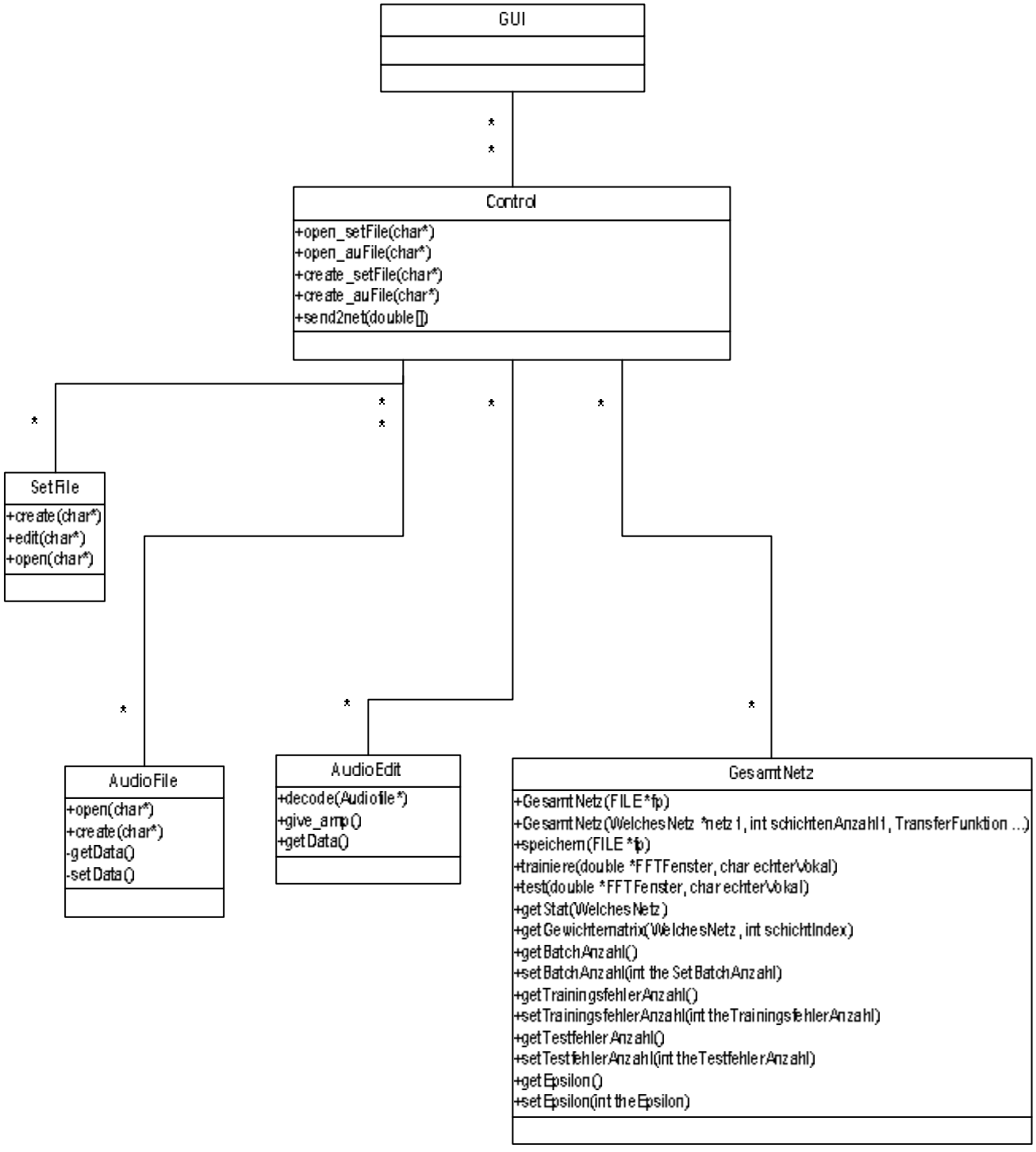
Klassenschema:

dieses Diagramm soll kurz die PUBLIC Methoden der einzelnen Klassen erkennbar machen und auch den Weg die Kommunikationswege zwischen den Klassen verdeutlichen.



