der Analyse und dem Design. Mittels der Datenflußmodellierung wird der Prozeß zwar gut erfaßt, Klassen und Beziehungen lassen sich aber nur schwer erkennen.

Der im folgenden vorgestellte Ansatz ist ein Versuch, die Vorteile beider Methoden zu vereinen. Er ermöglicht sowohl eine gute Erfassung des Prozesses als auch einen reibungslosen Übergang ins objektorientierte Design. Insofern unterscheidet er sich von den bisher bekannten "hybriden" Ansätzen, welche etwa strukturierte Analyse mit objektorientierten Design und Implementierung vereinen (vgl. z.B. [HS92]).

2 Werkzeuganalyse und die Notation

Das Verfahren gliedert sich in zwei Schritte: im ersten Schritt wird der Prozeß eines Werkzeugs beschrieben, im zweiten Schritt werden die zugehörigen Klassen entworfen. Der erste Schritt kann somit als eine Analysephase bezeichnet werden, der zweite bildet den Übergang ins Design. Die Analyse bedient sich einer speziellen Notation, die an die Datenflußmodellierung angelehnt ist. Das entstandene Analysemodell wird anschließend in ein Klassendiagramm überführt.

In diesem Kapitel wird die Analysephase samt der Notation ausführlich vorgestellt. Das nächste Kapitel schildert den Übergang ins Design. Am Rande werden auch einige Implementationsaspekte angesprochen.

Die Vorgehensweise lehnt sich an die Datenflußmodellierung der Strukturierten Analyse an, bekannt aus DeMarco [DeM79] und Yourdon [You89]. Zunächst wird die gesamte Funktionalität des Werkzeugs als ein einzelner Prozeß dargestellt. Anschließend wird das Modell schrittweise verfeinert, indem der Prozeß in kleinere Teile zerlegt wird (Top-Down-Verfahren). Über den Abbruch bzw. eine weitere Zerlegung der Prozesse entscheiden Faustregeln, die später genauer vorgestellt werden.

Abbildung 1 gibt einen Überblick über die spezielle Notation. Sie erinnert an die Datenflußdiagramme: Prozesse werden durch Kreise repräsentiert, und der Datenfluß wird durch beschriftete Pfeile dargestellt. Es gibt allerdings keine Begrenzer, wie sie aus der Datenflußmodellierung bekannt sind: die einzige Datenquelle und -senke ist der Benutzer des Werkzeugs. Ferner wird deutlich zwischen Daten und den einen Prozeß steuernden Parametern unterschieden. Parameter werden in Ovalen notiert und mit den entsprechenden Prozessen verbunden. Benötigt ein Prozeß bei wiederholter Ausführung Vergangenheitsdaten, so wird für diese das Symbol des Datenspeichers verwendet.

Ein Prozeß kann auf bereits vorhandene Werkzeuge zurückgreifen. Werkzeuge werden als Rechtecke dargestellt, um sie deutlich von den Prozessen zu unterscheiden. Die Werkzeuge sind nicht mit Analysemustern zu verwechseln. Hier handelt es sich um die Wiederverwendung konkreter, früher implementierter Inhaltsanalyse-Algorithmen.

