**Technische Universität Berlin**
Institut für Sprache und Kommunikation
**Fachgebiet Audiokommunikation**

Masterarbeit

---

# Granularsynthese mit Wavesets für Live-Anwendungen

---

eingereicht von

Fabian Seidl
[redacted]

Eingereicht am: **16. April 2016**
Studiengang: Master Informatik

Gutachter: **Prof. Dr. Stefan Weinzierl, Prof. Dr.-Ing. Sebastian Möller**
Betreuer: **Dr. Till Bovermann (Universität der Künste Berlin)**

# Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den

Unterschrift

# Zusammenfassung

Diese Arbeit umfasst die Entwicklung einer Granularsynthese mit Wavesets für Live-Anwendungen. Sie wurde spezifisch zur Verwendung mit anderen Musikinstrumenten ausgelegt. Das als SuperCollider-Erweiterung realisierte System bietet die Waveset-Resynthese eines Live-Signals. Aus dem Eingangssignal werden fortlaufend Wavesets extrahiert und gespeichert. Daraus ausgewählte Wavesets werden zu einem neuen Signal zusammengesetzt, synthetisiert. Ausgehend von der Echtzeitanalyse eines Eingangssignals wird untersucht, welche Waveset-basierten Resynthesemethoden sich anbieten und inwieweit sie für den Live-Einsatz geeignet sind. Die Anforderungen für die Resynthese wurden schrittweise erfasst und in Zwischenversionen implementiert. Realisiert wurden zwei grundlegende Synthesemethoden. Erstens eine nahtlose Aneinanderreihung der Wavesets, die *kontinuierliche Synthese*. Zweitens eine trigger-basierte Synthese, welche auch das parallele Abspielen mehrerer Wavesets erlaubt, die sogenannte *ereignisbasierte Synthese*. Letztere ermöglicht sowohl eine synchrone als auch eine asynchrone Synthese. Für die Auswahl von Wavesets wurde ein halbautomatisches Verfahren, basierend auf zuvor berechneten Merkmalen entwickelt. Für die Merkmale werden Sollwerte angegeben und das Waveset mit der geringsten Abweichung zu diesen Sollwerten ausgewählt. Das Waveset-basierte Resyntheseverfahren wurde in zwei Performances erfolgreich im Live-Einsatz erprobt. Es erlaubt dem Künstler neue Klangstrukturen zu schaffen, die einerseits Klangfarben des Eingabesignals erhalten, andererseits eine charakteristische Klangeigenschaft entwickeln, die als künstlich/digital empfunden wird.

# Abstract

In this thesis a granular synthesis using wavesets for live applications is developed. It is specific designed for the usage with other musical instruments. The system is realised as a SuperCollider extension and features a waveset resynthesis of a live signal. Wavesets are extracted continuously from the input signal and stored in a buffer. From that buffer, wavesets are selected and recomposed to a new sound. Based on the realtime analysis of an input signal, it is investigated which methods of synthesis are suitable for the live application. The requirements for the resynthesis were iterative collected, while the software was implemented in incremental steps. Two basic synthesis methods were realised. In the first one the wavesets are concatenated seamless, called *continuous synthesis*. Secondly a trigger based synthesis, which also allows a parallel playback of several wavesets, called *event based synthesis*. The latter is suitable for both synchronous and asynchronous synthesis. For the selection of wavesets an semiautomatic process based on extracted features was developed. Target values can be determined for the features and the waveset with the smallest difference to this target values will be selected. The waveset based resynthesis was proven successful for the live application in two performances. It enables the artist to create new sound structures based on the input signal. This structures have certain acoustic colours of the input signal on the one hand and a characteristic digital/artificial perceived component on the other.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

Granularsynthese ist eine Technik zur Klangerzeugung und Bearbeitung durch kurze Klangfragmente, sogenannter *Grains* (englisch für Korn oder Quäntchen). Nach den Pionierarbeiten von Denis Gabor [17] und Ianis Xenakis [47] erhielt die Idee der Erzeugung von Musik aus diesen kleinen Partikeln Einzug in den Mainstream elektronischer Musik [12]. Viele kommerzielle Software-Synthesizer sowie Software zur Musikproduktion nutzen heute Granularsynthese. Weit verbreitet ist dabei die Resynthese von aufgezeichnetem Audiomaterial. Dabei werden aus dem Audiomaterial durch eine Analyse Grains extrahiert (Granulierung) und anschließend in einer Synthese zu einem neuen Klang zusammengesetzt (siehe Abb. 1.1). Die Synthese wurde erstmals 1986 von Barry Truax in Echtzeit realisiert [42] und kurz darauf auch die Granulierung in Echtzeit [43]. Obwohl die Echtzeit-Resynthese heute technisch keine Herausforderung mehr darstellt findet man noch relativ wenige Anwendungen, welche die Resynthese eines Live-Signals durchführen.

Eine spezielle Art der Grains sind die Wavesets nach Trevor Wishart [45]. Sie werden von einem Eingangssignal extrahiert indem es an den Nullstellen geteilt wird. Wishart nutzt sieht sie vor allem als Elemente für Transformationen, bei denen er Audioaufzeichnungen durch Neuanordnung der Wavesets bearbeitet. Die Wavesets lassen sich jedoch auch vielseitiger für unterschiedliche Synthesemethoden einsetzen [12]. Im Rahmen des Projekts 3DMIN (Design, Development and Dissemination of New Musical Instruments)[1] [9] soll erprobt werden inwiefern sie sich eine Resynthese mit Wavesets live in Kombination mit neu entwickelten Instrumenten einsetzen lässt.

[1] `http://www.3dmin.org`

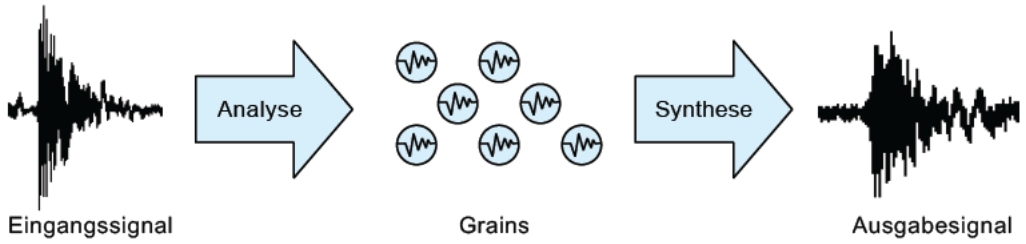


**Abbildung 1.1:** Allgemeiner Ablauf einer Granular-Resynthese. Ein Eingangssignal wird mit einer Analyse granuliert, die extrahierten Grains werden anschließende mit einer Synthese zu einem neuen Signal zusammengesetzt.

Hierfür wird eine Implementierung von Granular-Analyse sowie Synthesemethoden mit Wavesets für die Software SuperCollider[2] geschaffen, die für den Live-Einsatz geeignet ist. Um eine Resynthese von Live-Signalen zu ermöglichen, muss diese sowohl die Granulierung als auch die Synthese in Echtzeit unterstützen. Es wird untersucht welche Synthesemethoden für einen Live-Auftritt geeignet sind. Nicht alle Techniken für die Verarbeitung von aufgezeichnetem Audiomaterials lassen sich direkt auf eine Live-Resynthese übertragen. Häufig werden Segmente im Ausgangsmaterial oder sogar Grains für die Synthese manuell ausgewählt, um einen bestimmten Klang zu erzielen. Dies ist bei einer Granulierung eines Echtzeit-Signals aber nur eingeschränkt möglich, da sich das Signal und die Menge zur Verfügung stehender Grains fortlaufend ändert. Deshalb wird ein Verfahren für eine automtisierte Auswahl von Grains nach Merkmalen entwickelt. Konkret werden in der Arbeit folgende Fragestellungen behandelt:

- Wie lässt sich eine Live-Resynthese mit Wavesets in SuperCollider umsetzen?
- Welche Synthesemethoden sind für den Live-Einsatz mit einem Instrument sinnvoll?
- Welche Verfahren eignen sich um Wavesets auszuwählen?

## 1.1 Ausgangssituation: Das Projekt 3DMIN

> Im Projekt »Design, Development and Dissemination of New Musical Instruments« beschäftigen sich Wissenschaftler und Künstler verschiedenster Disziplinen mit Fragestellungen rund um historische und zeitgenössische elektronische Musikinstrumente. In enger Zusammenarbeit mit internationalen Künstlern, und in praktischen Lehrveranstaltungen mit Studierenden der beteiligten Hochschulen werden dazu Prototypen neuer Musikinstrumente entwickelt. Die dabei entstandenen Entwürfe werden hinsichtlich ihrer Vielseitigkeit und Expressivität in der künstlerischen Praxis evaluiert. [48]

Eines der dabei entwickelten Musikinstrumente ist PushPull [20] (siehe Abb. 1.2). Es besteht aus einem Blasebalg ähnlich einem Akkordeon und kombiniert mecha-

[2] `http://supercollider.github.io`

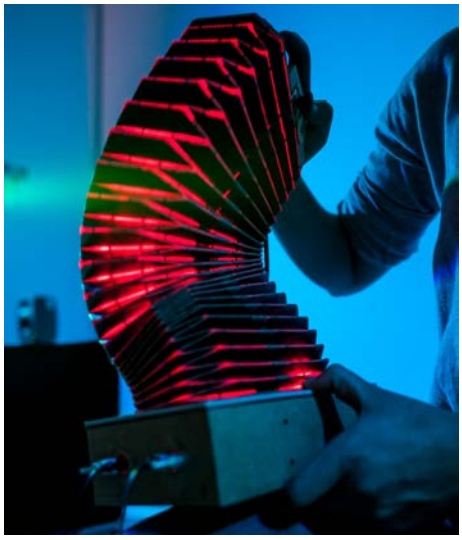

**Abbildung 1.2:** Das Musikinstrument PushPull [20]. Ein Blasebalg ähnlich einem Akkordeon erzeugt einen Luftstrom über zwei Mikrofone. Das daraus entstehende Signal dient zusammen mit den Daten von Beschleunigungssensoren als Grundlage für eine digitale Synthese.

nisch-analoge Töne mit den Daten von Beschleunigungssensoren in einer digitalen Synthese. Durch Bewegung des Handteils entsteht ein Luftstrom über zwei Mikrofone, dessen Signal als Basis für die Synthese dient. Die Synthese kann durch die Beschleunigungssensoren und vier Tasten beeinflusst werden.

Auch bei anderen elektronische Musikinstrumenten wird für die Signalerzeugung ein digitaler Syntheseprozess eingesetzt. Hier soll zukünftig auch eine Granularsynthese mit Wavesets eingesetzt werden.

## 1.2 Stand der Technik

In diesem Abschnitt wird ein Überblick über bereits vorhandene Systeme, die eine Waveset-Resynthese umsetzen, sowie Systeme mit einer konventionellen Live-Resynthese gegeben.

Wishart selbst setzte mit Wavesets einige Transformationen im Rahmen des *Composers Desktop Projekt* (CDP)[3] [2] um. Das Ausgangsmaterial sind hier Audiodateien und die Synthese erfolgt nicht in Echtzeit [62].

Eine weitere Implementierung Wisharts Methoden bietet die SuperCollider-Erweiterung *Wavesets* von Alberto de Campo [12, 60]. Das Ausgangsmaterial, aus dem die

[3] `http://www.composersdesktop.com`

Wavesets extrahiert werden muss für die Analyse als ganzes Stück im Speicher liegen. Die Wavesets nutzt de Campo nicht für Transformationen wie Wishart, sondern für unterschiedliche Syntheseverfahren. Jedes Waveset bietet die Metadaten Länge und Amplitude, auf dessen Basis auch eine Auswahl von Wavesets realisiert werden kann.

Eine weitere SuperCollider-Erweiterung, die mit Wavesets arbeitet, stellte Olaf Hochherz mit *SPList* 2008 vor [22]. Sie ermöglicht Transformationen und beliebige andere Umstrukturierungen eines Ausgangsmaterials, die Hochherz für seine Kompositionen nutzt. Auch hier haben Wavesets ähnliche Metadaten auf dessen Basis sie ausgewählt und sortiert werden können. Sowohl die Analyse als auch die Synthese arbeitet mit Audiodateien und findet nicht in Echtzeit statt.

Während keine dieser Implementierungen mit Wavesets eine Granulierung in Echtzeit unterstützt finden sich im Bereich der konventionellen Granularsynthese Beispiele für Systeme, die eine Resynthese in Echtzeit basierend auf einem Live-Eingangssignal bieten:

Die von Terry Lee [25, 26] vorgestellte SuperCollider Erweiterung „GranCloud“ bietet eine asynchrone Granularsynthese basierend auf der Analyse eines Live-Signals. Neben der reinen SuperCollider-Klasse bietet es eine grafische Benutzeroberfläche mit der sich die Synthese steuern lässt. Das Projekt ist zwar nicht mehr aktiv und Lees Website offline, die Implementierung ist jedoch noch in der SuperCollider-Mailingliste verfügbar [58].

Mit SuperCollider Version 3.2 wurden von Joshua Parmenter einige Module zur Granularsynthese hinzugefügt. Darunter die Klasse *GrainIn*, welche ein Live-Eingangssignal granuliert und eine einfache Synthese erlaubt [53]. Die Hüllkurve und Länge lässt sich frei bestimmen und die zeitliche Anordnung der Grains wird durch ein Triggersignal gesteuert.

Mit „GrainProc“[4] entwickelten Sanganeria und Werner 2013 eine App für Apples iPad, das Granularsynthese für Live-Auftritte in Kombination mit beliebigen Instrumenten nutzbar machen soll [37]. Als Synthesemethode kommt dabei *Tapped Delay Line Granular Synthesis* [6] zum Einsatz, bei der zufällig Grains aus einem Puffer ausgewählt werden. Der Puffer wird durch die Granulierung eines beliebigen Live-Eingabesignals fortlaufend befüllt. Über den Touchscreen lässt sich die Synthese mit einfachen Parametern steuern. Da bei Einsatz klassischer Instrumente die Hände

[4] `http://www.grainproc.com`

oft nicht frei sind ist die Bedienelemente so gestaltet, dass auch eine Bedienung mit den Füßen möglich ist. Die App wird immernoch weiterentwickelt und kann im App-Store erworben werden.

Timothy Opie entwickelte ein Instrument in Form eines Fisches, genannt „Poseidon“ zur Steuerung einer Granularsynthese [31] und erprobte dieses in Live-Konzerten [30]. Gesteuert werden Parameter einer synchronen oder asynchronen Synthese basierend auf der Granulierung eines beliebigen Live-Eingangssignals. Gesteuert werden können die Grainlänge, Hüllkurve, Dichte und Verteilung der Grains.

Es gibt einige Implementierungen einer Granularsynthese auf Basis des Waveset-Konzepts nach Wishart. Zum Teil bieten diese auch eine Synthese in Echtzeit, jedoch unterstützt keine der bekannten Systeme die Verarbeitung eines Live-Eingangssignals bei der Analyse, d.h. eine Granulierung in Echtzeit. Die Waveset-Synthese von de Campo kann zwar potentiell in Live-Anwendungen eingesetzt werden, die Kombination mit einem live gespielten Instrument als Eingangssignal für die Granulierung ist aber nicht möglich. In der konventionellen Granularsynthese finden sich einige Systeme, die sowohl die Analyse als auch die Synthese in Echtzeit unterstützen und damit den Live-Einsatz zusammen mit anderen Instrumenten ermöglichen.

## 1.3 Überblick über die Arbeit

Das Zweite Kapitel beschreibt die zum Verständnis der Arbeit notwendigen Grundlagen sowie die eingesetzten Methoden. In Kapitel drei werden die Anforderungen gesammelt aufgelistet und in einem abstrakten Produktmodell dargestellt. Im vierten Kapitel „Softwaredesign“ wird das Produktmodell zu einer konkreten Softwarearchitektur weiterentwickelt. Das fünfte Kapitel „Implementierung“ beschäftigt sich mit den bei der Programmierung getroffene Entscheidungen und Problemlösungen.

# 2 Grundlagen

In diesem Kapitel werden die zum Verständnis der Arbeit notwendigen Grundlagen sowie eingesetzte Methoden erläutert.

## 2.1 Granularsynthese

Granularsynthese ist ein Verfahren bei dem Klänge in sehr kleine Fragmente (Körner, engl. Grains) zerteilt werden, die dann neu angeordnet werden um neue Klänge zu bilden.

### 2.1.1 Geschichte und Theorie

Teilchentheorien für den Schall vergleichbar mit der Quantenphysik gibt es seit langer Zeit. Eine detaillierte Teilchentheorie für die Akustik schlug erstmals der niederländische Wissenschaftler Isaac Beeckmann im Jahr 1616 vor [14]. Seiner Ansicht nach entsteht bei schwingenden Objekten Schall, indem sie umgebene Luft in kleine runde Teilchen spalten, die dann einen Einschlag im Trommelfell erzeugen.

Einen anderen Weg ging Dennis Gabor [18], der 1947 versuchte mithilfe von Theorien der Quantenphysik die auditive Wahrnehmung des Menschen zu erklären [44, S. 463]. Er erhielt eine Unschärferelation für Schall und betrachtete das Zeitsignal (welches keinerlei Informationen zur Frequenz enthält) und die Fourier-Transformierte (die keine Informationen über die Zeit enthält) als zwei Extreme der teilchenbasierten Sicht auf akustische Signale. Dennis Gabor führte ein Quantum akustischer Information ein, welches das Maximum der erreichbaren Bestimmtheit darstellt und nahm an, dass Ton in elementare Partikel zerlegt werden kann. Diese sind Vibrationen mit stationären Frequenzen moduliert mit einer Wahrscheinlichkeitsfunktion, die letztendlich eine Hüllkurve in Form einer Gauß-Funktion ist [44, S. 464].