Klassen (1):
(nähere Beschreibung des Typs)

class GUI::QWidget
dieses Widget beinhaltet alle anderen Anzeigeelemente/-objekte
Es stellt die Schnittstelle zwischen dem Benutzer und dem Programm dar.
Alle grafischen Objekte werden in er GUI vereint. Jegliche QFileOpenDialoge werde aus der GUI heraus aufgerufen.
Die GUI wird zu einer Ansammlung aller Objekte der grafischen Darstellung.
Zur Verarbeitung werden die „slots“ der Klasse Control genutzt.
Es gibt keine andere direkte Schnittstelle mit der GUI als über der Klasse Control!

class Control
Diese Klasse dient als Schnittstelle zwischen der GUI und den restlichen Klassen des Projektes.
Controll bietet größtmöglichsten Komfort bei der Realisierung der einzelnen Funktionen.

class SetFile
Diese Klasse behandelt jegliches Dateihandling mit den Set-Dateien. Eine Set-Datei ist eine speziell formatierte Datei mit einer Liste von Dateien inklusive deren Pfadangaben.
Diese Klasse kann die Dateien auch bearbeiten, bzw. neue erzeugen.

class AudioFile
Diese Klasse behandelt jegliches Dateihandling mit den au-Dateien.
Erstellen und öffnen ist realisiert. Außerdem kann der Header zur weiteren Auswertung übergeben werden. Diese Klasse überprüft jedoch schon selbstständig beim Öffnen, ob die Datei den geforderten Format(en) der Aufgabenstellung entspricht.

class AudioEdit
Diese Klasse beinhaltet alle Bearbeitungsprozesse zur Aufbereitung der Daten, um diese an das Neuronennetz zu übergeben.
Außerdem wird nach der Dekodierung, mittels eines Canvas Objektes die Wellenkurve der Audiodatei visualisiert/angezeigt.