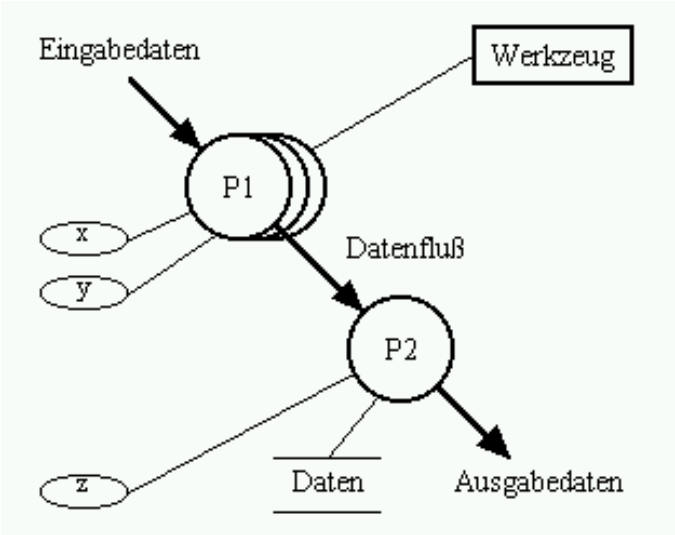

Abbildung 1: Analysediagramme - die Notation

Von einem Prozeß können mehrere Versionen existieren. So kann etwa der Vergleich zweier Audioaufnahmen sowohl auf der Basis ihrer Spektren als auch auf der Basis ihrer Lautstärke erfolgen. Versionen von Prozessen werden durch Mehrfachkreise angedeutet. Das Konzept der Versionen ist etwas abstrakter als die aus der OOA und den E/R-Diagrammen bekannte Generalisierung. Versionen müssen nicht unbedingt zu Vererbungshierarchien führen.

Der Abbruch der Zerlegung und der Übergang zur Klassenbildung erfolgt, wie schon erwähnt, aufgrund von Faustregeln. Der Abbruch findet statt, wenn für jeden Prozeß gilt:

1. Die Anzahl der Parameter, Datenspeicher und Werkzeuge, die mit ihm verbunden sind ist kleiner als E. E wird vor dem Beginn der Analyse festgelegt und beeinflußt entscheidend die Komplexität der Klassen. Im unserem Projekt gilt: $E = 6$.

2. Der geschätzte Umfang der Implementierung des Prozesses ist kleiner als F Funktionen mit maximaler Länge L. F und L sind wie E vor dem Beginn der Analyse festzulegen. In unserem Fall gilt: $F = 5$, $L = 15$ Programmzeilen (LOC). Zur Abschätzung der Anzahl und der Größe von Funktionen kann funktionale Zerlegung in Verbindung mit Pseudocode verwendet werden.

3. Nach einer weiteren Zerlegung des Prozesses wären seine Teile nicht besser wiederverwendbar als er selbst.

Bei der Zerlegung von Prozessen ist folgendes zu beachten: Die Anzahl der Parameter, die mehrere Prozesse gleichzeitig beeinflussen, sollte möglichst gering

sein. Damit wird die Datenredundanz verringert und Koordinationsprobleme vermieden. Dies ist natürlich nicht immer möglich.

3 Klassenbildung und der Übergang zum Design

Das in der Analyse entstandene Modell muß nun in ein Klassendiagramm transformiert werden. Die Umsetzung findet in drei Schritten statt:

1. Festlegung der Klassen.
2. Abbildung aller Symbole auf Attribute, Methoden und Assoziationen.
3. Festlegung von Klassenhierarchien und Einführung parametrisierter Klassen.

Anschließend können allgemeine Abläufe in abstrakten Basisklassen zusammengefaßt werden, um die Entwicklung "verwandter" Werkzeuge zu vereinfachen.

3.1 Festlegung der Klassen

Die Festlegung der Klassen erfolgt auf der Basis der identifizierten Prozesse. Jeder Prozeß ist ein Klassenkandidat. Eine Klasse, die einen Prozeß implementiert, bietet eine (Haupt-)Methode an, die die Verarbeitung startet, und das Ergebnis zurückgibt. Allerdings werden nicht alle Prozesse zu Klassen. Kleine Prozesse, die nur wenige Parameter aufweisen und weder Werkzeuge noch Datenspeicher benötigen, können als "normale" Funktionen implementiert werden. Sie können auch, soweit es die Wiederverwendbarkeit nicht beeinträchtigt, in der Klasse eines größeren Prozesses aufgehen. Ferner kann es sinnvoll sein, Prozesse, die durch dieselben Parameter gesteuert werden, in einer Klasse unterzubringen. Auch Prozesse, die gemeinsam etwa allgemeine Verwaltungsaufgaben erfüllen, "passen" in eine Klasse. Versionen werden zunächst durch eine einzelne Klasse repräsentiert.