Gabors Theorie wurde von dem Komponisten Iannis Xenakis aufgegriffen [47] und musikalisch für einige seiner Arbeiten eingesetzt. Seine ersten Arbeiten mit Granularsynthese setzte er durch das Zerschneiden eines Magnetbands in kleine Teile um, die er neu anordnete und mit Klebeband zusammenfügte. Nachdem Curtis Roads ein von Xenakis geleitetes Seminar zu diesem Thema besucht hatte, begann er

mit diesen Ideen auf einem Großrechner zu experimentieren. Seine ersten Versuche waren sehr zeitaufwändig. Die Programmierung erfolgte mit Lochkarten und die Berechnung von nur einer Minute Monosignal war ein aufwendiges Prozedere, das Tage dauerte [35, S. 109]. Mit dem technischen Fortschritt und der Verfügbarkeit erschwinglicher PCs war es Roads ab 1988 möglich, unterschiedliche Programme zur Granularsynthese zu implementieren (unter anderem Programme *Synthulate* und *Granulateur* 1992 sowie *Cloud Generator* 1995) [35, S. 111]. Nachdem Barry Truax einen Artikel von Roads aus dem Jahr 1978 [33] las, begann er sich mit dem Thema zu beschäftigen und an einer Umsetzung der Granularsynthese in Echtzeit zu arbeiten, was ihm erstmals 1986 [42] gelang. Von da an begann Granularsynthese langsam für eine immer breitere Masse an Musikern und Klangkünstlern verfügbar zu werden.

### 2.1.2 Funktionsweise

Grains sind kurze Schallereignisse auf Mikroebene mit einer Länge nahe an der Wahrnehmungsgrenze des menschlichen Gehörs im Bereich einer tausendstel bis zehntel Sekunde (entspricht 1 bis 100 ms). Jedes Grain besteht aus einer Wellenform, die von einer Hüllkurve modelliert wird (siehe Abb. 2.1). Ein einzelnes Grain fungiert als Baustein für Klangereignisse. Es eignet sich gut für die Repräsentation musikalischer Klänge, da es neben der Zeitebene (Dauer, Hüllkurve) auch Informationen im Frequenzbereich (Tonhöhe der Wellenform und Spektrum) umfasst. [35, S. 87] Die Wellenform eines Grains kann synthetisch erzeugt oder aus Audioaufzeichnungen extrahiert werden. Die Granulierung von aufgezeichnetem Audiomaterial ist ein mächtiges Mittel zur Veränderung von Tönen durch die Zerteilung eines Signals in Grains, die dann eventuell verändert und neu angeordnet werden [35, S. 98]. Eine spezielle Methode zu Granulierung von Audiosamples sind die sogenannten Wavesets, auf die in Abschnitt 2.2 eingegangen wird.

Granularsynthese erfordert eine große Menge Daten zur Steuerung (z.B. 1000 Grains pro Sekunde mit jeweils 10 Parametern). Deshalb ist für den praktischen Einsatz eine Organisationseinheit auf übergeordneter Ebene erforderlich. Besonders für den Live-Einsatz ist wichtig, dass sich der Künstler auf abstrakter Ebene ausdrücken kann und die Details von einem automatisierten Prozess gefüllt werden. Die verschiedenen Granularsynthese-Techniken unterscheiden sich vor allem durch die

Art der übergeordneten Organisation und den Algorithmen hierfür. Hier gibt es nach Roads [35, S. 91-101] fünf Hauptformen:

**Matrizen und Screens auf Zeit-Frequenz-Ebene:** Durch Analyse (z.B. Gabor Transformation, Kurzzeit-Fourier-Transformation, Wavelet-Transformation) eines Signals kann dieses als zweidimensionale Matrix mit Frequenz- und Zeitachse dargestellt werden, was eine Vielzahl von Möglichkeiten zur Synthese bieten. Vergleichbar damit sind *Screens* nach Xenakis [47], bei denen die Grains auf Frequenz-Amplituden-Ebene angeordnet sind und die *Screens* als Momentaufnahme über die Zeit. Anstatt einer Analyse werden hier die *Screens* algorithmisch mit Grains gefüllt.

**Pitchsynchrone Granularsynthese:** Basierend auf einer Spektralanalyse wird eine Resynthese erzeugt bei der die Formanten erhalten bleiben. Die Zeit-Frequenz-Ebene wird in Zellen unterteilt von denen jede von einem Grain repräsentiert wird. Für die Zellen eines Zeitpunkts wird die Grundfrequenz ermittelt und für die Zellen eines Frequenzbandes die Koeffizienten eines Filters. Es wird eine FIR-Filterbank konstruiert, die für die Resynthese von einer Impulswelle mit der erkannten Grundfrequenz angeregt wird (mehr dazu in [13]).

**Synchrone und quasi-synchrone Synthese:** Bei synchroner Granularsynthese entstehen Klänge durch einen oder mehrerer Ströme von Grains. Jeder Strom besteht aus einer Folge von Grains angeordnet in Zeitabständen gleicher Länge. Sie eignet sich gut um metrische Rhythmen zu erzeugen, vor allem wenn die Zeitspanne zwischen den Grains groß und damit die Grain-Dichte niedrig (0,1 bis 20 Grains pro Sekunde) ist. Bei einer höheren Frequenz verschmelzen die Grains zu kontinuierlichen Tönen mit einer stark ausgeprägten Grundfrequenz, oft auch mit Seitenbändern. Bei der quasi-synchronen Granularsynthese werden die festen Intervalle durch eine zufällige Abweichung variiert ohne die Grain-Dichte zu verändern.

**Asynchrone Granularsynthese:** Hier werden die Grains nicht linear angeordnet sondern im Bereich einer vorgegebenen Zeit und Frequenz als „Wolken“ (engl. clouds) gestreut. Die Grains innerhalb einer Wolke werden ungleichmäßig verteilt, gesteuert durch einen stochastischen oder chaotischen Algorithmus.

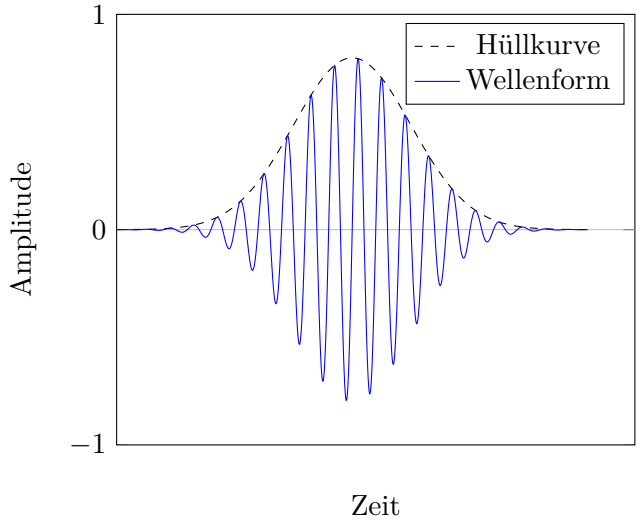


**Abbildung 2.1:** Beispiel eines Grains dargestellt im Zeitbereich. Die Wellenform wird von einer Hüllkurve modelliert.

Üblicherweise werden die Wolken durch eine Reihe von Parametern definiert. Etwa Startzeit und Dauer, Grain-Länge, Grain-Dichte, Frequenzbereich, Amplitude der Grains, Wellenform der Grains, räumliche Anordnung bei mehreren Kanälen.

**Physikalische oder Abstrakte Modelle:** Diese Methode basiert auf der mathematischen Beschreibung akustischer Tonerzeugung. Etwa die realer Musikinstrumente oder auch von virtuellen Welten aus abstrakten oder chaotischen Algorithmen. (Mehr dazu in [35, S. 97].)

## 2.2 Wavesets

Trevor Wishart stellte das Konzept der Wavesets 1994 in seinem Buch Audible Design [45, Anhang 2] vor und implementierte damit eine Reihe von Transformationen im Composers Desktop Projekt (CDP) [2]. Diese nutzte er auch in einigen seiner Werke (z.B. Tongues of Fire, 1994). Wishart betrachtete Wavesets vor allem als Elemente für Transformationen von Audioaufnahmen. Allgemeiner betrachtet kann mit ihnen ein beliebiges Eingangssignal in eine Sammlung von Fragmenten verwandelt werden. Diese können dann sehr unterschiedlich genutzt werden. [12]

Wisharts Definition nach ist ein *Waveset* ein Teil eines Audiosignals zwischen

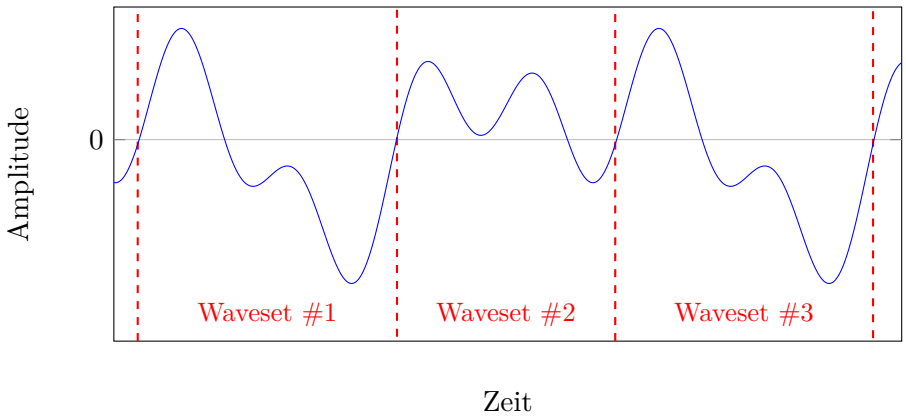


**Abbildung 2.2:** Beispiel der Zerteilung eines Signals in Wavesets. Das Signal wird an den negativ-positiv-Nulldurchgängen geteilt (rote gestrichelte Linie).

einem negativ-positiv Nulldurchgang und dem nächsten (siehe Abb. 2.2). Durch die Trennung an den Nullstellen ist eine Synthese mit Wavesets zu einem kontinuierlichen Signal im Gegensatz zur konventionellen Granularsynthese auch ohne Hüllkurve möglich.

Für rein sinusförmige Signale und Signale mit einer stark ausgeprägten Grundfrequenz entspricht ein Waveset genau einer Periode. Bei harmonischen Tönen mit starken Obertönen kann die Wellenform mehr als zwei Nulldurchgänge in einer Periode haben (siehe Abb. 2.2). In diesem Fall ist ein Waveset kürzer als eine Periode. Bei komplexen Signalen (z.B. Sprache) mit Rauschanteilen führt dies zu unterschiedlichen, technisch willkürlichen aber musikalisch potentiell interessanten Artefakten [62]. Wishart definierte eine Reihe von Transformationen, die er in seinen Werken einsetzt, um Klänge zu bearbeiten. Zum Beispiel ersetzt er einzelne Wavesets durch andere Wellenformen (z.B. Sinus oder andere Wavesets), tauscht die Reihenfolge von Wavesets, verändert die Abspielgeschwindigkeit, spielt sie rückwärts ab oder modelliert sie mit Hüllkurven. Eine Auswahl an Transformationen mit Beschreibung ist in Tabelle 2.1 aufgelistet.

**Tabelle 2.1:** Eine Auswahl an Waveset-Transformationen nach Wishart [44, S. 493]

| | |
|---|---|
| Transposition | e.g., take every second waveset, play back at half speed |
| Reversal | play every waveset or group time reversed |
| Inversion | Turn every half waveset inside out |
| Omission | play silence for every m out of n wavesets (or randomly by percentage) |
| Shuffling | switch every 2 adjacent wavesets or groups |
| Distortion | multiply waveform by a power factor (i.e., exponentiate with constant peak) |
| Substitution | replace waveset with any other waveform (e.g., sine, square, other waveset) |
| Harmonic distortion | add double-speed and triple-speed wavesets to every waveset; weight and sum them |
| Averaging | scale adjacent wavesets to average length and average their waveforms |
| Enveloping | impose an amplitude envelope on a waveset or group |
| Waveset transfer | combine waveset timing from 1 source with waveset forms from |
| Interleaving | take alternating wavesets or groups from 2 sources |
| Time stretching | repeat every waveset or group n times (creates "pitch beads") |
| Interpolated time | interpolate between waveforms and durations of adjacent waveset |
| Stretching | over n crossfading repetitions |
| Time shrinking | keep every nth waveset or group |

## 2.3 SuperCollider

SuperCollider (SC)[1] ist eine Software und Programmiersprache für Echtzeit-Klangsynthese und algorithmische Komposition. Die Entwicklung wurde 1996 von James MacCartney [28] begonnen und wird seit 2002 als Open Source unter der GNU GPL fortgesetzt. SuperCollider wird von Kunstschaffenden, KomponistInnen und WissenschaftlerInnen in den Bereichen Klang, Musik und Medienkunst eingesetzt. Das Standardwerk zur Software ist *The SuperCollider Book* [44] aus dem Jahr 2011.

SuperCollider besteht aus mehreren Komponenten: Der Programmiersprache, einer Entwicklungsumgebung, dem Syntheseserver *SCSynth* und dem Interpreter *SCLang*. Letztere beiden kommunizieren über das Netzwerkprotokoll *Open Sound Control* (OSC) [46] miteinander (siehe Abb. 2.3). Im Server erfolgt die Signalerzeugung und Verarbeitung mit als Graph angeordneten Synthesebausteinen, die als *Unit Generators* (kurz UGens) bezeichnet werden. Konfiguriert und gesteuert werden diese vom Client *SCLang* mit der SC-Programmiersprache. Durch das offene OSC-Netzwerkprotokoll können aber auch andere Programmiersprachen und Anwendungen eingesetzt werden, die auch über das Netzwerk verteilt sein können. Die Entwicklungsumgebung vereint die beiden Komponenten mit einer Benutzeroberfläche zum Starten von Server und Interpreter sowie einen Texteditor zum Schreiben und Ausführen von SC-Programmiercode.

### 2.3.1 Die Programmiersprache

Die SC-Programmiersprache [29] ist eine Skriptsprache mit einer objektorientierten Struktur ähnlich wie Smalltalk und einer Syntax ähnlich zu C. Sie ist eine funktionale Programmiersprache mit dynamischer Typisierung und einer echtzeitfähigen Garbage Collection. Es wird versucht den Anforderungen der Echtzeitsignalverarbeitung gerecht zu werden bei gleichzeitiger Flexibilität und Einfachheit von abstrakten Skriptsprachen. Durch speziell für die Computermusik nützlichen Abstraktionen kann sie als domänenspezifische Sprache eingeordnet werden.

Wie in funktionalen Programmiersprachen üblich werden Funktionen als *first-class-Objekte* implementiert, d.h. sie können Variablen zugewiesen und an andere Funktionen übergeben sowie zurückgegeben werden. Funktionen und Methoden

[1] `http://supercollider.github.io`

unterstützen Standardparameter und variabel lange Argumentenlisten. Closures[2] werden lexikalisch abgegrenzt, während der Sichtbarkeitsbereich von Variablen (engl. *Scope*) lexikalisch und dynamisch ist. Weitere für funktionale Programmiersprachen typische und von SC unterstützte Merkmale sind Currying [38], Tail-Call-Optimierung [1], Koroutinen [24, S. 229] und Mechanismen für eine einfache Verarbeitung von Listen (engl. *List Comprehension* [21]).

Laut McCartney sind grundsätzlich auch andere abstrakte Programmiersprachen geeignet um Computermusik zu beschreiben. Jedoch fehlt häufig eine echtzeitfägige Garbage-Collection, sowie eine Möglichkeit zur einfachen Beschreibung von Sequenzen wie die Pattern und Streams [29].

### 2.3.2 Der Syntheseserver

Der Syntheseserver *SCSynth* bietet die Infrastruktur für die von den UGens ausgeführten Signalverarbeitungsprozesse. Er empfängt eingehende OSC-Nachrichten, die interpretiert werden und greift auf die benötigten Ressourcen des Betriebssystems wie Audiotreiber, Dateisystem und Speicherverwaltung zurück. Lokale Instanzen des Servers können aus *SCLang* heraus oder in der IDE gestartet werden. SCSynth wird in mehreren Threads mit unterschiedlichen Aufgaben ausgeführt [52]:

- *Main Thread:* initialisiert die SuperCollider-Umgebung mit Audiotreiber und Com-Ports
- *Com Port Threads:* warten auf einkommende OSC-Nachrichten
- *Audio Thread:* gestartet vom Audiotreiber, führt den Audio-Callback aus
- *Audio Engine NRT:* audiobezogene aber nicht echtzeitkritische Aufgaben
- *Disk I/O Thread:* Ausführung von Dateisystemoperationen

Für Erweiterungen bietet der Syntheseserver über die *InterfaceTable* eine abstrakte Schnittstelle. Diese Datenstruktur bietet über Funktionszeiger unter anderem die Registrierung neuer UGens und eine echtzeitfähige Speicherverwaltung an. Der Zugriff auf die Schnittstelle soll durch Präprozessormakros vereinfacht werden, welche eine globale Variable *InterfaceTable* als globale Variable vorraussetzen. Beides gilt in

[2] Als Closure oder Funktionsabschluss wird eine anonyme Funktion mit zugehörigem Erstellungskontext bezeichnet, auf den diese Zugriff erhält.

C++ als nicht mehr zeitgemäß (siehe [19, S. 370]) und ist vermutlich auf das Alter der Software zurückzuführen. Im Sourcecode ist wenig Dokumentation in Form von Kommentaren zu finden, aber ein grober Überblick über die Komponenten ist auf der Website zu finden [52]. Ein Nachteil der SCSynth Implementierung ist in Zeiten der Mehrkernprozessoren die Ausführung der Signalberechnung in nur einem Thread. Als Alternative gibt es die modernere Implementierung *Supernova* von Tim Blechmann [7], die durch Parallelisierung mehrere CPU-Kerne nutzen kann.

### 2.3.3 UGens

Unit Generators sind die Module, aus denen Signalverarbeitungsketten in SuperCollider gebildet werden. Mit ihnen können komplexe, aber dennoch effiziente Audionetzwerke erstellt werden, die als UGen Graphen [34] bezeichnet werden. Sie sind in C++ implementiert und werden beim Starten des Servers aus einer Bibliothek geladen. Eine große Auswahl an UGens bringt SuperCollider bereits mit, hunderte weitere stehen über den Paketmanager *Quarks* von der Entwicklergemeinde zur Verfügung. Eigene UGens können über eine einfache Programmierschnittstelle implementiert werden. Um auf sie von der SC-Programmiersprache aus zurückzugreifen sind sie dort als Klassen repräsentiert. Mit der Klasse *SynthDef* wird beschrieben, welche UGens verwendet und wie diese kombiniert werden. Als Eingangswerte können skalare Werte, Audiosignale und Steuersignale übergeben werden bzw. im Allgemeinen Gleitkommazahlen mit Werten im Audiotakt oder im Steuertakt (mehr zu den unterschiedlichen Taktraten später). Die *Synthdefs* werden per OSC zum Server geschickt auf dem Instanzen der serverseitigen Repräsentation des UGens erstellt und zu einem Graph angeordnet werden. Nur auf dem Syntheseserver werden Signale von den UGens verarbeitet und produziert. Deshalb ist das Ausgabesignal eines UGens auch grundsätzlich nicht in der Programmiersprache verfügbar, sofern dieser nicht explizit vom Server zum Client übertragen wird.