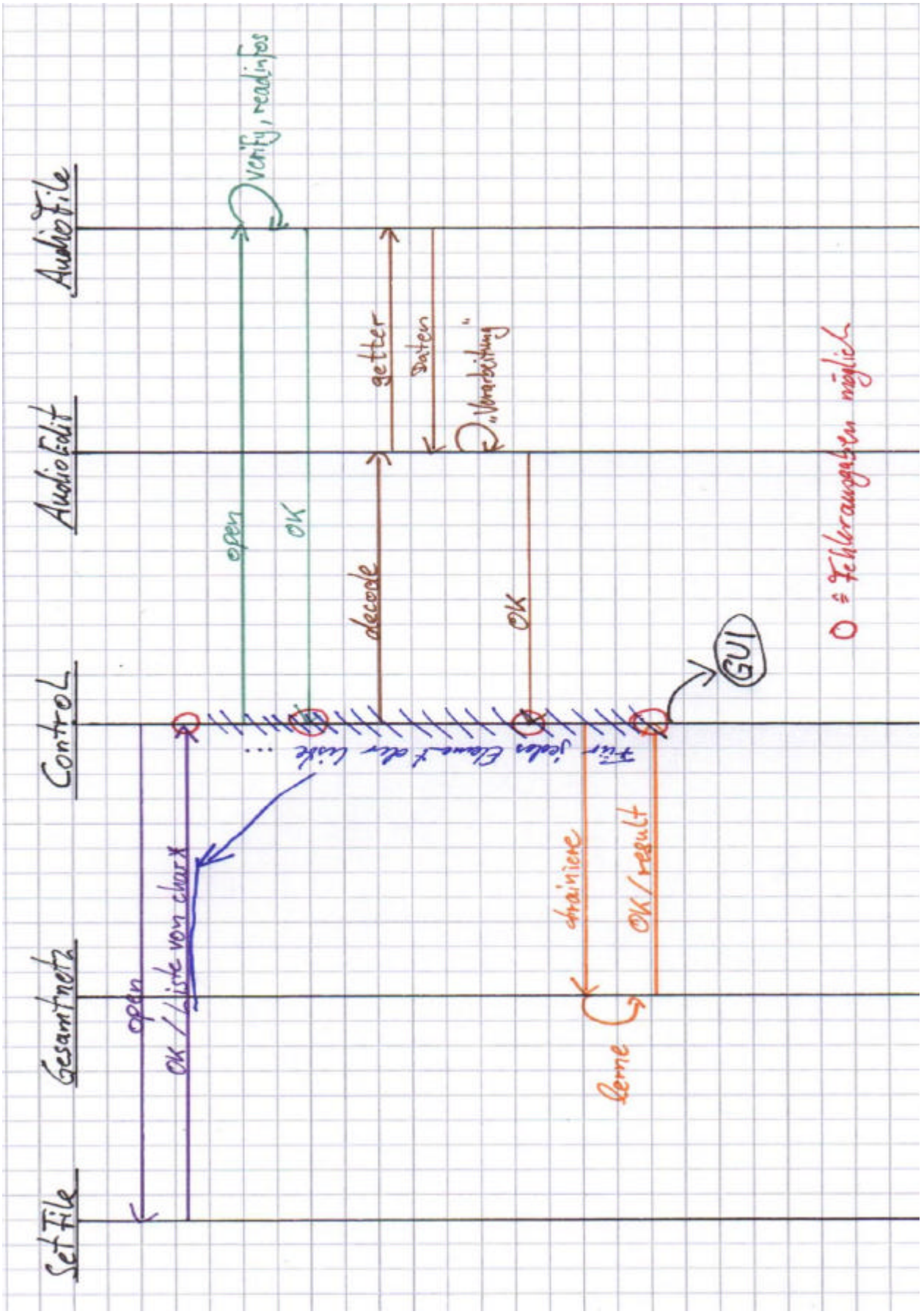
class Gesamtnetz
Dies ist das neuronale Netz. Es beherrscht die neuronalen Algorithmen und jegliche verbundenen Lernprozesse. Außerdem werden Methoden zum Setup des Netzes angeboten. Einstellungen und erlernte Fähigkeiten können entsprechend abgespeichert werden.

Klassen und Methoden:
(ein kleines Lexikon für unsere Projektrealisierung)

Klasse	PUBLIC Methoden	Kommentar
GUI	-	Grafische Oberfläche -keine Planungen bisher-
Control	open_setFile(char*)	Öffnen einer SET Datei, zum Einspielen in das neuronale Netz
	open_auFile(char*)	Öffnen einer einzelnen AU Datei, zum Einspielen in das neuronale Netz
	create_setFile(char*)	Eine auswahl von AU dateien zu einer SET Datei zusammenstellen, oder Bearbeitung einer existierenden SET Datei
	create_auFile(char*)	Aufnahme einer neuen AU Datei
	sent2net()	Sendet alle aufbereiteten Daten an das neuronale Netz
AudioFile	open(char*)	Öffnet eine AU Datei und checkt den Header nach Gültigkeit bzgl. der Vorgaben
	create(char*)	Nimmt neue AU Datei mittels *audiorecord* auf (intern: commad-line Operation) Parameter werden den Projektvorgaben entsprechend gesetzt.
	getter() Methoden	Liefern wichtige Daten der AU Datei für unsere Verarbeitung
AudioEdit	decode(audioFile*)	Dekodiert die AU Datei
	double[] give_amp()	Übergibt Amplitudenwerte [-1.0…1.0] der dekodierten AU Datei (zur Darstellung)
	double[] getData	Übergibt dekodierte Daten der AU Datei
Gesamtnetz	Puuuh…	
	…much2do…	

Bitte jeder für seine Klasse(n) ergänzen

Time-Line Diagramm(e):



allgemeine Infos:

Fehlercodes:

Wert	Bedeutung	Beispiel
<0	Nicht definiert…	Grober Bug
0	// leerer Konstruktor	Datei öffnen Vorverarbeitung Lernprozeß
1	OK, Vorgang erfolgreich durchgeführt	Datei öffnen Vorverarbeitung Lernprozeß
>1	Problemspezifischer Fehlercode der i.a. nur bei Ausnahmesituationen auftritt	debugging

Genormte Fehlercodes vereinfachen das Debugging und Verstehen des Codes für alle Entwickler. Die Benutzung von Methoden verschiedener Klassen wird einheitlich.