3.2 Abbildung von Symbolen auf Attribute, Methoden und Assoziationen

Nachdem die Klassen feststehen, werden die im Analysemodell mit jedem Prozeß verbundenen Symbole auf Klassenattribute, Methodenargumente und Assoziationen abgebildet. Tabelle 1 faßt die sinnvollsten Möglichkeiten zusammen.

3.3 Klassenhierarchien und parametrisierte Klassen

In dem so erhaltenen Klassendiagramm fehlen noch völlig Klassenhierarchien. Diese werden im letzten Schritt eingeführt. Es kann folgende Gründe für die Verwendung einer Klassenhierarchie geben:

Symbol	Im Klassendiagramm	Konstrukt in C++
Eingabedaten	Parameter der Hauptmethode.	Parameter der Hauptmethode.
Ausgabedaten	Rückgabewert der Hauptmethode	Rückgabewert der Hauptmethode oder Parameter (call by reference).
Parameter	Klassenattribut	Klassenelement
Datenspeicher	Klassenattribut oder eine 1:1-Beziehung zu einer Klasse, die die Daten speichert.	Klassenelement oder ein Zeiger auf ein Objekt, das die Daten speichert.
Werkzeug	Eine 1:1-Beziehung zu einer Klasse, die das Werkzeug implementiert.	Zeiger auf ein Werkzeug-Objekt oder Zeiger auf eine Funktion oder ein Aufruf einer Funktion.

Tabelle 1: Symbole im Analysediagramm und im Klassendiagramm.

- Versionen von Prozessen
- Abstrakte Datenbezeichnungen.

Existieren mehrere Versionen eines Prozesses, so können diese mit Hilfe einer Klassenhierarchie realisiert werden. Die Basisklasse enthält in dem Fall alle gemeinsamen Attribute und Beziehungen sowie (zumindest) eine rein virtuelle Methode. Die abgeleiteten Klassen überschreiben diese Methode entsprechend der Prozeßversion, die sie implementieren, und fügen ggf. Verbindungen zu spezifischen Werkzeugen hinzu.

Abbildung 2 zeigt ein Werkzeug zur Berechnung der Energie eines im AudioCache gespeicherten Signalabschnitts. Die Berechnung erfolgt, je nach Version, entweder im Zeit- oder im Frequenzbereich. Die zweite Version benötigt zur Durchführung der Transformation das FFT-Werkzeug.

Es ergibt sich das Klassendiagramm in Abbildung 3. Das Diagramm verwendet die UML-Notation. Die Erklärung der Symbole ist z.B. in [FS97] zu finden.

Hier die Implementation der abstrakten Basisklasse in C++:

```
#include <math.h>

class Energy {
  public:
    virtual ~Energy() { };
    virtual double energy(const AudioCache&) const = 0;
};
```

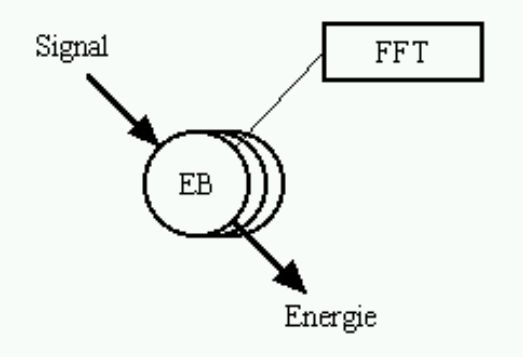

Abbildung 2: Energiemessung - das Analysediagramm.