Die Aufgabe eines UGens ist üblicherweise, ein Ausgabesignal zu generieren. Was aber im UGen zur Signalerzeugung konkret passiert kann sehr unterschiedlich sein. Je nach Einsatzzweck wird die Berechnung in verschiedenen Taktraten ausgeführt [44, S. 703]:

- *Audio-Rate* entspricht der Abtastrate für Audiodaten, die für den Server konfiguriert wurde. Die Berechnung erfolgt hier in Blöcken, standardmäßig mit einer

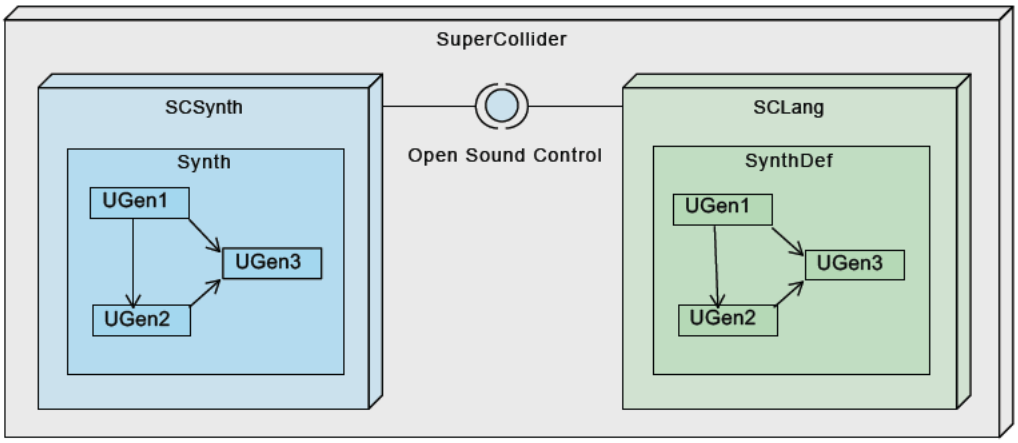


**Abbildung 2.3:** Die Client-Server-Architektur von Supercollider. Auf dem Server *SCSynth* (blau) wird die Signalverarbeitung durch als Graph angeordnete Synthesebausteine ausgeführt. Der Server wird vom Client *SCLang* (grün) über das Protokoll *Open Sound Control* konfiguriert und gesteuert.

Länge von 64 Samples.

- *Control-Rate* leitet sich aus der Blockgröße für Audiodaten ab. Pro Block wird ein Control-Rate Wert berechnet. Für Steueraufgaben, bei denen eine geringere zeitliche Auflösung ausreicht, kann so Rechenleistung gespart werden.
- *Demand-Rate* stellt einen Sonderfall dar. Bei Demand-Rate UGens kann die Berechnung nach Bedarf zu bestimmten Zeitpunkten mit einem Triggersignal ausgelöst werden.

Zum Datenaustausch zwischen den UGens stehen sogenannte *Buffer* zur Verfügung. Diese stellen einen auf dem Server reservierten Speicherbereich dar, etwa zum Schreiben und Lesen von Audiosignalen. Ihre Allokation wird vom Client aus angestoßen, anschließend werden sie anhand einer eindeutigen Nummer identifiziert. Für den Zugriff in einem UGen wird diese Nummer als Parameter übergeben. SC unterstützt eine beliebige Anzahl von Ein- und Ausgabekanälen, für Systeme mit sehr vielen Kanälen ist SC bestens geeignet [61].

### 2.3.4 SC-Erweiterungen und deren Entwicklung

SuperCollider erlaubt die Erweiterung der Klassenbibliothek, Dokumentation und UGens. Die Erweiterungen werden an einem plattformabhängigen Pfad abgelegt

```
MyExtension
├── classes
│   ├── myClass.sc
│   └── myUGens.sc
├── plugins
│   └── myUGenPlugin.scx
├── HelpSource
│   ├── MyClass.schelp
│   ├── MyUGen1.schelp
│   └── MyUGen2.schelp
└── Guides
    └── MyExtensionGuide.schelp
```

**Abbildung 2.4:** Die Ordnerstruktur für SuperCollider-Erweiterungen (nach [55]). Sie umfasst SC-Klassendefinitionen (*classes*), C++-Module für UGens (*plugins*), Hilfedateien (*HelpSource*) und Anleitungen (*Guides*).

und beim Start von Entwicklungsumgebung, Interpreter oder Syntheseserver automatisch geladen. Jede Erweiterung besteht aus einem Ordner in dem die einzelnen Komponenten in einer vorgegebenen Struktur abgelegt werden (siehe Abb. 2.4).

UGens bestehen aus einem C++ Modul auf Serverseite *SCSynth* (.scx-Datei) und einer zugehörigen SuperCollider-Klasse auf Interpreterseite (.sc-Datei). In den Listings 2.1 und 2.2 ist ein komplettes Beispiel für die Implementierung eines einfachen UGens zu sehen. Das UGen *SimpleSine* gibt ein einfaches Sinussignal aus, dessen Frequenz und Phase über zwei Parameter bestimmt werden kann. Die Schnittstelle mit den Parametern auf SCLang-Seite wird durch die Klassendefinition festgelegt (siehe Listing 2.1). Entsprechend der üblichen Namenskonvention für UGens hat es eine Methode *ar*, die das UGen mit Audiotaktrate instanziiert. In der Methode macht dies die Funktion *multiNew*, welche automatisch eine Instanz pro Kanal erzeugt. Die vorgegebene Schnittstelle für die C++ Implementierung eines UGens ist eine klassenähnliche Struktur aus unterschiedlichen Funktionen und einer Datenstruktur (siehe Listing 2.2). In der Datenstruktur (*struct*) können Attribute bzw. zur Laufzeit des UGens relevante Daten deklariert werden. In den Funktionen werden jeweils Abläufe zum Erzeugen (Konstruktor), Löschen (Destruktor) sowie die Berechnung des

**Listing 2.1:** Minimalbeispiel eines UGens, SuperCollider-Klasse. Es wird eine UGen-Klasse *SimpleSine* definiert, die ein einfaches Sinussignal mit den Parametern Frequenz und Phase erzeugt.

```
SimpleSine : UGen {
    *ar { arg freq=440, phase=0;
        ^this.multiNew('audio', freq, phase)
    }
}
```

Signals implementiert. Letztere wird regelmäßig im Takt der Control-Rate aufgerufen und berechnet im Falle eines Audio-Rate-UGens einen Block von üblicherweise 64 Samples. Für Control-Rate-UGens entsprechend nur einen Wert (in der Funktion wird *inNumSamples=1* übergeben). Auf die Ein- und Ausgabesignale eines UGens kann nicht über auf SCLang-Seite definierten Namen zugegriffen werden, sondern nur über den Index innerhalb der Parameterliste. Zum Beispiel erhält man mit den Makros *IN(k)* bzw. *OUT(k)* einen Zeiger auf das Feld des $k$ten Ein- bzw. Ausgangs.

Die im C++ Modul implementierten UGens müssen beim Starten des Servers registriert werden, damit diese zur Verfügung stehen. Auch hierfür gibt es entsprechende Makros, die das UGen im Syntheseserver registrieren.

Für die Erstellung eines UGens schlägt Stowell [40] den folgenden Arbeitsablauf vor:

1. Sinnvolle Abgrenzung der Funktionalität für ein UGen
2. Verfassen einer SC-Hilfedatei als Spezifikation
3. Implementierung der SC-Klasse
4. Implementierung des C++ Moduls

Ein kurzer Leitfaden zur Implementierung von UGens und SC-Klassen ist auch in der SuperCollider-Dokumentation zu finden ([57] und [56]).

Der von Stowell beschriebene Buildprozess mit der Entwicklungsumgebung Xcode ist veraltet, stattdessen sollte das neuere Vorgehen auf cmake-Basis eingesetzt werden. Für den Einstieg kann auf das Paket SC3-Plugins [51] als Beispiel für einen funktionierenden cmake-Buildprozesses zurückgegriffen werden.

**Listing 2.2:** Minimalbeispiel eines UGens, C++ Implementierung. Das C++ Modul besteht aus einem klassenähnlichem Konstrukt aus Struktur für die Datenelemente und Funktionen für Erzeugung, Signalberechnung und Zerstörung des UGens.

```
#include "SC_PlugIn.h"
#include <math.h>

/** Definition of the UGens data structure */
struct SimpleSine : public Unit {
    float phase; // declare member variables
};

/** Function Declarations */
void SimpleSine_Ctor(SimpleSine *unit);
void SimpleSine_next(SimpleSine *unit, int inNumSamples);
void SimpleSine_Dtor(SimpleSine *unit);

/** Constructor: Initialize UGen */
void SimpleSine_Ctor(SimpleSine *unit)
{
    SETCALC(SimpleSine_next); // 1. set the calculation function.
    unit->phase = 0; // 2. set member variables
    SimpleSine_next(unit, 1); // 3. calculate one sample of output.
}

/** Process Signal */
void SimpleSine_next(SimpleSine *unit, int inNumSamples)
{
    // 1. get input and output buffers
    float *inFreq = IN(0);
    float *inPhase = IN(1);
    float *out = OUT(0);

    // 2. calculate samples
    for ( int i=0; i<inNumSamples; ++i){
        out[i] = sin(unit->phase + inPhase[i]);
        unit->phase += 2.0 * M_PI / 44100 * inFreq[i];
    }
}

/** Destructor: Destroy UGen */
void SimpleSine_Dtor(SimpleSine *unit){
    // i.e. free allocated memory
}

static InterfaceTable *ft; // Pointer to server interface

/** Register UGen on Load */
PluginLoad(ExampleUGens)
{
    // this entry point is called by the host when the plug-in is loaded
    // InterfaceTable *inTable ist implicitly given as argument, store it:
    ft = inTable;
    DefineSimpleUnit(SimpleSine);
}
```

## 2.4 Vorgehen zur Softwareentwicklung

In der heutigen Softwaretechnik ist eine große Vielfalt von Vorgehensmodellen zur Softwareentwicklung vertreten (einen Überblick bieten [39, 3]). Neben traditionellen, linearen Vorgehensweisen wie das Wasserfall- oder V-Modell erfreuen sich seit den 90er Jahren Agile Methoden wie Scrum und Extreme Programming zunehmender Beliebtheit. Diese gehen meist iterativ vor und können flexibler auf Änderungen der Anforderungen reagieren. Eine universelle Lösung für alle Projekte sollte in keinem der Vorgehensmodelle gesucht werden, vielmehr müssen die passenden Methoden für jedes Projekt individuell ermittelt werden [3, Kap. 24].

In diesem Projekt standen zu Beginn noch nicht alle Anforderungen fest oder zumindest war nur eine sehr vage Vorstellung vorhanden. Gleichzeitig sollte eine umfangreiche Dokumentation als schriftliche Arbeit erstellt werden. Das Team ist mit zwei beteiligten Personen, davon ein Entwickler sehr klein. Um mit den veränderlichen Anforderungen umgehen zu können wurde ein iteratives Vorgehen gewählt. Dieses wurde mit den Aktivitäten Anforderungserfassung, Anforderungsanalyse, Entwurf, Implementierung aus den traditionellen Vorgehensweisen kombiniert, die eine ausführliche Dokumentation bieten.

Zunächst erfolgte in einem Gespräch mit dem Anwender die Erfassung der Anforderungen für den nächsten Schritt. Diese wurden mit der objektorientierten Analyse modelliert und anschließend mit objektorientiertem Design zu einem Softwareentwurf weiterentwickelt und implementiert. Die Ergebnisse wurden nach jedem Schritt überprüft und diskutiert. Dabei wurden die Anforderungen für die nächste Iteration festgelegt.

### 2.4.1 Anforderungserfassung

Bei der Anforderungserfassung werden Eigenschaften für das System festgelegt, mit denen sich die Visionen und Ziele realisieren lassen. Dazu gehören nach Balzert [4]:

- Rahmenbedingungen: organisatorische Rahmenbedingungen (Anwendungsbereich, Zielgruppe, Betriebsbedingungen) oder technische Rahmenbedingungen (technische Produktumgebung, Anforderungen an die Entwicklungsumgebung)
- Umgebung: Der Kontext, in die das System eingebettet ist wird definiert. Es kann Abhängigkeiten zu materieller (z.B. Sensoren, Personen, technische

Systeme) und immaterieller Umgebung (Schnittstellen zu anderen Systemen) geben.

- Funktionale Anforderungen: Eine funktionale Anforderung legt eine vom Softwaresystem, einer seiner Komponenten bereitzustellende Funktion oder bereitzustellenden Service fest.
- Nichtfunktionale Anforderungen sind Qualitätsanforderungen, die keine Funktion beschreiben. Zum Beispiel Genauigkeit, Verfügbarkeit, Nebenläufigkeit, Zuverlässigkeit, Sicherheit.

Zur Ermittlung der Anforderungen gibt es unterschiedliche Techniken (siehe [36, 4]). In dieser Arbeit wurde die Umgebung des Systems und die Bedürfnisse des Anwenders im direkten Gespräch ermittelt. Sie wurden anschließend ausgewertet und schrittweise strukturiert festgehalten.

Es sollten nur Anforderungen aufgenommen werden, die gewissen Qualitätskriterien entsprechen, gegebenenfalls sind sie zu überarbeiten. In der Fachliteratur [32, 4, 36] und dem IEEE Standard zur Spezifikation von Software [23] finden sich weitgehend ähnliche Qualitätskriterien für Anforderungen:

- Notwendigkeit: Die Anforderung reflektiert die Bedürfnisse der Auftraggeber oder der Benutzer.
- Frei in der Umsetzung: Die Anforderung legt keine unnötigen Einschränkungen für die Art der Realisierung fest.
- Vollständigkeit: Die Anforderung beschreibt die geforderte Funktionalität vollständig.
- Eindeutigkeit: Die Anforderung wird von allen Beteiligten gleich interpretiert.
- Konsistenz: Die Anforderung muss in sich widerspruchsfrei sein.
- Umsetzbar: Die Anforderung ist technisch realisierbar mit den zur Verfügung stehenden Mitteln und passt in die Rahmenbedingungen.
- Überprüfbar: Die Anforderung ist so beschrieben, dass ihre Umsetzung überprüfbar ist.

- Verfolgbar: Die Anforderung lässt sich eindeutig identifizieren, etwa durch eine Nummer.
- Atomar: Eine Anforderung ist atomar, wenn sie einen isolierten Sachverhalt beschreibt.

### 2.4.2 Objektorientierte Analyse

Die Anforderungen wurden in jedem Iterationsschritt mit einer objektorientierten Analyse (OOA) [4, S. 548] geprüft und strukturiert. Es werden unterschiedliche Modelle erstellt, die unterschiedliche Perspektiven auf das System bieten und als Ganzes ein abstraktes Produktmodell ergeben. Die Modelle werden als grafische Diagramme unter Benutzung der *Unified Modeling Language (UML)* dargestellt. Wichtig ist, dass zugunsten der Übersichtlichkeit unwichtige Details weggelassen werden und durch Darstellung der zum Verständnis wichtigen Eigenschaften eine schnell erfassbare Abstraktion entsteht.

Unterschieden wird dabei zwischen dem statischen und dem dynamischen Teilmodell. Das statische Teilmodell beschreibt die stabile Struktur des Systems, welche sich über die Zeit nicht ändert. In diesem Fall der Aufteilung von Produktfunktionen und Daten in Klassen bzw. UGens und deren Anordnung bzw. Verhältnis zueinander. Entscheidend beeinflusst wird dieses durch die von SuperCollider vorgegebene Struktur der Synthesebausteine „Unit Generators“ mit Ein- und Ausgangssignalen. Das dynamische Teilmodell beschreibt das Verhalten und die Veränderungen des Systems während der Laufzeit. Der Ablauf von Operationen und Interaktionen zwischen Klassen bzw. UGens werden mittels Zustandsautomaten und Szenarien beschrieben. Bei bekannten Problemstellungen sollten, wo sinnvoll, standardisierte Lösungen in Form von OOA-Mustern [4, S. 550] zum Einsatz kommen.

### 2.4.3 Objektorientiertes Design

Mit dem objektorientierten Design (OOD) wird das erstellte allgemeine Modell weiterentwickelt zu einer konkreten Softwarearchitektur. Der Softwareentwurf enthält Informationen über die Details der Umsetzung und dient direkt als Vorlage für die Implementierung. Auf den Entwurfsprozess wirken eine Vielzahl von Einflussfaktoren, die sich oft gegenseitig beeinflussen und nicht selten Konflikte darstellen, für die

ein Kompromiss gefunden werden muss. Balzert nennt folgende Einflussfaktoren [5, Kapitel 10]:

- Funktionale Anforderungen
- Nichtfunktionale Anforderungen
- Anwendungsart
- Verteilungsart
- Kontext des Anwendungssystems
- Art der softwaretechnischen Infrastruktur

Beim objektorientierten Design wird zunächst die globale Architektur entworfen. Es erfolgt eine Unterteilung in Subsystemen und Komponenten, die zu einer logischen Architektur angeordnet werden (Grobentwurf). Anschließend werden die einzelnen Subsysteme und Komponenten im Detail entworfen (Feinentwurf). Wo sinnvoll, sollen in beiden Phasen etablierte Lösungsansätze für bekannte Problemstellungen eingesetzt werden. Diese werden als Architektur- bzw. Entwurfsmuster bezeichnet.

### 2.4.4 Implementierung

Bei der Implementierung werden die im Entwurf beschriebenen Konzepte mit den Mitteln der gewählten Programmiersprache umgesetzt. Hierzu gehört die Konzeption von Datenstrukturen und Algorithmen, die Strukturierung des Programms sowie die Dokumentation von Problemlösungen und Implementierungsentscheidungen durch Verbalisierung und Kommentare.