.h Dateien:

Gesamtnetz.h:

```
typedef struct {
  int trainingsLaenge;              // die Laengen von den Arrays
  int testLaenge;
  double *trainingsFehler;
  double *testFehler;
} Statistik;

typedef struct {
  // die Pseudowahrscheinlichkeiten von den jeweiligen Moeglichkeiten [0.0,1.0]:
  double a, e, i, o, u;
  char gewaehlt;      // == 'a', 'e', 'i', 'o' oder 'u', je nachdem,
                      // welches die groesste Pseudowahrscheinlichkeit hat
  double sicherheit; // enthaelt die Pseudowahrscheinlichkeit von gewaehlt
  bool neuesPlot;     // == 1 wenn die FehlerKurve einen neuen Punkt gerade
                      // bekommt haben (fuers Neuzeichnen), sonst == 0
} Vokal;

typedef struct {
  int neuronenAnzahl;
  int inputDim;
  double **gewichte;   // wobei gewichte[0-(neuronenAnzahl-1)][0-(inputDim-1)]
  double *schwellen;   // dies hat Laenge neuronenAnzahl
} GUS;   // GUS steht fuer GewichteUndSchwellen

typedef enum {Gesamt, Aouei, Ouvoid, Eivoid} WelchesNetz;
typedef enum {Tanh, Logi} TransferFunktion;

class GesamtNetz {
 protected:
  Netz *Aouei, Ouvoid, Eivoid;
  bool trainiert;
  int batchAnzahl;
  int trainingsfehlerAnzahl;
  int testfehlerAnzahl;
  double epsilon;
 public:
  // zuerst ein Konstruktor, wobei der Parameter ein Filepointer ist.
  // Das Gesamtnetz wird von der Datei geladen (drei Netze, trainiert, batchAnzahl, Geschichte)
  // Es wird von mir angenommen, dass fp schon geoeffnet wurde und in dem richtigen Modus ist.
  // Ihr sollt nachher fclose(fp) durchfuehren! Ich mache das nicht!
  GesamtNetz(FILE *fp);

  // und dann ein Konstruktor, der ein untrainiertes Netz schoepft.
  // Die meisten von diesen Parametern koennen nicht spaeter geandert werden, weil
  // es zum Beispiel bedeutungslos waere, ein paar neuen Schichten mit untrainierten
  // Gewichten/Schwellen zu einem trainierten Netz hinzufuegen.
  //
  // Fuer jedes der Netze 1,2,3 soll es spezefiziert werden:
  //   netz: welches es eigentlich ist (Gesamt, Aouei, Ouvoid oder Eivoid)
  //   schichtenAnzahl: wie viele Schichte es besitzt
  //   tf: welche TransferFunktion pro Schicht benutzt werden soll (Tanh oder Logi)
  //   neuronenAnzahl: wie viele Neuronen es pro Schicht geben soll
  // anschliessend auch einmal:
  //   batchAnzahl: ob Online-learning benutzt werden soll (=1) oder die Anzahl der Schritte
  //                zwischen Updates = die Laenge der Epoche (S. 20)
  //   trainingsfehlerAnzahl: wie viele Trainingsdaten angekuckt werden, vor ein neuer Datenpunkt
  //                          zu den Statiken zugefuegt werden soll. (ankucken: trainiere)
```

```
  //   testfehlerAnzahl:      wie viele Testdaten angekuckt werden, vor ein neuer Datenpunkt
  //                          zu den Statiken zugefuegt werden soll. (ankucken: test)
  //   oft ist batchAnzahl = trainingsFehlerAnzahl = testFehlerAnzahl = Laenger der Epoche