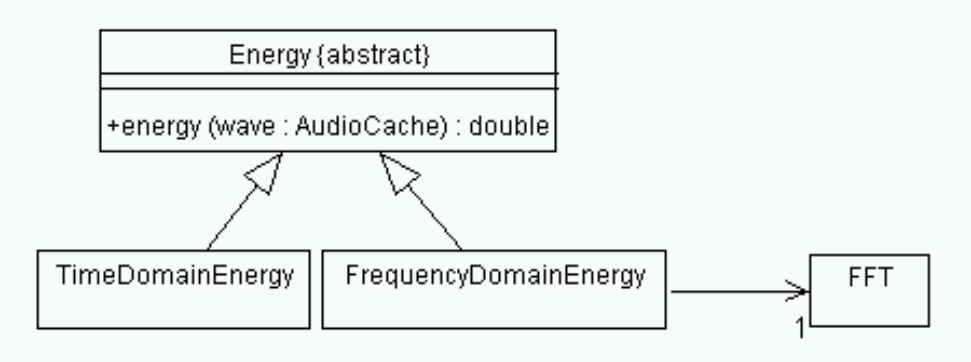

Abbildung 3: Energiemessung - Klassendiagramm in UML-Notation.

Das Werkzeug, das die Energie im Zeitbereich berechnet, überschreibt lediglich die Methode energy():

```
class TimeDomainEnergy : public Energy {
  public:
    double energy(const AudioCache& cache) const;
};

double TimeDomainEnergy::energy(const AudioCache& cache) const {
  double sum = 0.0;
  for (int i = 0; i < cache.size(); i++)
    sum += (1.0 * cache[i] * cache[i]);
  return sum;
}
```

Das zweite Werkzeug benötigt zusätzlich einen Zeiger auf ein Objekt, das die Fourier-Transformation durchführt:

```
class FrequencyDomainEnergy : public Energy {
  public:
    FrequencyDomainEnergy() : _fft(new FFT()) { }
    ~FrequencyDomainEnergy() { delete _fft; }
    FFT& fft() { return *_fft; }
    const FFT& fft() const { return *_fft; }
    double energy(const AudioCache& cache) const;
  private:
    FFT* _fft;
};

double FrequencyDomainEnergy::energy(const AudioCache& cache) const {
  FFT::Spectrum* spec;
  double sum = 0.0;

  _fft->fft(cache, spec);
  for (int i = 0; i < spec->size(); i++)
    sum += (1.0 * norm((*spec)[i]) * norm((*spec)[i]));
  return sum;
}
```

Andere Möglichkeiten für die Umsetzung von Versionen wären etwa Zeiger auf Funktionen als Klassenelemente oder Flags. Im Falle kleiner Prozesse, die als Funktionen implementiert werden, kommt das Überladen der Funktion in Frage.

Die Bezeichner für Eingabedaten, Ausgabedaten, Datenspeicher im Analysemodell haben oft einen sehr abstrakten Charakter. Es wird ein Indikator ermittelt, ein Merkmal abgespeichert etc. Im konkreten Fall sind es nun die Energie eines Signals, seine Lautstärke oder sein Frequenzspektrum. Um das hohe Abstraktionsniveau zu behalten, können Klassenhierarchien eingesetzt werden. Im obigen Beispiel wären eine abstrakte Basisklasse 'Indikator' und von ihr abgeleitete konkrete Indikatoren denkbar. Es gibt allerdings zwei Gründe, die gegen eine Klassenhierarchie sprechen. Zum einen ist der Zusammenhang zwischen Größen wie Energie und Spektrum zu gering, um die Generalisierung zu rechtfertigen. Zum anderen liegen die Datenstrukturen bereits fest, wenn das Werkzeug entwickelt wird. Die Einführung einer Klassenhierarchie ist dann nur bedingt möglich - so wird z.B. die Signalenergie im Normalfall durch einen fundamentalen Datentyp, wie double, repräsentiert und nicht durch eine Klasse.