In der Literatur finden sich einige Prinzipien, die dabei eingehalten werden sollten [5, S.495]:

- Prinzip der Verbalisierung: aussagekräftige Namensgebung, Kommentare und selbstdokumentierender Programmierstil
- Prinzip der problemadäquaten Datentypen: Daten und Kontrollstrukturen des Problems sollen sich in der Implementierung möglichst unverfälscht wiederspiegeln

- Prinzip der integrierten Dokumentation: Kein externes Dokument für die Dokumentation, Beschreibung des Programms im Vorfeld und sinnvolle Kommentare im Sourcecode
- Prinzip des defensiven Programmierens: Potentielle Fehlerquellen möglichst konstruktiv vermeiden; Fehlersituationen abfangen (try/catch), Assertions, else-Zweig kommentieren

Zwischen jedem Implementierungsschritt sollte überlegt werden, ob Verbesserungen in der Programmstruktur möglich sind. Hierfür gibt es Kriterien für schlechten Programmcode, sogenannte Antipattern [11] und *Code-Smells* [16, S. 63], die eine Überarbeitung nahelegen. Beispiele hierfür sind duplizierter Code, lange Methoden, große Klassen, lange Parameterlisten und Switch-Anweisungen.

# 3 Anforderungen

Die schrittweise ermittelten Anforderungen sowie das Umfeld des Systems für die Waveset-Resynthese in Echtzeit werden in diesem Kapitel gesammelt dargestellt (mehr zum Vorgehen in Abschnitt 2.4). Zunächst wird definiert, wie das System eingesetzt werden soll sowie ein Überblick über das Produkt mit Kontext gegeben. Anschließend werden die funktionalen sowie nichtfunktionalen Anforderungen zusammengefasst. Diese werden anschließend in einem abstrakten Produktmodell dargestellt (fachliches Lösung), ohne technische Details der späteren Umsetzung.

## 3.1 Produkteinsatz

KünstlerInnen soll es ermöglicht werden, das Audiosignal eines Musikinstruments live mithilfe von Wavesets kreativ zu bearbeiten bzw. basierend auf dem Ausgangssignal neue Klänge zu schaffen. Sie interagieren dabei mit dem Instrument welches ein Audiosignal liefert sowie mit Bedienelementen (Controller), mithilfe derer die Granularsynthese gesteuert wird. Dabei sollen drei Szenarien berücksichtigt werden (siehe Abb. 3.1).

- **Szenario A:** Die Granularsynthese, welche das Signal eines Musikinstrumentes verarbeitet, wird als eigenständiges Instrument betrachtet.
- **Szenario B:** Das Instrument bildet mit dem Controller eine Einheit und wird inklusive der Waveset Resynthese als neues Instrument betrachtet.
- **Szenario C:** Die Resynthese wird als Filter bzw. Effekt zur Bearbeitung des Signals betrachtet.

Aus technischer Sicht unterscheiden sich die Szenarien nicht. In allen Fällen werden das Audiosignal eines Musikinstruments sowie die Steuerbefehle an einen Computer mit SuperCollider geleitet, auf dem eine Signalverarbeitung mittels Wavesets stattfindet.

Diese Arbeit konzentriert sich auf die Entwicklung einer SuperCollider-Erweiterung für die Waveset-Resynthese. Das Musikinstrument, die Bedienelemente oder eine Benutzerschnittstelle sind nicht Teil der Arbeit.

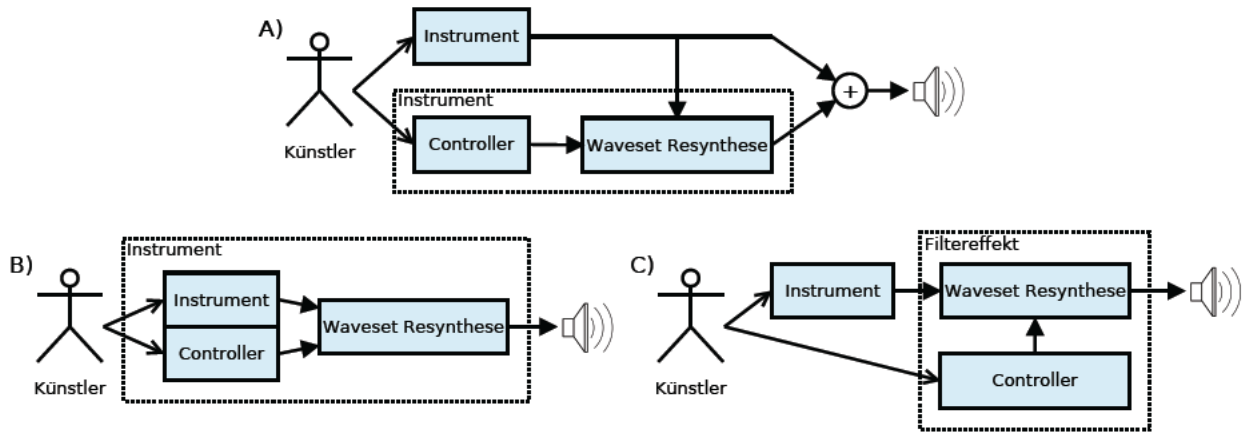


**Abbildung 3.1:** Drei Anwendungsszenarien der Granularsynthese mit Wavesets. Die Resynthese kann als eigenständiges Instrument, als neues Instrument zusammen mit einem Musikinstrument oder als Filtereffekt zur Bearbeitung eines Live-Signals betrachtet werden.

## 3.2 Produktübersicht

Die Waveset-Resynthese wird als modulare SuperCollider-Erweiterung mit den Hauptfunktionen Waveset-Analyse, Auswahl und Synthese umgesetzt (siehe Abb. 3.2). Ausgehend von einem Eingangssignal werden durch die Analyse Wavesets extrahiert und gespeichert. Aus der entstehenden Sammlung werden dann Wavesets ausgewählt, die mittels einer Synthese zu einem neuen Signal zusammengesetzt werden. Die umgebenden Komponenten können je nach Anwendung sehr unterschiedliche Anforderungen haben und müssen daher für jedes Setup individuell gestaltet werden. Das gilt für die Schnittstelle zu einem externen Controller ebenso wie eine eventuell notwendige Vorverarbeitung des Eingangssignals.

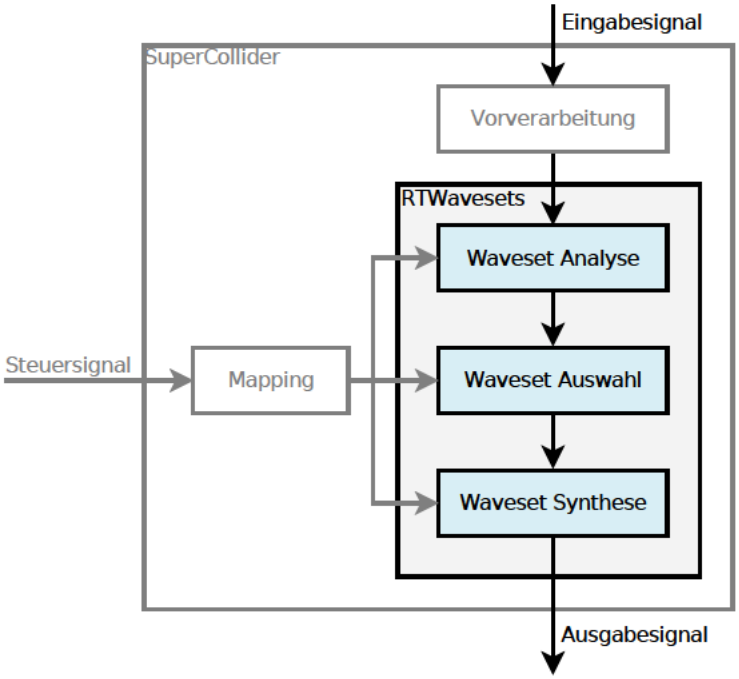


**Abbildung 3.2:** Überblick über die Waveset-Resynthese in SuperCollider mit Kontext. Der Fokus der Arbeit liegt auf der SC-Erweiterung *RTWavesets* während die umgebenden Komponenten für jede Anwendung individuell gestaltet werden müssen.

## 3.3 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben Funktionen und das dabei gewünschte Verhalten und die gespeicherten Daten des Systems für die Waveset-Resynthese (siehe Abschnitt 2.4.1). Damit sie im weiteren Verlauf referenzierbar sind, werden sie mit eindeutigen Identifikatoren versehen.

- **F1:** Waveset-Analyse eines Live-Signals

  Ein beliebiges Mono-Eingabesignal wird in Echtzeit analysiert um Wavesets zu extrahieren.

  - **F1.1:** Das Signal wird auf Nulldurchgänge hin untersucht und Wavesets festgelegt.
  - **F1.2:** Die Wavesets werden in einer zentralen Einheit gespeichert.
  - **F1.3:** Die Analyse kann angehalten und wieder fortgesetzt werden.
  - **F1.4:** Um zu kurze, nicht für die Synthese geeignete Wavesets zu vermeiden, soll eine Mindestlänge festgelegt werden können. Wird diese unterschritten,

werden so viele kürzere Wavesets zu einem zusammengefasst, bis diese erreicht ist.

- **F2:** Synthese basierend auf Wavesets

  Die durch eine Analyse extrahierten und im Speicher liegenden Wavesets werden durch unterschiedliche Syntheseverfahren zu neuen Klängen zusammengesetzt. Die Wavesets für die Synthese werden manuell ausgewählt oder durch einen automatisierten Prozess bestimmt (siehe F4). Die Wavesets sollen in der Synthese mit unterschiedlichen Methoden weitgehend frei angeordnet werden können:

  - **F2.1:** Ereignis-basierte Synthese

    Die Wavesets sollen sich im Zeitbereich frei gesteuert durch Ereignisse anordnen lassen. Ausgelöst durch ein Triggersignal werden angegebene Wavesets bzw. Gruppen (siehe F2.3) wiedergegeben. Wird eine neues Wiedergabeereignis ausgelöst, während vorhergehende Wavesets noch andauern, werden diese parallel wiedergegeben und summiert. Das Waveset bzw. die Gruppe werden manuell oder durch einen automatisierten Prozess ausgewählt (siehe F4).

    Damit können abhängig vom Trigger sehr unterschiedliche Waveset-Anordnungen geschaffen werden. Auch Anordnungen vergleichbar mit denen der klassischen synchronen oder asynchronen Granularsynthese (siehe Abschnitt 2.1.2) mit metrischen Rhythmen, kontinuierlichen Tönen oder Wolken. Entscheidender Unterschied ist jedoch die nicht festgelegte Grain-Dauer sowie der Verzicht auf die Modellierung mit einer Hüllkurve.

  - **F2.2:** Kontinuierliche Synthese

    Die Wavesets sollen lückenlos aneinander gereiht angeordnet werden. Der zeitliche Abstand wird also durch die Länge der Wavesets (oder Gruppen) vorgegeben, wobei die Freiheit in der Auswahl und Reihenfolge der Wavesets besteht. Die Auswahl erfolgt auch hier manuell oder durch einen automatisierten Prozess (siehe F4).

    Diese Anordnung ist gut geeignet um statische Töne zu erzeugen oder Transformationen ähnlich denen Wisharts (Tabelle 2.1 auf Seite 11) in Echtzeit umzusetzen.

  - **F2.3:** Anordnung zu Waveset-Gruppen

Mehrere aufeinanderfolgende Wavesets sollen zu Gruppen zusammengefasst werden können, die in der Synthese als ganzes Stück angeordnet werden. Eine Gruppe wird ausgehend von einem gegebenen Waveset und einer gewünschten Größe aus den umgebenden Wavesets erstellt. Wenn möglich sollen ebenso viele vorhergehende wie nachfolgende Wavesets einbezogen werden.

– **F2.4:** Wiedergabegeschwindigkeit

  Für die Synthese sollen die Wavesets durch Angabe einer Wiedergabegeschwindigkeit im Zeitbereich skaliert werden können. Der Wertebereich für die Abspielrate wird auf 0.01 und 100 sowohl im positiven als auch im negativen (Wiedergabe rückwärts) festgelegt. Die Wellenform der Wavesets ist für Werte zwischen den zeitdiskreten Abtastpunkten zu interpolieren.

– **F2.5:** Wiederholungen

  Für die Wavesets bzw. Gruppen soll sich eine beliebige Anzahl Wiederholungen festlegen lassen, die nahtlos aneinander gereiht werden.

Für alle Syntheseformen sollen sich mehrere aufeinanderfolgende Wavesets zu einer Gruppe zusammenfassen lassen, die als eine Einheit wiedergegeben wird. Für diese Einheiten soll eine beliebige Anzahl Wiederholungen sowie eine beliebige Wiedergabegeschwindigkeit (auch Rückwärts) festgelegt werden können.

- **F3:** Merkmale

  Für jedes Waveset sollen u.a. zur automatischen Auswahl (Anforderung F4) Merkmale (engl. *Feature*) folgende Merkmale zur Verfügung stehen:

  – Die Länge des Wavesets in Sekunden. Da sie den Abstand zwischen drei Nullstellen darstellt, erlaubt sie ähnlich der *Zero Crossing Rate* [27, S. 62] Rückschlüsse auf das Frequenzspektrum des Signals. Kürzere Wavesets deuten auf mehr hochfrequente Anteile hin, während lange Wavesets ein Hinweis für niedrige Frequenzen ist. Dieser Zusammenhang kann aber durch die Mindestlänge für Wavesets (F1.4) aufgehoben werden.

  – Das Quadratische Mittel (RMS für engl. Root Mean Square) der Wellenform gibt Auskunft über die wahrgenommene Intensität eines Tons [27, S. 73].

Es wird für alle Samples $x_i$ eines Wavesets berechnet mit

$$v_{\mathrm{RMS}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}.$$

  - Die Anzahl lokaler Maxima des Signals (engl. *peaks*).
  - **F3.1:** Die Merkmale werden während der Analyse berechnet.
  - **F3.1:** Die berechneten Merkmale stehen in SC für andere UGens zur Verfügung, damit individuelle Algorithmen zur Auswahl von Wavesets implementiert werden können.

- **F4:** automatisierte Auswahl von Wavesets

  Da eine manuelle Auswahl von Wavesets für die Synthese bei der Verarbeitung von Live-Signalen nur bedingt praktikabel ist, soll ein automatisierter Prozess entwickelten werden. Dieser soll Wavesets aus der sich fortlaufend ändernden Menge aus Wavesets auswählen.

  - **F4.1:** Für die verfügbaren Merkmale (siehe F3) sollen Sollwerte angegeben werden können, woraufhin das Waveset mit der besten Übereinstimmung ausgewählt wird. Die beste Übereinstimmung wird bestimmt über die minimale mittlere quadratische Abweichung der Sollwerte zu den Istwerten.
  - **F4.2:** Die Menge der berücksichtigten Wavesets soll auf die letzten $n$ neuesten Wavesets begrenzt werden können.

- **F5:** Speicher für Waveset-Daten

  Die bei der Analyse festgelegten Wavesets und Merkmale müssen an zentraler Stelle gespeichert werden, damit sie für andere Prozesse wie die Synthese zur Verfügung stehen. Sie sollen in einer Liste angeordnet werden und über einen ganzzahligen Index identifiziert werden können.

  In Anbetracht der Tatsache, dass das verarbeitete Live-Signal zeitlich nicht begrenzt ist aber nicht unbegrenzt Speicher zur Verfügung steht, müssen diese Daten irgendwann verworfen werden. Die Größe des Speichers soll frei wählbar sein und bei Erschöpfung des Speichers automatisch die ältesten Daten gelöscht werden.

## 3.4 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben in welche Qualität die gewünschte Funktionalität zu erbringen ist (siehe Abschnitt 2.4.1).

- **N1:** Echtzeitanforderung

  Die Verarbeitung und Erzeugung eines Live-Signals erfordert die Einhaltung harter Echtzeitbedingungen. Um Unterbrechungen im Audiosignal zu verhindern muss sichergestellt sein, dass Berechnungen für das Signal innerhalb der zu Verfügung stehenden Zeit abgeschlossen werden, wobei noch genug Reserven für andere Aufgaben übrig bleiben müssen. Dies hängt natürlich von der eingesetzten Hardware ab. Als Mindestanforderung wird ein Notebookprozessor der vorletzten Generation vorausgesetzt, konkret wird ein Zweikernprozessor mit 2,4 GHz (Intel Core 2 Duo) eingesetzt.

- **N2:** Wiederverwendbarkeit

  Die Software soll für unterschiedliche Anwendungen eingesetzt werden können und deshalb möglichst Modular und flexibel in der Funktionalität umgesetzt werden.

- **N6:** Weiterentwickelbarkeit

  Das System soll für zukünftige Projekte leicht anpassbar und erweiterbar sein.

- **N3:** Benutzbarkeit

  Die SC-Erweiterung soll sich leicht in individuelle Anwendungen einbinden lassen und dabei möglichst viele Möglichkeiten für die Anpassung des Analyse- und Syntheseprozesses bieten. Gleichzeitig sollen die Schnittstellen einfach gehalten werden, damit sie am Ende in Live-Einsatz in der Komplexität auch für die KünstlerInnen handhabbar ist.

- **N4:** Plattform

  Das System wird als Erweiterung für SuperCollider unter Mac OS X implementiert.

- **N5:** Zuverlässigkeit

  Das System darf keine Audioausfälle erzeugen. Dafür muss es neben den Echtzeitanforderungen auch Fehlertolerant sein. Für Fehlerfälle sind Konzepte zu entwickeln, wie ein Prozess sinnvoll fortzusetzen ist. Nur als letzte Option darf Stille ausgegeben werden.

## 3.5 Produktmodell

Ausgehend von den Anforderungen wurde mittels Objektorientierter Analyse (siehe Abschnitt 2.4.2) ein abstraktes Produktmodell geschaffen. Entscheidend beeinflusst ist dieses vom Konzept der Synthesebausteine (*Unit Generators*, kurz *UGens*) in SuperCollider. Zunächst werden die Funktionen auf UGens mit Ein- und Ausgangssignalen verteilt und anschließend deren Verhalten genauer beschrieben.