  //   epsilon: siehe S. 20, [10^-3 0.5], vom Benutzer einstellbar
  // Es muss natuerlich jedes der drei Netzetypen genau einmal in diesen Parametern auftreten.
  GesamtNetz(WelchesNetz *netz1, int schichtenAnzahl1, TransferFunktion *tf1, int *neuronenAnzahl1,
             WelchesNetz *netz2, int schichtenAnzahl2, TransferFunktion *tf2, int *neuronenAnzahl2,
             WelchesNetz *netz3, int schichtenAnzahl3, TransferFunktion *tf3, int *neuronenAnzahl3,
             int batchAnzahl, int trainingsFehlerAnzahl, int testfehlerAnzahl, double epsilon);

  // Das Gesamtnetz wird in dieser Datei gespeichert, damit es spaeter durch GesamtNetz(char *)
  // geladen werden kann. Es wird angenommen, dass fp schon geoeffnet wurde und in dem richtigen
  // Modus ist (und leer ist).
  // Ihr sollt nacher fclose(fp) durchfuehren! Ich mache das wieder nicht!
  void speichern(FILE *fp);

  // Trainiere das Gesamtnetz mit der Ausgabe von der Vorverarbeitung.
  // Die Laenge von FFTFenster muss gleich der neuronenAnzahl[0] von den drei Netzen sein.
  // Falls ein Punkt zu dem Plot zugefuegt werden soll, wird Vokal.neuesPlot == 1
  // Vokal.gewaehlt und Vokal.sicherheit koennen falsch sein: welches ebenso wahrscheinlicher
  // ist, wenn die Netze noch nicht gut trainiert sind.
  // Damit der Fehler berechnet werden kann, muss natuerlich auch den richtigen Vokal eingegeben
  // werden (als 'a', 'e', 'i', 'o' oder 'u').
  Vokal trainiere(double *FFTFenster, char echterVokal);

  // Das gleiche wie trainiere, aber die Schwellen und Gewichten werden nicht geaendert.
  // Der Testfehler anstatt der Trainingsfehler wird natuerlich hier berechnet.
  // Hinweis: Diese Fehler werden aber hier nicht zurueckgegeben. Sie muessen durch getStat()
  // zugegriffen werden.
  Vokal test(double *FFTFenster, char echterVokal);


  // Getter/Setter:
  // Damit ihr die Plots darstellen koennt (wie auf S. 13)
  Statistik getStat(WelchesNetz);
  // Um die Werte fuer ein Plot wie auf S. 21 zu bekommen
  // Fuer ein Netz mit 3 Schichten (wie auf S. 16) ist die 0. Schichte trivial aber
  // trotzdem darstellbar. Die 1. und 2. Schichten sind mehr interessant.
  GUS getGewichtematrix(WelchesNetz, int schichtIndex);

  bool trainiert(void){ return trainiert; }
  int getBatchAnzahl(void){ return batchAnzahl; }
  void setBatchAnzahl(int theSetBatchAnzahl):setBatchAnzahl(theSetBatchAnzahl){}
  int getTrainingsfehlerAnzahl(void){ return trainingsfehlerAnzahl; }
  void setTrainingsfehlerAnzahl(int
theTrainingsfehlerAnzahl):trainingsfehlerAnzahl(theTrainingsfehlerAnzahl){}
  int getTestfehlerAnzahl(void){ return testfehlerAnzahl; }
  void setTestfehlerAnzahl(int theTestfehlerAnzahl):testfehlerAnzahl(theTestfehlerA nzahl){}
  int getEpsilon(void){return epsilon; }
  void setEpsilon(int theEpsilon):epsilon(theEpsilon){}
};
```