Die Alternative zur Klassenhierarchien sind parametrisierte Klassen bzw. Funktionen. Das folgende Beispiel zeigt eine Funktion zur Berechnung der euklidischen Distanz zwischen zwei Vektoren. Zunächst die Lösung mit einer Basisklasse Vector:

```
class Vector {
  // Konstruktor, Destruktor usw.
  double operator[](int i) const;
  int size() const;
};

double euclVecDist(const Vector& v1, const Vector& v2) {
  double dist = 0.0;
  for (int i = 0; i < v1.size() && i < v2.size(); i++)
    dist += pow(v1[i] - v2[i], 2);
  return sqrt(dist);
}
```

Alle Größen, für die die euklidische Distanz berechnet werden soll, müssen von Vector abgeleitet werden. Nun die Lösung mittels parametrisierter Funktion:

```
template<class Vector>
double euclVecDist(const Vector& v1, const Vector& v2) {
  double dist = 0.0;
  for (int i = 0; i < v1.size() && i < v2.size(); i++)
    dist += pow(v1[i] - v2[i], 2);
  return sqrt(dist);
}
```

Jetzt ist es lediglich notwendig, daß jede solche Größe die Methode `size()` und den Index-Operator definiert.

3.4 Zusammenfassen allgemeiner Abläufe

Prozesse, die in mehreren Versionen vorkommen, sind meistens viel komplizierter als die im Abschnitt 3.3 vorgestellte Berechnung der Signalenergie. Die funktionale Zerlegung eines solchen Prozesses liefert eine ganze Reihe von Methoden, die typischerweise in zwei Gruppen zerfallen: allgemeine und spezialisierte Methoden. Die allgemeinen Methoden führen solche Aufgaben aus wie das Reservieren und Freigeben von Speicher, Fortschreiben der Vergangenheitsdaten, Verwalten von Containern etc. Sie liegen in einer einzigen Version vor. Ihnen gegenüber stehen die spezialisierten Methoden. Sie arbeiten mit bestimmten Größen auf eine ganz bestimmte Art und Weise. Sie unterscheiden sich deshalb von Version zu Version. Durch das Zusammenfassen und Separieren der allgemeinen Aufgaben in abstrakten Basisklassen ist es möglich, mehrere Versionen eines Werkzeugs mit minimalem Aufwand zu implementieren.

Das nächste Beispiel zeigt ein Werkzeug, das ein Signal abschnittsweise untersucht und jedesmal einen Zähler zurückgibt. Der Zähler besagt, wie viele der

bisher analysierten Abschnitte um weniger als "MaxEntfernung" von einer Bezugsgröße entfernt waren. Der Begriff "Entfernung" soll, genauso wie die Art der Bezugsgröße, erst in einer konkreten Version festgelegt werden. Abbildung 4 zeigt das Analysemodell nach der letzten Zerlegung.