### 3.5.1 Aufteilung in UGens

Die Produktfunktionen wurden auf unterschiedliche Bausteine (UGens) verteilt, die je nach Anwendung individuell zu einer Anwendung zusammengesetzt werden können (siehe Abb. 3.3). Sie haben unterschiedliche Ein- und Ausgabesignale (siehe Abb. 3.4) und besitzen alle eine Referenz auf eine zentrale Einheit zum Verwalten der Waveset-Daten.

### 3.5.2 Waveset-Analyse

Das UGen *Analysis* führt die Waveset-Analyse eines Eingangssignals durch, es kann auf Wunsch angehalten werden und berechnet Merkmale (Anforderungen F1.1, F1.3 und F3.1). Die festgelegen Wavesets mit Merkmalen werden in einer zentralen Einheit *WavesetData* gespeichert (Anforderung F5). Die Eingangssignale (siehe Abb. 3.4) des *Analysis* UGens sind:

- *wsData:* zentrale Einheit zum Speichern der Waveset-Daten.
- *in:* Eingabesignal, das analysiert wird.
- *active:* Analyse aktiv (Wert $> 0$) oder anhalten (Wert $<= 0$).
- *minWSLen:* Mindestlänge für Wavesets in Sekunden (Anforderung F1.4).

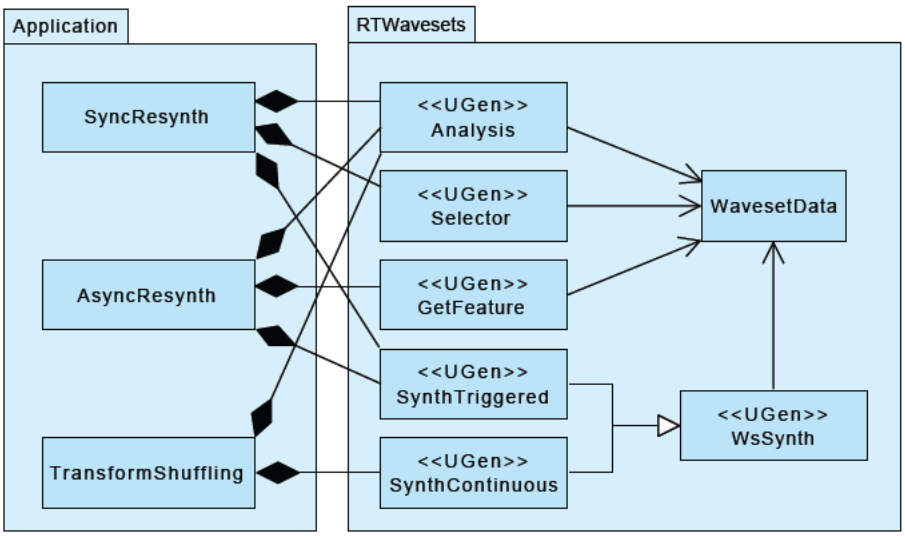


**Abbildung 3.3:** UML-Klassendiagramm mit UGens und Beispiele für die Komposition zu Anwendungen. Eine synchrone sowie asynchrone Resynthese kann aus dem Analyse-UGen und dem UGen für die ereignisgesteuerte Synthese (*SynthTriggered*) zusammengesetzt werden. Für Waveset-Transformationen eignet sich das UGen *SynthContinuous*.

Das Ausgabesignale sind:

- *firstIdx:* der Index des ältesten verfügbaren Wavesets.
- *lastIdx:* der Index des neuesten verfügbaren Wavesets.

In Abbildung 3.5 wird der Vorgang der Waveset-Analyse mit einem Zustandsdiagramm genauer spezifiziert. Die Analyse kann durch das Eingangssignal *active* gestartet (Wert $> 0$) oder angehalten werden (Wert $\leq 0$). Ist die Analyse aktiv werden die Samples des eingehenden Audiosignals fortwährend gespeichert und auf Nulldurchgänge von negativ nach positiv untersucht. Sind zwei aufeinanderfolgende Nulldurchgänge gefunden, stellen diese das Intervall für ein mögliches Waveset dar. Die Länge des Intervalls wird geprüft und wenn diese die Mindestlänge für Wavesets erreicht hat, wird es abgespeichert. Ist die Mindestlänge nicht erreicht, wird auf den nächsten Nulldurchgang von negativ nach positiv gewartet. Wurde ein Waveset festgelegt wird das Ende des letzten Wavesets zum Startpunkt des nächsten.

### 3.5.3 Merkmalswerte

Das UGen *GetFeature* stellt die berechneten Merkmalswerte für anderen UGens zur Verfügung (Anforderung F3.1). Das gewünschte Merkmal wird als Symbol angegeben

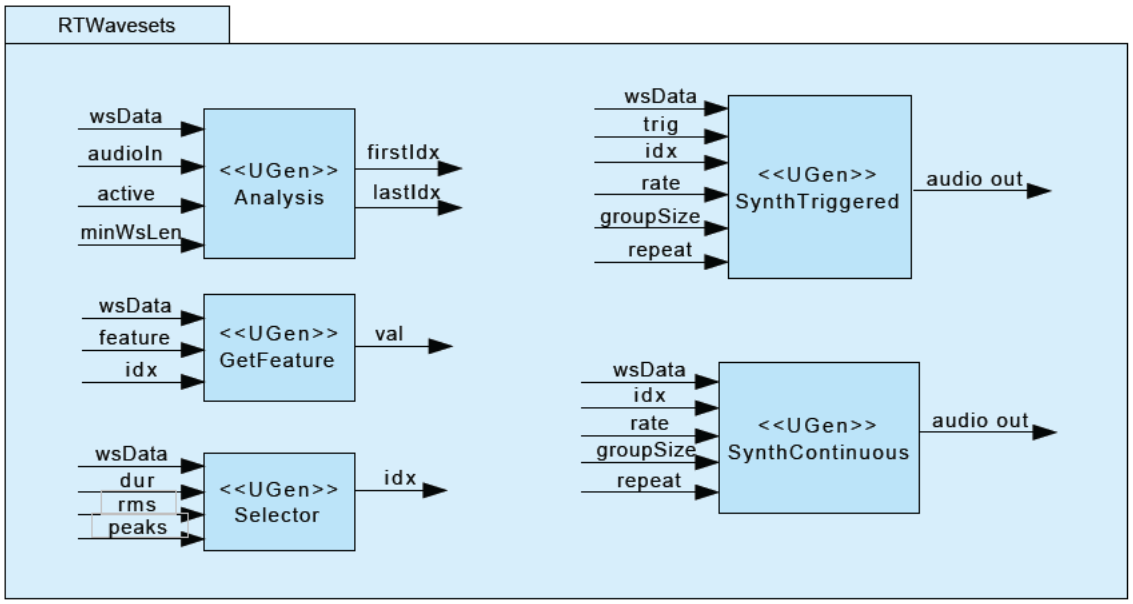


**Abbildung 3.4:** Die UGens mit Ein- und Ausgabesignalen. Alle UGens erhalten eine Referenz auf die zentrale Einheit für die Verwaltung von Waveset-Daten.

(z.B. \dur, \rms, \peaks). Das betreffende Waveset wird wie üblich über den Index angegeben.

### 3.5.4 Waveset-Auswahl

Das UGen *Selector* führt die in der Anforderung F4 geforderte automatische Auswahl von Wavesets durch. Es greift auf die während der Analyse berechneten Merkmale zurück und ermittelt anhand der gegebenen Kriterien das Waveset mit der besten Übereinstimmung. Die Eingangssignale des *Selector* UGens sind:

- *wsData:* Zentrale Einheit mit Waveset-Daten.
- *desiredFtrs:* Sollwerte für die Merkmale des Wavesets (z.B. *dur*, *rms*, *peaks*).
- *lookback:* Nur die letzten $n$ Wavesets werden für die Auswahl berücksichtigt (Anforderung F4.2).

Ausgangssignal ist der Index des ausgewählten Wavesets.

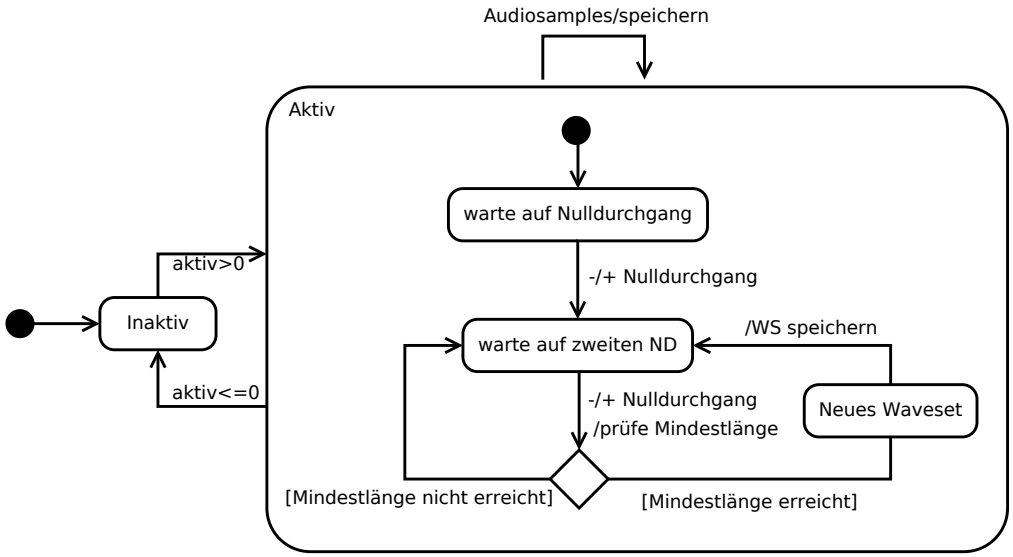


**Abbildung 3.5:** Das Zustandsdiagramm zur Waveset-Analyse. Ist die Analyse aktiv wird das Signal zwischen zwei negativ-positiv-Nulldurchgängen als Waveset gespeichert, sofern die Mindestlänge erreicht wird.

### 3.5.5 Waveset-Synthese

Die beiden Synthesetypen werden in zwei getrennten UGens umgesetzt. Das UGen *SynthTriggered* erlaubt die in Anforderung F2.1 geforderte Event-basierte Synthese. Mit dem Eingangssignal *trig* wird die zeitliche Anordnung eines Wavesets bestimmt, d.h. die Wiedergabe eines Wavesets bzw. einer Gruppe ausgelöst. In Abbildung 3.6 ist der zeitliche Verlauf einer Synthese mit dem UGen SynthTriggered dargestellt. Der Wiedergabevorgang eines Wavesets wird ausgelöst, wenn das Trigger-Signal die Grenze zum Positiven überschreitet. In diesem Moment werden die Parametersignale für die Wiedergabe abgegriffen und die Wiedergabe des Wavesets mit dem angegebenen Index gestartet. Für eine laufende Waveset-Wiedergabe können keine Parameter mehr verändert werden. Das UGen *SynthContinuous* setzt die nahtlose Aneinanderreihung von Wavesets nach Anforderung F2.2 um. Am Ende jedes Wavesets wird als nächstes das Waveset mit dem aktuell angegebenen Index eingereiht (siehe Abb. 3.7).

Beide Synthese-UGens haben folgende einheitliche Eingangssignale um die Synthese zu steuern:

- *idx:* Identifikator des Wavesets
- *repeats:* gibt die Anzahl der Wiederholungen pro Gruppe an

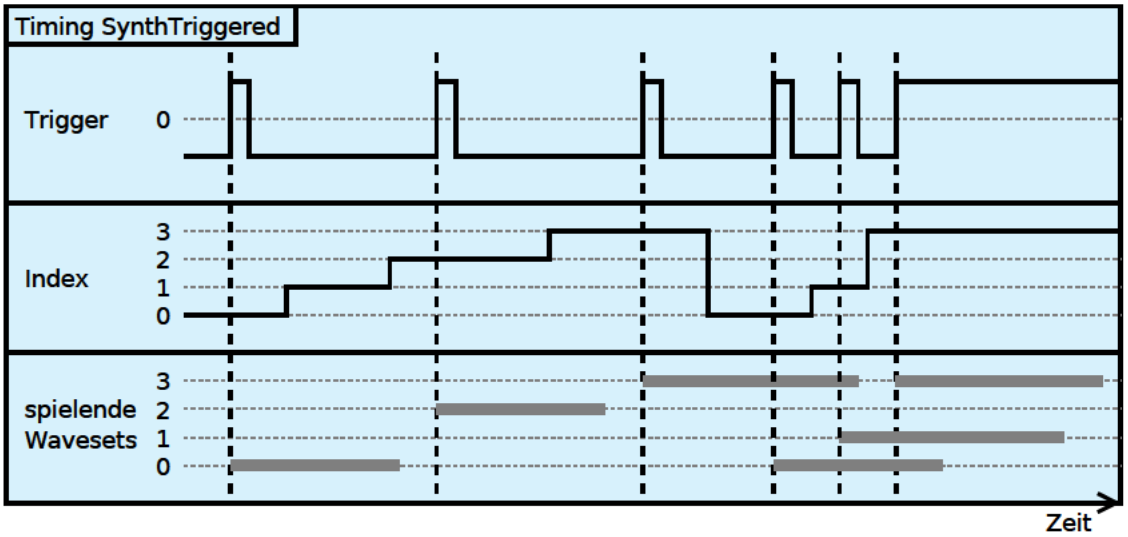


**Abbildung 3.6:** Timing-Diagramm der Trigger-gesteuerten Synthese. Beim Auslösen des Triggers (gestrichelte vertikale Linie) wird eine neue Wiedergabe mit dem aktuell als Index angegebenen Waveset gestartet.

- *rate:* die Skalierung der Abspielgeschwindigkeit
- *groupSize:* die Anzahl Wavesets, die zu einer Gruppe zusammengefasst werden

### 3.5.6 Waveset-Daten

In der Klasse *WavesetData* werden die Waveset-Daten mit Merkmalen in einer Liste gespeichert. Die Wavesets werden über einen eindeutigen Index identifiziert. Die gespeicherten Daten umfassen:

- Waveset Start- und Endpunkt
- die berechneten Merkmale zu den Wavesets
- das Audiosignal

### 3.5.7 Zusammensetzung zu einer Anwendung

Die UGens können je nach Bedarf der jeweiligen Anwendung individuell zu einer Resynthese zusammengesetzt werden. Als Beispiel ist in Abb. 3.8) eine Transformation ähnlich der Transposition nach Wishart zu sehen (siehe Tabelle 2.1 auf Seite 11).

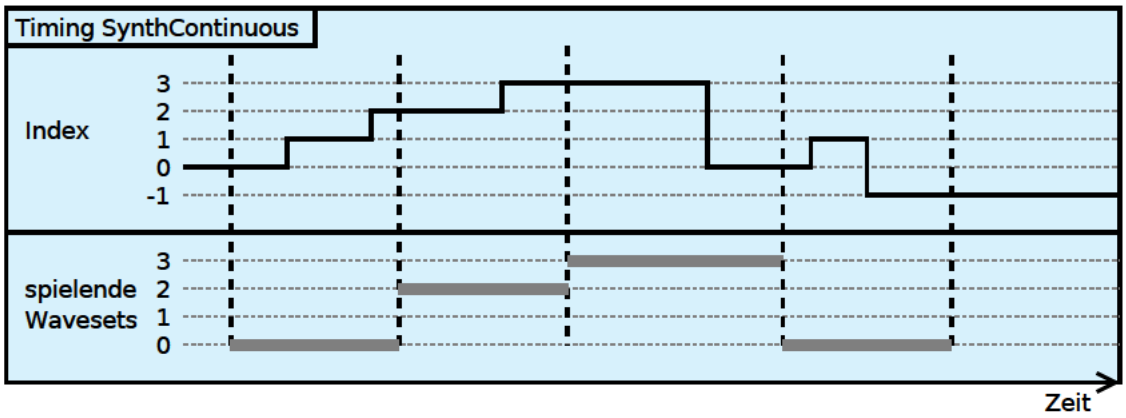


**Abbildung 3.7:** Timing-Diagramm der kontinuierlichen Synthese. Am Ende eines Wavesets folgt nahtlos das zu diesem Zeitpunkt als Index angegebene Waveset.

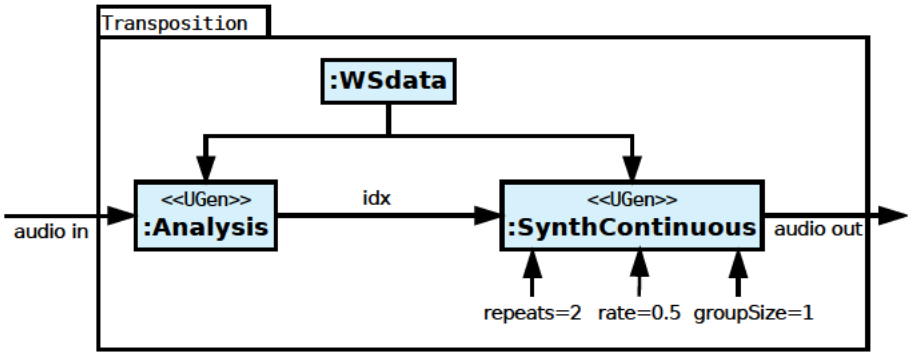


**Abbildung 3.8:** Signalfluss einer Live-Transposition aufgebaut aus RTWaveset-UGens. In einer Instanz des Analysis-UGens werden werden Wavesets extrahiert und gespeichert. Der Index des jeweils letzten Wavesets wird an die kontinuerliche Synthese weitergegeben. Diese nimmt das jeweils letzte Waveset und gibt es mit jeweils halber Geschwindigkeit wieder.

Das jeweils letzte Waveset wird dabei mit halber Geschwindigkeit wiedergegeben. Dies führt zu einem Weglassen von Wavesets und einer Veränderung der Tonhöhe bei gleichzeitig Waveset-typischer Verzerrung. Zu beachten ist, dass durch die unterschiedliche Länge der Wavesets nicht genau jedes zweite Waveset wiedergegeben wird, sondern das jeweils aktuellste. Damit entspricht sie nicht genau der für Aufnahmen definierten Transposition nach Wishart, kann aber als eine „Live-Variante“ davon betrachtet werden.