AudioEdit.h:

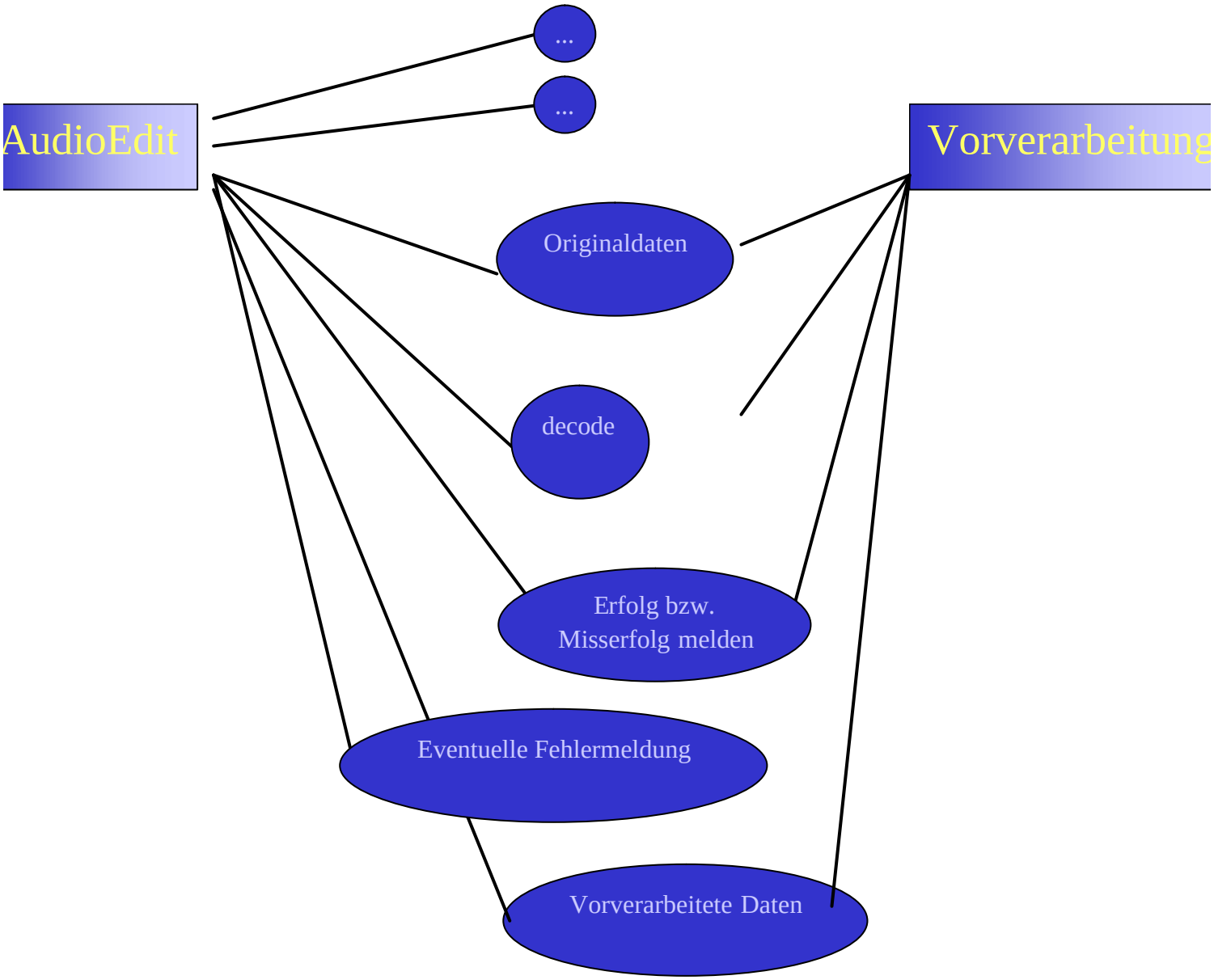
```
class AudioEdit
{
	public:
		.
		.
		.
		void decode(AudioFile* af);
		/* Beispielrumpf
		{
			Vorverarbeitung *vv = new Vorverarbeitung();
			vv.setDataOriginal(data);
			vv.setDataLength(length);
			vv.setRedundance(red);
			vv.setScaleFactor(sf);
			vv.runDecode();
			if (vv.getUmgewandelt()==true)
			{
				this.dataOriginal = vv.getDataOriginal(); // sollte überflüssig
sein
				this.dataFFT = vv.getDataFFT();
				this.data = vv.getData();
			}
			else
			{
				Fehlermeldung: anderes Signal verwenden!
			}
		}*/
		.
		.
		.

	private:
		.
		.
		.
		// Daten zum Zeichnen
		double[] dataOriginal; // Originaldaten
		double[] dataFFT; // Daten nach FFT (positive Werte)
		double[] data; // vollständig vorverarbeitete Daten
		.
		.
		.
}
```