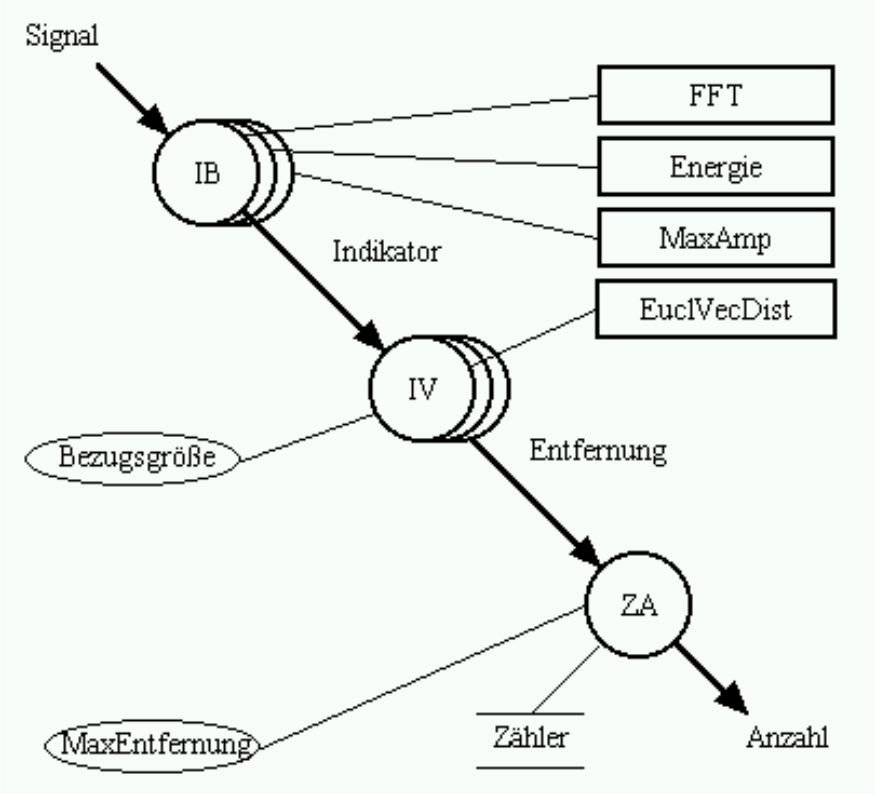

Abbildung 4: Ein Beispiel-Werkzeug mit drei Prozessen - 'Indikator Berechnen' (IB), 'Indikatoren Vergleichen' (IV) und 'Zähler Aktualisieren' (ZA)

Die allgemeinen Methoden von 'IB' (Indikator Berechnen) und 'IV' (Indikatoren Vergleichen) sowie der gesamte Prozeß 'ZA' (Zähler Aktualisieren) werden in einer einzelnen Basisklasse `Counter` zusammengefaßt. Die Klasse erhält einen Template-Parameter, um die abstrakte Datenbezeichnung "Indikator" zu realisieren. Hier die Definition der abstrakten Basisklasse:

```
template<class Indicator>
class Counter {
  public:
    Counter(const Indicator& ind, double dist)
      : _ind(ind), _maxDist(dist), _count(0) { }
    virtual ~Counter() { }
    int next(const AudioCache&);
```

```
  protected:
    Indicator _ind;
    double    _maxDist;
    int       _count;
    virtual Indicator calcInd(const AudioCache&) = 0;
    virtual double calcDist(const Indicator&, const Indicator&) = 0;
};

template<class Vector>
double euclVecDist(const Vector&, const Vector&);
```

Der allgemeine Teil des Prozesses kann bereits in der Basisklasse implementiert werden. Er besteht lediglich aus der Methode `next()`. Hier wird der Indikatorwert zu dem gegebenen AudioCache angefordert und anschliessend mit dem Bezugswert verglichen. Ist der Abstand kleiner als der Grenzwert, so wird der Zähler inkrementiert. Die Methode benötigt kein spezielles Wissen über den verwendeten Indikator:

```
template<class Indicator>
int Counter<Indicator>::next(const AudioCache& cache) {
  Ind ind = calcInd(cache);
  if (calcDist(ind, _ind) < _maxDist)
    _count++;
  return _count;
}
```

Hier die Definition eines konkreten Werkzeugs sowie die Implementation der speziellen Methoden `calcInd()` und `calcDist()`. Sie berechnen den Wert des Indikators bzw. ermitteln den Abstand zwischen diesem Wert und der Bezugsgröße:

```
class CounterUsingSpectrum : public Counter<FFT::Spectrum> {
  public:
    CounterUsingSpectrum(const FFT::Spectrum& ind, double dist)
      : Counter(ind, dist), _fft(new FFT(ind.size())) { }
    ~CounterUsingSpectrum() { delete _fft; }
  protected:
    FFT* _fft;
    FFT::Spectrum calcInd(const AudioCache&);
    double calcDist(const FFT::Spectrum&, const FFT::Spectrum&);
};
```

```
FFT::Spectrum CounterUsingSpectrum::calcInd(const AudioCache& cache) {
  FFT::Container spec(_fft->winSize());
  _fft->fft(cache, &spec);
  return spec;
}

double CounterUsingSpectrum::calcDist(const FFT::Spectrum& spec1, const
FFT::Spectrum& spec2) {
  return euclVecDist(spec1, spec2);
}
```