In Abbildung 3.9 wird der Signalfluss einer Beispielanwendung mit synchroner Synthese dargestellt. Das UGen *SynthTriggered* lässt sich mit einem durch das *Impulse*-UGen [54] erzeugten Trigger einfach für eine synchrone Synthese einsetzen. Es muss nur die Grain-Dichte als Frequenz übergeben werden. Die vom *Selector*-

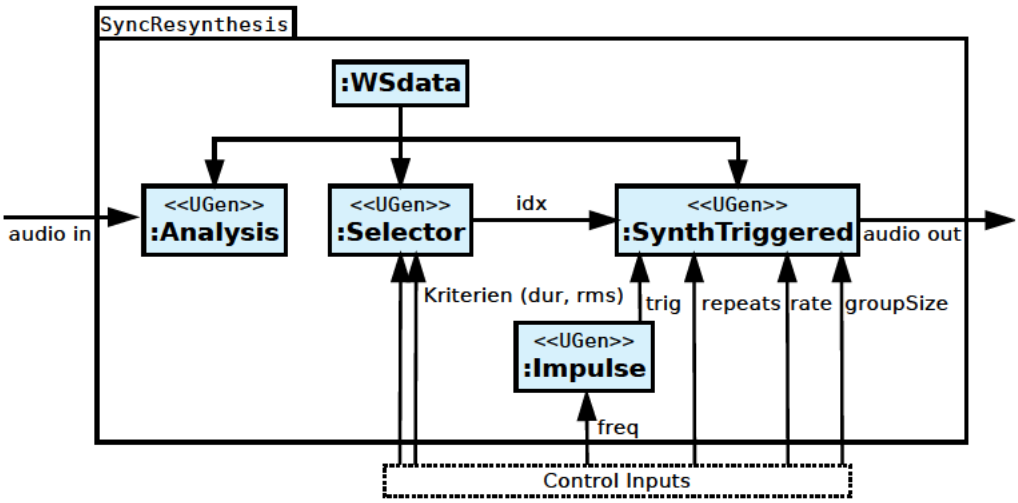


**Abbildung 3.9:** Signalfluss einer synchronen Waveset-Resynthese als Beispiel. In einer Instanz des Analysis-UGens werden werden Wavesets extrahiert und gespeichert. Anhand von Kriterien wird mit dem *Selector*-UGen ein Waveset ausgewählt. Das Impulse-UGen setzt sich mit dem SynthTriggered UGen zu einer synchronen Waveset-Synthese zusammen.

UGen ausgewählten Wavesets werden dann in regelmäßigen Abstand entsprechend der Frequenz angeordnet. Weitere Eingangssignale zur Steuerung der Synthese sind die Kriterien für die Waveset-Auswahl, die Anzahl der Wiederholungen pro Gruppe, die Wiedergabegeschwindigkeit und die Gruppengröße.

# 4 Softwaredesign

Das im letzten Kapitel aus den Anforderungen abgeleitete allgemeine Produktmodell wurde mittels objektorientiertem Design zur einer konkreten Softwarearchitektur weiterentwickelt. Zunächst wird die globale Architektur beschrieben und anschließend die einzelnen Subsysteme und Komponenten im Detail. Auf konkrete Einflussfaktoren (siehe Abschnitt 2.4.3), die Designentscheidungen beeinflusst haben, wird im weiteren Verlauf eingegangen.

## 4.1 Globale Architektur

Wichtiger Faktor für die Architektur ist die Umsetzung des Systems als SuperCollider-Erweiterung (umfasst die Einflussfaktoren: Anwendungsart, Verteilungsart, Kontext des Anwendungssystems, Art der softwaretechnischen Infrastruktur nach Balzert).

Die fertige SC-Erweiterung besteht aus den Artefakten SC-Klassendefinition, SC-Hilfedateien und dem kompilierten C++-Modul als dynamische Bibliothek (siehe Abb. 4.1). Diese müssen in der für Erweiterungen vorgegebene Ordnerstruktur (siehe Abschnitt 2.3.4) angeordnet werden. Zur Installation wird diese Ordnerstruktur in das Erweiterungsverzeichnis von SuperCollider kopiert. Von dort wird beim Starten des Servers (*SCSynth*) die dynamische Bibliothek in den Arbeitsspeicher geladen und eingebunden. Auf Clientseite wird die SC-Klassendefinition beim Start des Interpreters (*SCLang*) während des Aufbaus der Klassenbibliothek geladen. Der Server und der Client-Prozess können durch die grundsätzliche Server-Client Architektur von SuperCollider auf unterschiedlichen Rechnern ausgeführt werden.

Entsprechend der Verteilung ist das System in die zwei Subsysteme SC-Klassendefinition und C++-Modul aufgeteilt. Nach dem Architekturprinzip „Trennung von Zuständigkeiten“ wird das das C++ Modul weiter unterteilt in die Komponenten *RTWavesetUGens*, WavesetProcessing und WavesetStorage (siehe Abb. 4.2). In der Komponente *RTWavesetUGens* werden die UGens nach der von SC vorgegebene C++-Schnittstelle implementiert. Die Eingangssignale werden über entsprechende Schnittstellen an die Komponente *WavesetProcessing* weitergegeben, wo die Erzeugung und Verarbeitung von Wavesets umgesetzt wird (Anwendungslogik). Im

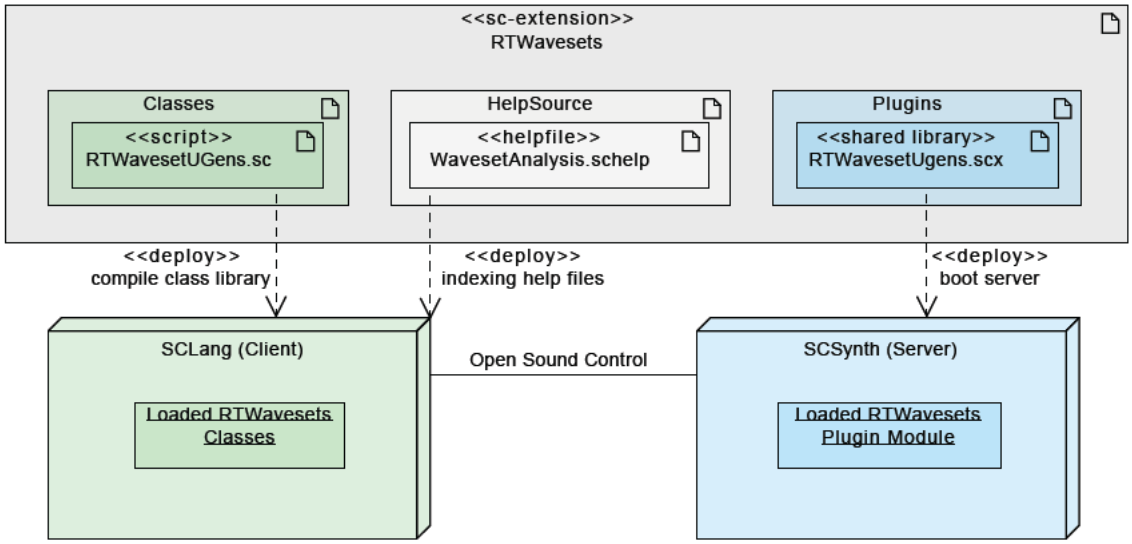


**Abbildung 4.1:** Das Verteilungsdiagramm der Artefakte des Systems. Die SC-Klassendefinition (*RTWavesetUGens.sc*) wird beim Starten des Interpreters geladen (in der Abbildung grün). Das C++-Modul (*RTWavesetUgens.scx*) wird vom Server beim Start als dynamische Bibliothek eingebunden (in der Abbildung blau).

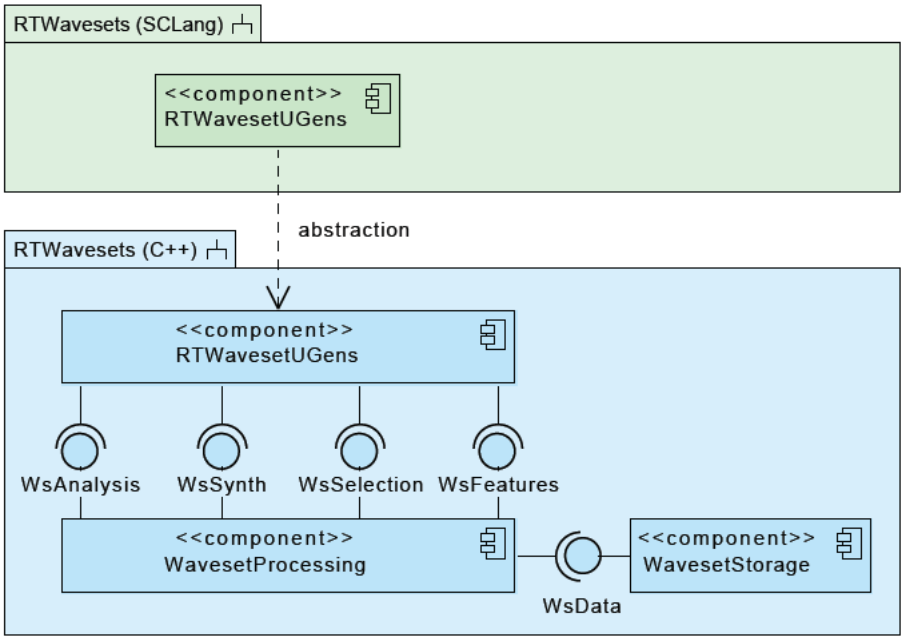


**Abbildung 4.2:** Die Komponenten des RTWavesets Systems. Auf SCLang-Seite (grün) werden die UGens als SC-Klassen definiert, welche die Schnittstellen für die Benutzung mit Ein- und Ausgabesignalen sind. Im C++-Plugin (blau) werden die Signalverarbeitungsprozesse durch mehrere über Schnittstellen verbundene Komponenten realisiert.

Paket WavesetStorage werden die Schnittstellen zur Speicherung und Verwaltung der Waveset-Daten realisiert.

Die Trennung der Zuständigkeiten führt zu einer modularen Architektur und erhöht die Sichtbarkeit [5, S. 31]. Durch die Programmierung gegen Schnittstellen zwischen den Komponenten wird die Abhängigkeit gegenüber konkreten Klassen vermieden. Dies unterstützt die nichtfunktionalen Anforderungen Wiederverwendbarkeit (N2) und Weiterentwickelbarkeit (N6) [5, S. 120]. Zum Beispiel kann leichter die Realisierung des Waveset-Datenspeichers ausgetauscht oder die Anwendung auf andere Plattformen portiert werden.

## 4.2 Unit Generators

Die UGens können unmittelbar nach den bei der objektorientierten Analyse (Abbildung 3.4 auf Seite 33) definierten Synthesebausteinen mit Ein- und Ausgabesignalen entworfen werden. Jedes UGen wird auf SCLang-Seite durch eine Klasse mit *ar*-Methode (Signale im Audiotakt) und/oder *kr*-Methode (Signale im Steuertakt) mit den definierten Eingangssignalen als Argumente umgesetzt (siehe Abb. 4.3). Da SuperCollider kein Konzept wie Pakete oder Namensräume bietet, wurde den UGens beim Namen das Prefix *Waveset** vorangesetzt. Sie erben von der Klasse *UGen* beziehungsweise *MultiOutUGen* bei mehreren Ausgangssignalen.

Im C++-Modul werden die UGens entsprechend der vorgegebenen Schnittstelle als klassenähnliche Konstrukte mit Konstruktor, Destruktor und *next*-Methode umgesetzt. Die Ein- und Ausgabesignale sind hier nicht im Entwurf sichtbar, sondern werden zur Laufzeit abgefragt. Die Aufgabe der C++-Ugens ist hier darauf begrenzt, der SC-Schnittstelle zu entsprechen und die Eingangssignale entgegen zu nehmen und gegebenenfalls zu interpretieren. Die Signale werden dann über entsprechende Schnittstellen an das Paket *WavesetProcessing* weitergegeben. Für allgemeine Aufgaben steht übergeordnet die Klasse *WavesetUGen*, von der alle anderen UGens erben.

Auf eine explizite Definition abstrakter Schnittstellen für das *WavesetProcessing*-Paket wurde in diesem Fall verzichtet. Die Entscheidung fiel nach einer Abwägung zwischen dem Vorteil einer losen Kopplung zwischen den UGens und den Klassen des *WavesetProcessing*-Pakets gegenüber der zusätzlichen Komplexität und Zusatzaufwand bei Änderungen der Schnittstellen. Da der Austausch der *WavesetProcessing*-

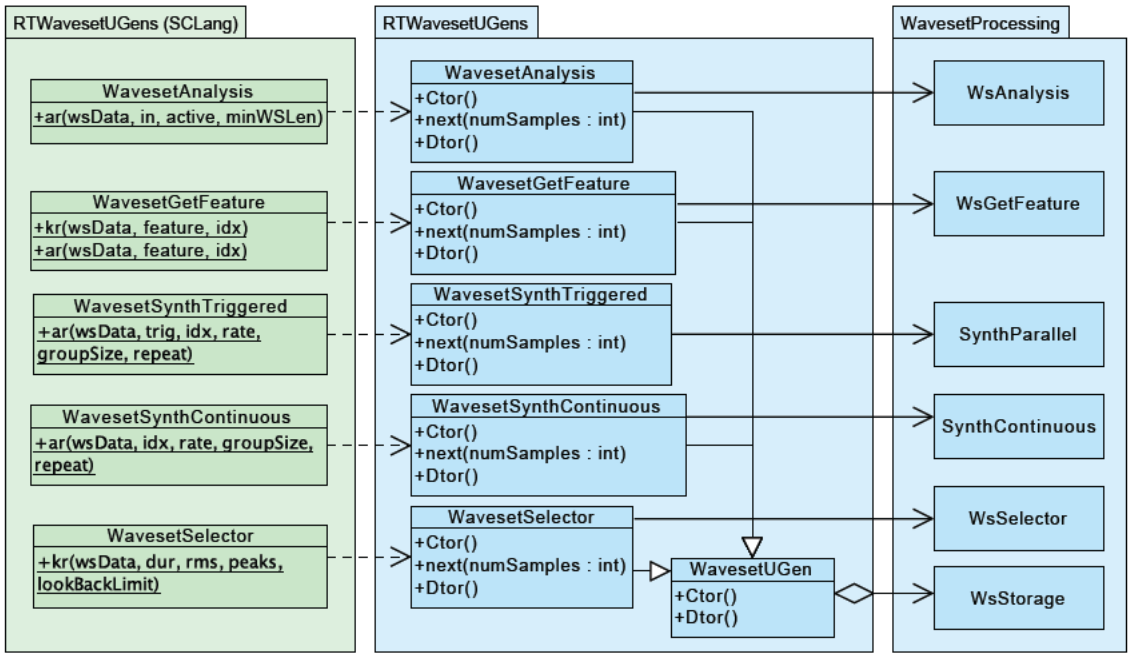


**Abbildung 4.3:** Klassendiagramm mit den UGens auf SCLang-Seite (grün) und C++-Seite (blau). Jedes UGen wird durch eine Klasse auf SCLang-Seite und einer Klasse in C++ repräsentiert. Die eigentliche Signalverarbeitung findet separat im Paket *WavesetProcessing* statt.

Implementierung weniger wahrscheinlich als die Änderung von Ein- und Ausgängen erscheint, fiel die Entscheidung gegen die explizite Definition von Schnittstellen und es wird direkt auf die Klasse verwiesen.

## 4.3 Waveset-Analyse

Bei der Analyse wird das Eingangssignal in Wavesets aufgeteilt, die zusammen mit berechneten Merkmalen gespeichert werden. Das UGen *WavesetAnalysisUGen* nimmt das Eingangssignal zusammen mit den Parametern entgegen. Die Interpretation des *active* Parameters zum Anhalten und Fortsetzen der Analyse wird direkt im UGen umgesetzt, da diese Schnittstelle zum Anhalten/Fortsetzen als SuperCollider spezifisch eingeordnet werden kann. Die restliche Anwendungslogik, wie im Zustandsdiagramm Abbildung 3.5 spezifiziert, wird von der Klasse *WsAnalysis* umgesetzt.

Die Erzeugung neuer Waveset-Instanzen erfolgt nach dem Erbauer Entwurfsmuster über die Schnittstelle *WavesetBuilder*. Diese erlaubt das kontinuierliche Sammeln von Audiosamples für ein Waveset und nach Abschluss das automatische Speichern

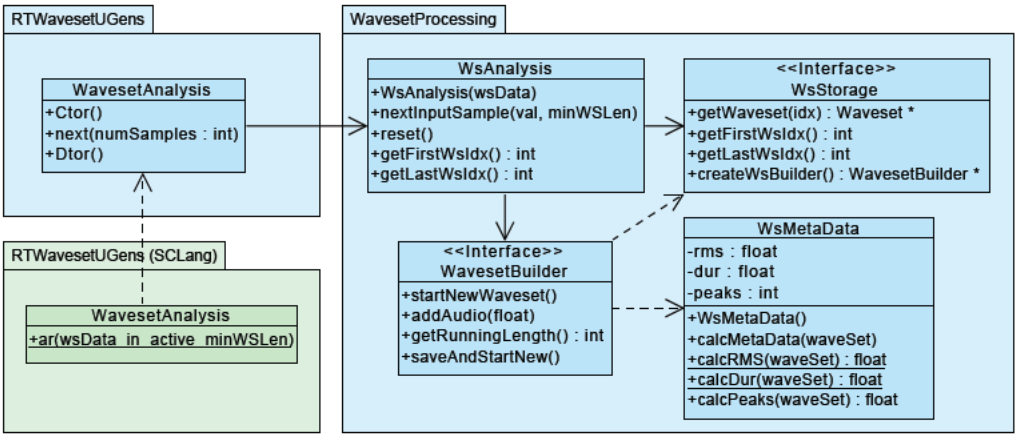


**Abbildung 4.4:** Diagramm der an der Analyse beteiligten Klassen. Auf SCLang-Seite (grün) wird das *WavesetAnalysis*-UGen mit Ein- und Ausgabesignalen definiert. Auf C++-Seite (blau) nimmt die Klasse WavesetAnalysis über die vorgegebene Schnittstelle die Signale entgegen und gibt sie zur Verarbeitung an das *WavesetProcessing*-Paket weiter.

des Wavesets. Die Berechnung von Merkmalen erfolgt dabei implizit durch die Klasse *WsMetaData* beim Erzeugen des Wavesets durch den Erbauer.