Vorverarbeitung.h:

```
// Klasse zur Audio VV
class Vorverarbeitung
{
	public:
		Vorverarbeitung(); // Konstruktor
		~Vorverarbeitung(); // Destruktor

		void setDataOriginal(double[] data); // setzt die Originaldaten in den Vektor dataOriginal zur Weiterverarbeitung
		void setDataLength(long length); // setzt die Länge des Originaldatenvektors in dataLength

		void setRedundance(int red); // setzt die vom Benutzer eingestellte Redundanz - für reduceData
		void setScaleFactor(double sf); // setzt den vom Benutzer eingestellten Skalierungsfaktor - für reduceData

		void runDecode(); // startet den Vorverarbeitungsvorgang

		double[] getDataOriginal(); // zum Auslesen der Originaldaten, um diese zu zeichnen (womöglich überflüssig)
		double[] getDataFFT(); // zum Auslesen der Daten nach der FFT (Betrag)
		double[] getData(); // zum Auslesen der vollständig vorverarbeiteten Daten

		bool getUmgewandelt(); // gibt zurück, ob die Umwandlung der Daten Erfolg hatte
	private:
		bool umgewandelt = true; // falls false, dann ist Vorverarbeitung gescheitert (z.B. wenn Audiodatei zu kurz, oder Amplitude über 50% der Maximalamplitude nicht gefunden)
		long dataLength; // Länge der Originaldaten
		double[] dataOriginal; // Originaldaten
		double[] dataFFT; // Daten nach FFT (positive Werte)
		double[] data; // vollständig vorverarbeitete Daten
		int redundance; // Überlappung in % (0 bis 95) - für reduceData
		double scaleFactor; // Streckungsfaktor - für reduceData

		double[] semiinv(double[] data); // 1.Schritt: Semiinvertierung (spiegeln der positiver Werte bzgl. 64-Achse
		double[] cut8192(double[] data); // 2.Schritt: Ausschneiden der 8K Daten, welche betrachtet werden sollen
		double[] expoScale(double[] data); // 3.Schritt: exponentielle Skalierung: x´ = (2048)^(x/127)
		double[] runFFT(double[] data); // 4.Schritt: startet mit den bisher gewonnenen Daten die FFT -> 16K Daten
		double[] divVector(double[] data); // 5.Schritt: teilt den aus der FFT gewonnenen Vektor in der Mitte (wegen reller FFT) -> 8K Daten
		double[] computeAmpl(double[] data); // 6.Schritt: Amplituden berechnen: Ampl = (Re^2+Im^2)^(1/2) -> 4K Daten
		double[] reduceData(double[] data); // 7.Schritt: Daten auf ca. (ganz wenig) Elemente reduzieren (mithilfe von redundance und scaleFactor)
};
```

Ablauf zwischen AudioEdit und Vorverarbeitung:

Use-Case Diagramm



SetFile.h:

```
/*
        Klasse, die der Verwaltung von *.set Dateien dient
        In den Methoden werden später in erster Linie irgendwelche
        Dateioperationen implementiert

        Eine *.set Datei ist eine ASCII-Datei, die die absoluten Pfade von mehreren *.au Dateien
        enthält. Dies dient der besseren Verwaltung von größeren Test-und Trainingsdatensätzen.


        Aufbau einer *.set Datei:

        In der ersten (0-basiert) Zeile steht die Anzahl der aktuell enthaltenen Pfade.
        Dann folgt in jeder Zeile genau ein Pfad.
        Eventuell ist die Reihenfolge der Pfade alphabethisch.


        Ein paar der Methoden sind wahrscheinlich unnötig, aber es stellt sich wohl erst bei der
        Implementierung raus, welche Operationen wir letztendlich wirklich brauchen.
*/


class SetFile
{
        public:
                // leerer Konstruktor
                SetFile();

                // Konstruktor mit Namen der *.set Datei
                SetFile(char ** SetFileName);

                // Destruktor
                ~SetFile();

                // setzt SetFileName, also den Namen der *.set Datei, die verwaltet werden soll
                setFileName(char SetFileName);

                // fügt den Pfad einer *.au Datei am Ende der *.set Datei ein;
                // wenn wir uns für alphabetische Anordnung entscheiden, wird der
                // Pfad an der richtigen Stele einsortiert
                // Rückgabewert ist -1 bei Fehler
                int add(char ** AuFileName);

                // löscht den Pfad einer *.au Datei aus der *.set Datei
                // diese Operation ist weniger aufwendig, wenn das Ganze alphabetisch sortiert ist
                // Rückgabewert ist -1 bei Fehler
                int rem(char ** AuFileName);

                // fügt Pfad in i-ter Zeile der *.set Datei ein
                // der Rest rutscht eins weiter nach unten (die Zeile wird also nicht überschrieben)
                // macht wohl auch nur Sinn, wenn alphabetisch sortiert
                // Rückgabewert ist -1 bei Fehler
                int add(int zeile);

                // löscht die i-te Zeile der *.set Datei
                // macht wohl auch nur Sinn, wenn alphabetisch sortiert
                // Rückgabewert ist -1 bei Fehler
                int rem(int zeile);
```

```
        // liefert den Pfad in der i-ten Zeile
        // liefert NULL bei Fehler
        char ** getPath(int i);

        // liefert den Pfad, auf den Zeiger ptr gerade weist
        // und zählt dann den Zeiger auf den nächsten Pfad hoch
        // liefert NULL bei Fehler
        char ** getPath();

        // liefert die Anzahl der Pfade in der Datei
        int getDim();

        // liefert die Zeile, auf die ptr aktuell zeigt
        int getPtr();

        // setzt den Zeiger ptr auf eine bestimmte Zeile
        // Rückgabewert ist -1 bei Fehler (z.B. wenn auf nicht existierend Zeile gesetzt)
        int setPtr(int i);

        // setzt den Zeiger ptr auf 0
        // vielleicht überflüssig wegen setPtr(int i) davor
        void reset();

    private:

        // gibt an, wieviele Pfade die *.set Datei enthält
        // entspricht der 1. Zeile der *.set Datei
        int dim;

        // der Name der *.set Datei, die verwaltet wird
        char ** SetFileName;

        // Zeiger auf die aktuelle Zeile
        int ptr;

        // schreibt die Dimension in die *.set Datei
        // Rückgabewert ist -1 bei Fehler
        int writeDim(int dim);

};
```

AudioFile.h

```
class AudioFile
{

public:
                // leerer Konstruktor
                AudioFile();

                // Konstruktor mit Namen der *.au Datei
                AudioFile(char ** SetAudioFileName);

                // Destruktor
                ~SetAudioFile();

                // setzt SetAudioFileName (der Name der *.au Datei)
                setAudioFileName(char SetAudioFileName);

                //Dateinamen abfragen
                char* getAudioFileName();

                //Vokal abfragen
                char getVokal();

                //den Datenblock der .au Datei abrufen
                double* getAudioFileDataBlock();

                //die Länge des Datenblocks abfragen
                int getAudioFileLength();

                //eine neue .au Datei erzeugen
                //wird letzlich über command-line erzeugt (Programm audiorecord)
                create(char* FileName)

private:
                //prüft .au Datei nach Korrektheit im Format
                int verify();

                //speichert alle aus dem Header gewonnenen Infos bzgl. der Verarbeitung in die
                //entsprechenden VARs ab
                void read_infos();

                double* data[];  //Datenblock
                int length;      //Länge des Datenblocks
                char voc;        //der Vokal der Datei
                char* name;      //und schließlich auch der Name der Datei


}
```