4 Diskussion des Ansatzes

Die hier vorgestellte Methode stellt nur für ganz spezielle Fälle eine Alternative zur objektorientierten Analyse dar. Sie unterstützt die Entwicklung erweiterbarer und wiederverwendbarer Werkzeuge im Rahmen von Bibliotheken und Systemen die sich durch folgende Eigenschaften auszeichnen:

- Trennung von Daten und Funktionen.
- Hohe Komplexität und hohes Abstraktionsniveau der Funktionen.

In solchen Fällen spielen die Unzulänglichkeiten der Methode keine große Rolle:

1. Keine Modellierung der Datenstrukturen:
   Die Trennung von Daten und Funktionen ist nur dann sinnvoll, wenn die meisten Algorithmen mit den meisten Datenstrukturen umgehen müssen. In solchen Fällen werden zunächst die Mechanismen festgelegt, nach denen die Algorithmen auf die Daten zugreifen (z.B. die Iteratoren in der C++-STL, vgl. [Str98], oder die Klasse AudioCache in unserem Fall). Die Entwicklung beider Teile erfolgt danach unabhängig voneinander. Das Fehlen einer genauen Beschreibung der Datenstrukturen im Analysediagramm eines Werkzeugs ist also kein gravierender Nachteil.

2. Beschränkte Größe der Diagramme wegen abnehmender Übersichtlichkeit:
   Die Methode wird eingesetzt bei der Entwicklung einzelner Werkzeuge und nicht eines kompletten Systems. Die Größe der Diagramme hält sich deshalb in Grenzen, und sie bleiben übersichtlich.

3. Probleme beim Einsatz moderner Softwarelebenszyklus-Modelle:
   Durch die Verwendung zweier unterschiedlicher Notationen ist der Bruch zwischen Analyse und Design nicht ganz behoben. Nur der Übergang von der Analyse ins Design ist einfach, denn hierfür existieren Regeln, nicht

aber der umgekehrte Weg. Moderne Softwarelebenszyklus-Modelle verwenden iteratives Vergehen, sie benötigen also einen fließenden Übergang in beide Richtungen. Die Iterationen beziehen sich aber vielmehr auf das Gesamtsystem als auf solch kleine Teile wie die einzelnen Werkzeuge, die ja in einem "Durchgang" entwickelt werden.

5 Zusammenfassung

In diesem Beitrag wurde eine neue objektorientierte Analysetechnik für die Entwicklung von Audioanalyse-Algorithmen vorgestellt. Sie ist geeignet für alle Entwicklungsprozesse, deren Ziel hauptsächlich die Erstellung von komplexen Funktionen ist, während die Variablilität der zu handhabenden Daten sehr gering ist. Unsere Erfahrungen beim Einsatz der Analysetechnik waren sehr positiv - sie unterstützt den Entwickler auch darin, sich über die Details der Funktionen und ihre Interaktion ein klares Bild zu machen. Wir werden die Technik auch bei zukünftigen Studien- und Diplomarbeiten, die in diesem Bereich durchgeführt werden, einsetzen.

Literatur

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[DeM79] T. DeMarco. *Structured Analysis and Systems Specification.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

[FS97] M. Fowler and K. Scott. *UML Distilled. Applying the Standard Object Modelling Language.* Addison Wesley, 1997.

[HS92] B. Henderson-Sellers. *A Book of Object-Oriented Knowledge - Object-Oriented Analysis, Design and Implementation: A New Approach to Software Engineering.* Prentice Hall, New York, 1992.

[Str98] B. Stroustrup. *The C++-Programming Language.* Addison Wesley, 3rd edition, 1998.

[You89] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, Englewood Cliffs, New Jersey, 1989.