## 4.4 Waveset-Merkmale und automatische Auswahl

Für die automatische Auswahl von Wavesets werden für jedes Waveset Merkmale berechnet und gespeichert. Die zur Verfügung stehenden Merkmale werden an zentraler Stelle in einer eigenen Klasse *WsMetaData* definiert (siehe Abb. 4.5). Diese wird genutzt für:

- die Berechnung der Merkmalswerte bei der Analyse (siehe Abschnitt 4.3)
- die Speicherung der Merkmalswerte als Instanz der Klasse zusammen mit den Wavesets (mehr dazu in Abschnitt 4.6)
- den Zugriff auf die Merkmalswerte für die direkte Ausgabe
- den Zugriff auf die Merkmalswerte für die automatische Auswahl von Wavesets

Zur Abfrage von Merkmalswerten mit dem UGen *WavesetGetFeature* wird auf SCLang-Seite als Parameter das gewünschte Merkmal als Symbol (String) und der Index des Wavesets angegeben. Da zwischen SCLang und SCSynth nur numerische Signale ausgetauscht werden können, ist zur Auswahl von Merkmalen die Festlegung

von eindeutigen und ganzzahligen Identifikatoren notwendig. Die Festlegung erfolgt in der Klasse *WsMetaData* mit einem Aufzählungstyp. Vor der Übergabe an SCSynth wird das Symbol in den entsprechenden Identifikator umgesetzt. Der Zugriff und die Rückgabe des Werts durch die Klasse *WsFeatureExtractor* ist trivial. Die zusätzliche Klasse hierfür wurde trotzdem zugunsten der konzeptionellen Integrität entworfen.

Für die automatische Auswahl von Wavesets werden Sollwerte für die Merkmale übergeben, die im C++-Teil des *WavesetSelector* UGens zu einem *WsMetaData*-Objekt zusammengesetzt werden. In der Klasse *WsSelector* wir dann nach den gegebenen Kriterien das beste Waveset gesucht und dessen Index zurückgegeben.

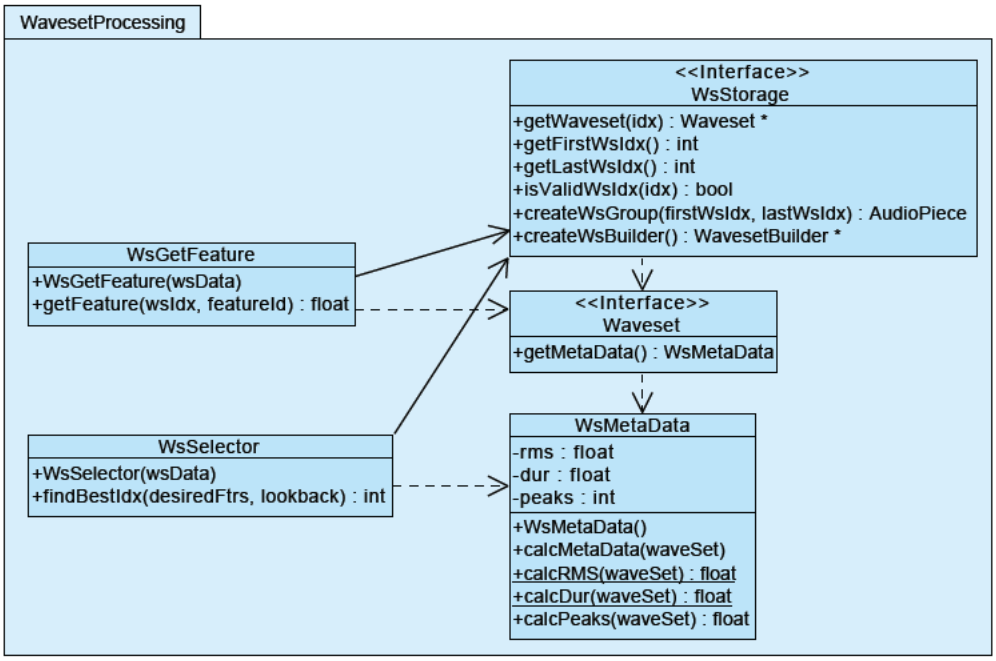


**Abbildung 4.5:** Diagramm mit den bei der Verarbeitung von Merkmalen beteiligten Klassen. Die Klasse *WsMetaData* ist zentrale Stelle für die Speicherung und Berechnung von Merkmalen. Die Klassen *WsGetFeature* und *WsSelector* können über die Waveset-Schnittstelle auf berechnete Merkmale zugreifen.

## 4.5 Waveset-Synthese

Für die Waveset-Synthese gibt es zwei UGens wie im Produktmodell (Abschnitt 3.5.5) definiert. Die Anwendungslogik der Synthese ist auf die Klassen *SynthParallel*, *SynthContinous* und *WsPlayer* verteilt (siehe Abb. 4.6).

Für die nahtlose Aneinanderreihung von Wavesets mit dem *WavesetPlayerConti-*

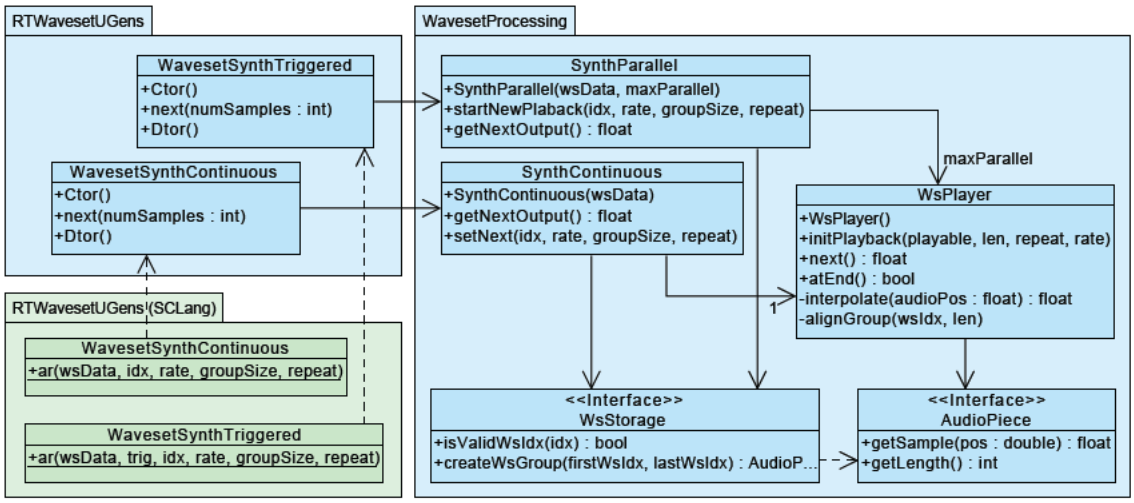


**Abbildung 4.6:** Diagramm mit den an der Synthese beteiligten Klassen. Auf SCLang-Seite (grün) werden die Synthese-UGens mit Ein- und Ausgabesignalen definiert. Auf C++-Seite (blau) werden die Signale für die Steuerung der Synthese entgegengenommen (Paket (*RTWavesetUGens*) und das Signal der Synthese berechnet (Paket *WavesetProcessing*).

*nous* UGen werden die Wiedergabeparameter jeweils aktuell an die Klasse *SynthContinous* übertragen. Diese startet intern direkt nach Ende des aktuellen Wavesets eine neue Wiedergabe mithilfe einer Instanz der Klasse *WsPlayer*.

Für die Trigger-gesteuerte Synthese wird im UGen *(WavesetPlayerTriggered)* das Trigger-Signal beobachtet und bei Auslösung ein Methodenaufruf der Klasse *SynthParallel* ausgeführt, um eine neue Wiedergabe zu starten, die Wiedergabeparameter werden durchgereicht. Die parallelen Wiedergabevorgänge werden durch mehrere Instanzen der Klasse *WsPlayer* realisiert. Die Anzahl der Instanzen entspricht dabei der maximal möglichen Anzahl paralleler Wiedergaben.

Beide Synthesetypen greifen auf die Klasse *WsPlayer* zurück, in der die eigentliche Wiedergabe umgesetzt wird. Für einen neuen Wiedergabevorgang wird zunächst die abzuspielende Waveset-Gruppe festgelegt und abgefragt. Auf die Audiodaten kann dann über die Schnittstelle *AudioPiece* zugegriffen werden. Darauf aufbauend wird in einer entsprechenden Methode die Interpolation für die Wiedergabe mit veränderter Geschwindigkeit implementiert.

## 4.6 Zugriff und Verwaltung der Waveset-Daten

Für die Waveset-Daten gibt es eine zentrale Einheit, in der diese verwaltet werden und die sie für alle UGens zur Verfügung stellt. Die Realisierung mit zwei getrennten Puffern für Audio- und Waveset-Daten ist getrennt von einer abstrakten Schnittstelle für den Zugriff auf die Daten (siehe Abb. 4.7).

### 4.6.1 Schnittstelle für den Zugriff

Im Paket *WavesetProcessing* sind abstrakte Schnittstellen definiert, die den Zugriff weitgehend unabhängig von der Art der internen Speicherstruktur und Verwaltung ermöglichen. Durch die Programmierung gegen abstrakte Schnittstellen wird die Abhängigkeit zu konkreten Klassen reduziert, was die Weiterentwickelbarkeit begünstigt (nichtfunktionale Anforderung N6, vgl. [5, S. 503]). In diesem Fall hat es den Vorteil, dass neue Varianten der Speicherung von Wavesets implementiert werden können, ohne Änderungen in der Anwendungslogik (Paket *WavesetProcessing*).

Konkret wird der Zugriff auf Wavesets über einen eindeutigen Index ermöglicht. Jedes Waveset hat eine Reihe von Audiosamples sowie Metadaten (z.B. Merkmalswerte), auf die zugegriffen werden kann. Aufeinanderfolgende Wavesets können zu Gruppen zusammengefasst werden, auf dessen Audiosamples über die gleiche Schnittstelle (*AudioPiece*) zugegriffen werden kann. Für die Erzeugung neuer Wavesets steht ein Erbauer (engl. *Builder*) nach dem Erbauer-Entwurfsmuster zur Verfügung.

### 4.6.2 Realisierung mit zwei Puffern

Im Paket *WavesetStorage* werden die abstrakten Schnittstellen in Form einer getrennten Speicherung von Audio- und Waveset-Daten in je einem Ringpuffer realisiert (siehe Abb. 4.7). Diese werden in zwei als SC-Puffer übergebenen Speicherbereichen fester Größe angelegt. Zentrale Einheit für den Zugriff und die Verwaltung der Daten ist die Klasse *WsStorageDualBuf*. Über sie erfolgt der externe Zugriff auf Wavesets und Gruppen sowie der interne Zugriff auf Audiodaten. Da die Wavesets abhängig von den Audiodaten sind, müssen die beiden Puffer konsistent gehalten werden. Hierfür müssen Wavesets, dessen Audiodaten nicht (mehr) verfügbar sind, entfernt werden (Methode *cleanUp()*).

Ein Waveset wird als Start- und Endposition innerhalb des Audiopuffers zusam-

men mit den Metadaten gespeichert (Klasse *WavesetDualBuf*). Entsprechend der vorgegebenen Schnittstelle sind Methoden zum Zugriff auf die Audiosamples und der Abfrage von Metadaten implementiert. Gruppen von Wavesets verhalten sich wie einzelne Wavesets, bieten jedoch keine Metadaten (Schnittstelle *AudioPiece*).

Für den Ringpuffer ohne dynamische Speicherverwaltung ist erforderlich, dass der Datentyp der Metadaten vollständig bekannt ist, deshalb wird hier fest auf die im Paket *WavesetProcessing* definierte Klasse *WsMetaData* verwiesen. Sollte bei der späteren Weiterentwicklung der Wunsch nach einem austauschbarem Datentyp aufkommen, kann dieser als Template-Typ definiert werden. Darauf wurde vorerst aus Gründen der Lesbarkeit verzichtet.

Der Erbauer zum Erzeugen neuer Wavesets erlaubt das kontinuierliche Sammeln von Audiodaten, die direkt im Ringpuffer abgelegt werden (Klasse *WavesetBuilderDualBuf*). Nach Abschluss eines Wavesets wird dieses direkt im Waveset-Puffer abgelegt. Damit der Ringpuffer für beide Elementtypen eingesetzt werden kann, ist er als Template-Klasse realisiert. Ist der zur Verfügung stehende Speicher erschöpft, werden stillschweigend ältere Daten überschrieben. Durch einen fortlaufenden sowie eindeutigen Index wird auf die Elemente zugegriffen. Damit kann nachvollzogen werden, welche Daten noch zu Verfügung stehen.

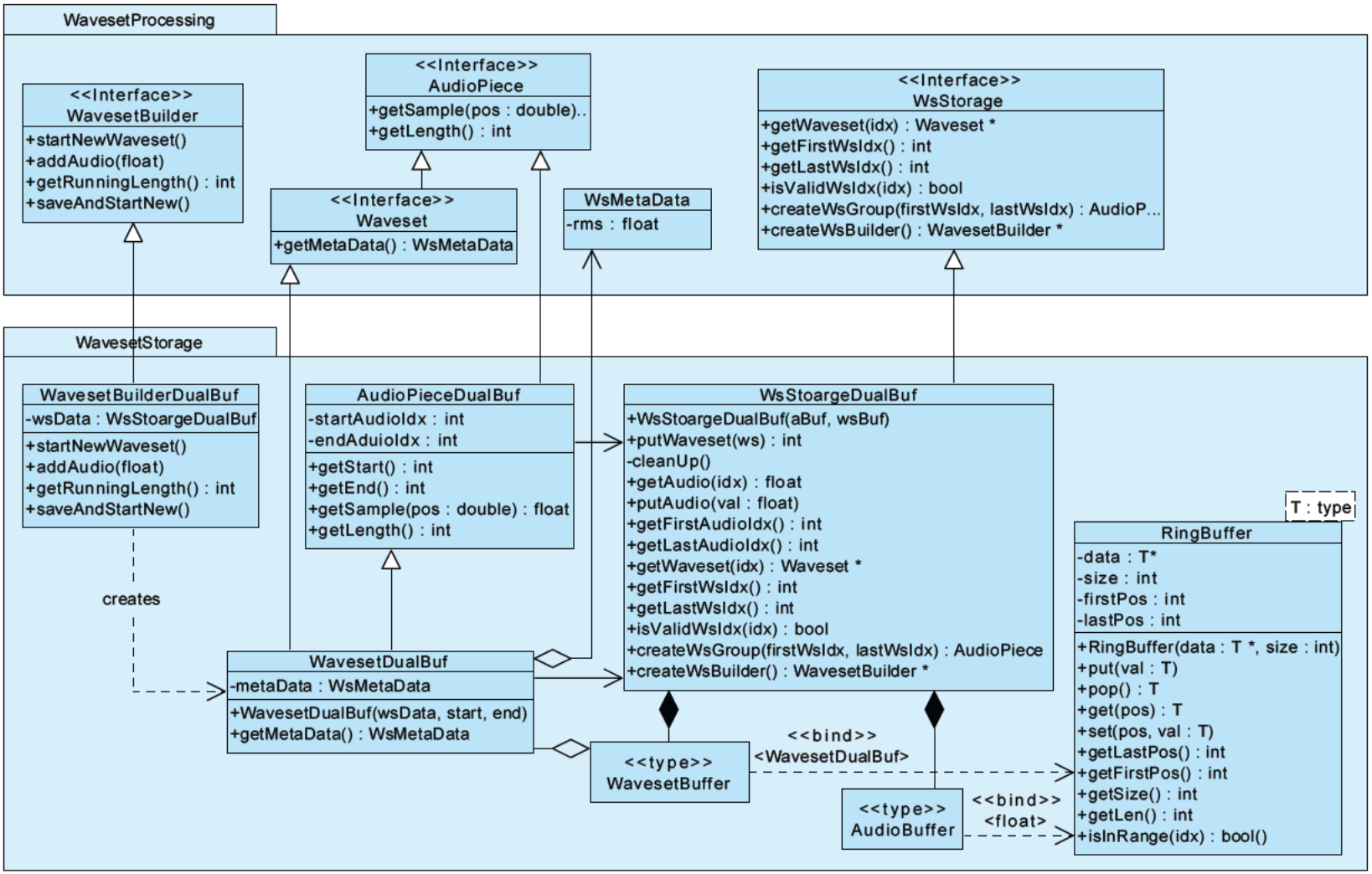


**Abbildung 4.7:** Diagramm mit den an der Waveset-Datenverwaltung beteiligten Klassen. Im Paket *WavesetProcessing* werden getrennt von der Implementierung (Paket *WavesetStorage*) abstrakte Schnittstellen für den Zugriff definiert.

# 5 Implementierung

In diesem Kapitel wird beschrieben, wie die im letzten Kapitel entworfenen Konstrukte mit den Programmiersprachen in SCLang und C++ als Erweiterung für den Syntheseserver umgesetzt wurden. Die getroffenen Implementierungsentscheidungen und Problemlösungen werden reflektiert und hinterfragt. Orientierung geben dabei die in Abschnitt 2.4.4 beschriebenen Implementierungsprinzipien.

## 5.1 Speicherverwaltung im Syntheseserver

ln C++ gibt es drei Arten, um Speicher für Daten zu belegen: global, auf dem Stack und dynamisch auf dem Heap. Für Echtzeitanwendungen sind die ersten beiden unkritisch, die Verwaltung des dynamischen Speichers kann dagegen ein Problem darstellen [10, S. 510]. Innerhalb eines UGens (Echtzeitkontext) darf ein dynamischer Speicher nicht mit den konventionellen Mitteln des Betriebssystems *(malloc, free, new* und *delete)* verwaltet werden, da diese die Ausführung unbestimmt lange blockieren können [57]. Die Nutzung dynamischen Speichers lässt sich aber in einigen Situationen gar nicht oder nur umständlich vermeiden. Insbesondere bei der objektorientierten Programmierung mit C++ spielt die dynamische Speicherreservierung eine zentrale Rolle. Etwa wenn gegen abstrakte Basisklassen bzw. Schnittstellen programmiert werden soll. Wird auf eine Instanz über den abstrakten Basistyp zugegriffen, kann das nur über Zeiger oder Referenzen erfolgen. Eine Deklaration als Variable auf dem Stack ist dann nicht möglich und auch die Rückgabe von Objekten ist nur als Zeiger oder Referenz möglich. Beides führt zu Situationen, in denen eine dynamische Speicherreservierung nur schwer zu umgehen ist, etwa bei der Umsetzung erzeugender Entwurfsmuster wie der abstrakten Fabrik oder Fabrikmethoden.

Abhilfe schafft eine eigene, echtzeitfähige Speicherverwaltung von SuperCollider, die mit den Funktionen *RTAlloc* und *RTFree* zur Verfügung steht. Trotzdem wird von der Nutzung der C++-Standardbibliothek (STL) und anderen Drittbibliotheken, die mit dynamischen Speicher arbeiten und gegebenenfalls auch andere blockierende Elemente enthalten können, abgeraten [57]. Bei einigen Bibliotheken lässt sich die Speicherverwaltung auf die Funktionen von SuperCollider adaptieren, was aber nicht

immer einfach ist. Auch die Nutzung von *RTAlloc* für die Objekte eigener Klassen ist etwas umständlich (siehe Listing 5.1). Zudem hängt der als Präprozessordefinition realisierte Befehl von einer globalen Variable mit der Schnittstelle zum Syntheseserver sowie einen im UGen verfügbaren verweis auf den SC-Kontext *(World)* ab.

Angeforderter dynamische Speicher muss in C++ explizit wieder freigegeben werden, wenn er nicht mehr benötigt wird. Den richtigen Zeitpunkt dafür zu finden ist nicht immer einfach und fehleranfällig, insbesondere, wenn sich mehrere Klassen eine Instanz teilen. Wird Speicher zu früh freigegeben, führt das zu undefiniertem Verhalten und Zugriffsfehlern. Wird er gar nicht freigegeben, steigt der Speicherbedarf während der Laufzeit zunehmend unnötig an (Speicherleck). Um die Speicherfreigabe in C++ zu vereinfachen, gibt es sogenannte intelligente Zeiger, wie *std::shared_ptr* [41, S. 122]. Sie zählen die Anzahl der Referenzen auf einen Speicherbereich und geben ihn automatisch frei, wenn alle Verweise gelöscht wurden. Grundsätzlich können sie die Speicherverwaltung in C++ erheblich vereinfachen und sind deshalb seit C++11 auch fester Bestandteil des Standards. Allerdings nutzen diese als Template-Klassen realisierten Hilfsmittel intern zur Verwaltung selbst dynamischen Speicher, was auf die Speicherverwaltung von SuperCollider adaptiert werden müsste.

Auf den Einsatz intelligenter Zeiger wurde in Anbetracht der überschaubaren Komplexität der Speicherfreigabe in diesem Projekt gegenüber dem erforderlichen Mehraufwand verzichtet. Zudem kann der Einsatz von Template-Klassen die Lesbarkeit des Programmcodes erschweren. Um die Erzeugung dynamischer Objekte zu vereinfachen, wurden eigene Varianten der *new* und *delete* Operatoren implementiert, die dann auf *RTAlloc* und *RTFree* zurückgreifen. Hierfür wurde eine abstrakte Basisklasse *ScObject* geschaffen, in der die Operatoren für alle eigenen Klassen zur Verfügung gestellt werden (siehe Listing 5.2). Von einem Überschreiben der allgemeinen *new* und *delete* Operatoren für alle Objekte wird von James McCartney abgeraten [59].

**Listing 5.1:** Die Speicherallokierung und Initialisierung eines Objekts mit RTAlloc. Der Speicher für das Objekt muss zunächst reserviert werden und wird dann zur Initialsierung des Objekts dem *new*-Operator übergeben.

```
static InterfaceTable *ft; // das Präprozessormakro RTAlloc greift unsichtbar auf diese
    ↪ globale Variable zu, die beim Laden des Plugins gesetzt werden muss.

WsAnalysis* createWsAnalysisInstance(WsAnalysisUGen* unit){

    // Speicher für das WsAnalysis Objekt reservieren
    void* space = RTAlloc(unit->mWorld,sizeof(WsAnalysis));

    // Objekt initialisieren
    WsAnalysis* instance = new (space) WsAnalysis(unit->wsData);

    return instance;
}
```

**Listing 5.2:** Die Implementierung angepasster new und delete Operatoren. In einer eigenen Klasse *ScObject* werden Verweise auf die zur Speicherreservierung benötigte SuperCollider-Umgebung gespeichert.

```
void *ScObject::operator new(size_t size)
{
    if(scInterface==NULL) throw "ScObject␣allocation␣error:␣interfacetable␣not␣set";
    if(scWorld==NULL) throw "ScObject␣allocation␣error:␣scWorld␣not␣set";

    void* space = scInterface->fRTAlloc(scWorld,size);

    if(space==NULL) throw "ScObject␣allocation␣error";

    return space;
}

void ScObject::operator delete(void * ptr)
{
    scInterface->fRTFree(scWorld,ptr);
}
```

## 5.2 Implementierung der Unit Generators

Die UGens können direkt nach der vorgegebenen Schnittstelle (siehe 2.3.4) mit Klassen in SCLang (siehe Listing 5.3) und in C++ mit einem klassenähnlichen Konstrukt aus einer Datenstruktur (*struct*) und Funktionen implementiert werden (Quellcode siehe Anhang A.2). Wichtig ist hier die explizite Initialisierung aller Variablen und Objekte. Der Standardkonstruktor der Struktur und dessen Objekten wird hier nicht wie von C++ Klassen gewohnt implizit aufgerufen, deshalb müssen Objekte in der Konstruktorfunktion manuell mit *new* initialisiert werden. Ebenso müssen Destruktoren von Objekten in der Desktruktorfunktion explizit aufgerufen werden.

**Listing 5.3:** Auszug der Definition der RTWaveset-UGens in SuperCollider (vollständige Definition siehe Listing A.1 im Anhang).

```
WavesetAnalysis : MultiOutUGen {
    *ar { arg wsData, in, active=1, minWSLen=0.0005;
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, in, active, minWSLen)
    }

    init { arg ... theInputs;
        inputs = theInputs;
        outputs = [OutputProxy(rate, this, 0),OutputProxy(rate, this, 1)];
        ^outputs
    }
}

WavesetSelector : UGen {
    *ar { arg wsData, dur=(-1), rms=(-1), peaks=(-1),lookBackLimit=(-1);
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, dur, rms, peaks,
            ↪ lookBackLimit)
    }
}

WavesetSynthTriggered : UGen {
    *ar { arg wsData, trig, idx, rate=1, groupSize=1, repeat=1;
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, trig, idx, rate, groupSize,
            ↪ repeat)
    }
}
```

Jedes UGen hat eine *WsStorage*-Instanz zum Zugriff auf die Waveset-Daten und eine Instanz der entsprechenden Klasse aus dem *WavesetProcessing*-Paket mit der Anwendungslogik. Auf eine dynamische Speicherreservierung wurde wo möglich verzichtet und die Instanzen direkt in der Struktur angelegt. Wie im Entwurf festgelegt

werden die Eingangssignale empfangen, gegebenenfalls einfache Steuermechanismen (z.B. Trigger, Ein/Aus-Schalter) interpretiert und an das entsprechende Objekt zur Verarbeitung weitergegeben.

Zwei Beispiele, wie die UGens in SCLang zu einer Resynthese zusammengesetzt werden, können sind in Listing 5.4 und 5.5 zu sehen.

**Listing 5.4:** Minimalbeispiel mit Waveset-Analyse und Synthese. Bei der Synthese werden die jeweils letzten fünf Wavesets zu einer Gruppe zusammengefasst und mit dreifacher Geschwindigkeit wiedergegeben. Dies führt zu Wiederholungen von Wavesets und hebt die Tonhöhe bei gleichzeitiger Waveset-typischer Verzerrung.

```
(
var wsData = WavesetData.new;

{
    var idx;
    var rate = 3.0;
    var groupSize = 5;
    var repeat = 10;
    #idxFirst,idxLast = WavesetAnalysis.ar(wsData, SoundIn.ar(0), 1);
    WavesetSynthContinuous.ar(wsData,idxLast,rate,groupSize,repeat);
}.play;
)
```

**Listing 5.5:** Erweitertes Beispiel für die Benutzung der RTWaveset-Ugens. Für eine asynchrone Synthese werden Wavesets anhand von Merkmalen ausgewählt und zufällig angeordnet.

```
(
var wsData = WavesetData.new;
{
    var idx;
    var rate = 0.5;
    var groupSize = 5;
    var repeat = 10;
    var trig = Dust.ar(50);
    var desiredDur = 0.001; // 1 ms
    var desiredRMS = 0.2;

    WavesetAnalysis.ar(wsData, SoundIn.ar(0), 1);
    idx = WavesetSelector.ar(wsData, desiredDur, desiredRMS);
    WavesetGetFeature.kr(wsData, \rms , idx).poll(label:"rms");
    WavesetGetFeature.kr(wsData, \dur, idx).poll(label:"dur");
    WavesetSynthTriggered.ar(wsData, trig, idx, rate, groupSize, repeat);

}.play;
)
```

## 5.3 Waveset-Analyse

Die Klasse *WsAnalysis* wird die Waveset-Analyse realisiert wie im Zustandsdiagramm 3.5 auf Seite 34 definiert. Alle eingehenden Audiosamples werden direkt an eine *WavesetBuilder*-Instanz weitergegeben und die Nulldurchgänge durch Vergleich des jeweils vorhergehenden Audiosamples mit dem aktuellen Sample. Wird ein Nulldurchgang erkannt, wird die Länge des aktuellen Wavesets geprüft. Erreicht sie die Mindestlänge wird das Waveset abgeschlossen und das nächste begonnen.

Im *WavesetBuilderDualBuf* werden die Audiosamples ohne Zwischenspeicherung direkt in den Audiopuffer geschrieben. Zum Abschluss eines Wavesets werden die Merkmale berechnet und das Waveset im Waveset-Puffer abgelegt.

**Listing 5.6:** Festlegung von Wavesets mit dem WavesetBuilder bei der Analyse. Im Falle eines negativ-positiv-Nulldurchgangs wird das potentielle Waveset auf Mindestlänge geprüft und gegebenenfalls gespeichert. (ganze Klasse siehe Listing A.4 im Anhang)

```
void WsAnalysis::nextInputSample(float audioIn, int minWSLen)
{
    this->wsBuilder->addAudio(audioIn);

    // look for a -/+ zero crossing
    if(this->lastIn <= 0.0 && audioIn > 0.0)
    {
        // get WS Length
        int len = wsBuilder->getRunningLegnth();

        // Perform actions depending on WS Length
        if(len>=minWSLen)
        {
            // long enough, save it!
            wsBuilder->saveAndStartNew();
        }
        else if(len<0)
        {
            // no WS started so far, start a new one!
            wsBuilder->startNewWaveset();
        }
    }

    this->lastIn = audioIn;
}
```

## 5.4 Waveset-Synthese

Für die Synthese mit einem kontinuierlichen Strom von Wavesets wird in der Klasse *SynthContinuous* eine Instanz der Klasse *WsPlayer* eingesetzt, die zur nahtlos aneinander gereihten Wiedergabe der Wavesets genutzt wird.

Für die parallele Wiedergabe von Wavesets wird in der Klasse *SynthParallel* eine Liste fester Länge von WsPlayer-Instanzen angelegt. Es wurde bewusst auf eine dynamische Speicherreservierung verzichtet und die Anzahl der Instanzen und damit die Anzahl paralleler Wiedergaben mit einer Konstanten (derzeit 512) festgelegt. Für den Start einer neuen Wiedergabe wird die Liste durchlaufen und der erste freie Player mit den gegebenen Wiedergabeparametern initialisiert. Für die Ausführung der Wiedergabe wird die Liste ebenfalls durchlaufen und das Ausgabesignal aller aktiven Player summiert. Zunächst wurde die Wiedergabe ohne Optimierung mit dem Durchlaufen der gesamten Liste bei jedem Ausgabesample implementiert. Später wurde die Performance verbessert, indem der Index des letzten aktive Player gespeichert wird und bei der Wiedergabe nur bis zu dieser Stelle über die Liste iteriert wird (siehe Listing 5.7). Abläufe, die in beiden Varianten der Synthese vorkommen wurden in eine Basisklasse *Synth* ausgelagert.

In der Klasse WsPlayer wird die Wiedergabelogik mit Wiederholungen und Wiedergabegeschwindigkeit realisiert. Die Anzahl der Wiederholungen wird als einfacher Zähler (*int*) realisiert und die unterschiedlichen Wiedergabegeschwindigkeiten mit einer als Gleitkommazahl (*double*) gespeicherten Position innerhalb des wiederzugebenen Audiostücks realisiert. Die Position wird mit null initialisiert und nach jedem Ausgabesample um die gegebene Wiedergaberate erhöht. Über die Schnittstelle *AudioPiece* können mit der Position als Gleitkommazahl interpolierte Samplewerte abgefragt werden. Erreicht die Position das Ende des Audiostücks wird es zurück auf 0 gesetzt und der Zähler für Wiederholungen dekrementiert. Erreicht letzterer null ist die Wiedergabe beendet.

Zu einer Ausnahmesituation kann es kommen, wenn die Audiodaten eines Wavesets überschrieben werden, während dieses wiedergegeben wird. In diesem Fall tritt beim Abfragen des nächsten Audiosamples ein Fehler auf, woraufhin die Wiedergabe gestoppt wird. Dem Benutzer wird eine entsprechende Warnung ausgegeben. Diese Situation sollte nur dann auftreten, wenn der Audio-Puffer im Verhältnis kleiner ist als der Waveset-Puffer.

**Listing 5.7:** Erzeugung des Ausgabesignals für die parallele Synthese. Für jeden parallelen Wiedergabevorgang gibt es eine *WsPlayer*-Instanz, angeordnet in einer Liste. Es wird über alle aktiven *WsPlayer* iteriert und deren Ausgabesignale summiert. Für die gesamte Klasse siehe Listing A.6 im Anhang.

```
float SynthParallel::getNextOutput()
{
        // Play Wavesets from Iterators
        float outSum = 0.0;
        int lastPlayedIterator=-1;
        for(int playIdx=0;playIdx<=lastActiveIteratorIdx;playIdx++)
        {
            WsPlayer* player = &this->wsPlayers[playIdx];
            try
            {
                // play parallel Wavesets from Iterators
                if(!player->endOfPlay())
                {
                    outSum += player->nextSample();
                    lastPlayedIterator = playIdx;
                }
            }
            catch(...)
            {
                printf("Waveset␣playback␣failed!␣(unknown␣exception)\n");
                *player = WsPlayer(); // stop playback by resetting
            }
        }

        this->lastActiveIteratorIdx = lastPlayedIterator;


        return outSum;
}
```

## 5.5 Interpolation

Für die zeitliche Skalierung von Wavesets bzw. Gruppen muss die Wellenform interpoliert werden (Anforderung F2.4). Diese wird wie im Entwurf festgelegt über die Methode *getSample(double pos)* der Schnittstelle *AudioPiece* angeboten und von der Klasse *AudioBpieceDualBuf* realisiert. Die Position als Gleitkommazahl beschreibt dabei die Position in Samples des Originalsignals. Liegt die Position zwischen zwei Samples (d.h. nicht ganzzahlig) wird eine Interpolation durchgeführt.

Um festzustellen ob eine Interpolation erforderlich ist, wurde für die Abweichung der Position vom nächsten Abtastwert ein Schwellwert von 1% festgelegt. Wird dieser überschritten wird je nachdem, wie viele umgebende Werte vorhanden sind, eine einfache lineare oder die bessere kubische Spline-Interpolation [15, S. 43] eingesetzt. Sind nur zwei Samplewerte (ein vorhergehendes, ein nachfolgendes) verfügbar, ist nur eine lineare Interpolation möglich, was einer einfachen Verbindung der beiden Stützpunkte durch eine Gerade entspricht. Das Ergebnis ist stetig aber nicht stetig differenzierbar. Sind vier Samplewerte (ein vorgehendes, zwei nachfolgende) kommt eine kubische Interpolation [15, S. 43] zum Einsatz, die bessere Ergebnisse liefert. Sie ist zweimal stetig differenzierbar und erfüllt eine Minimalbedingung für die zweite Ableitung. Für die Berechnung der Interpolation wird auf die in SuperCollider vorhandene Implementierung mit den Funktionen *cubicinterp* und *lininterp* (SC_SndBuf.h) zurückgegriffen.

Bei der Implementierung stellt sich die Frage, ob für die Interpolierung die umliegenden Samples des Ausgangsmaterials oder die im Abspielvorgang relevant sind. Diese Frage lässt sich nicht allgemein beantworten, sondern hängt von der Sichtweise ab. Werden einzelne Wavesets bzw. Gruppen skaliert und dann angeordnet oder zuerst angeordnet und dann skaliert? Einen Unterschied stellt dies nur an den Grenzen zwischen Wavesets bzw. Gruppen dar und ist vermutlich auditiv kaum wahrnehmbar. Aufgrund der einfacheren Implementierung wurde entschieden, die Skalierung der Wavesets isoliert zu betrachten und die umgebenden Samples aus dem Ausgangsmaterial zu verwenden. Dies äußert sich auch im Softwareentwurf, da die Interpolation im Datenspeicher und nicht im Syntheseprozess integriert ist.

## 5.6 Waveset-Auswahl

In der Klasse *WsSelector* wird die Auswahl eines Wavesets anhand von Sollwerten für Merkmale umgesetzt. Der Bereich betrachteter Wavesets kann dabei auf die letzten $n$ begrenzt werden. Für die Auswahl wird in einer Schleife über alle zu betrachteten Wavesets iteriert wird und für jedes die mittlere quadratische Abweichung der Merkmale berechnet. Die kleinste Abweichung und der Index des zugehörigen Wavesets wird als Ergebnis gespeichert. Zur Optimierung der Performance wird bei der nächsten Auswahl die Suche fortgesetzt, sofern sich die Suchparameter nicht verändert haben und das letzte Ergebnis nicht außerhalb des betrachteten Bereichs liegt. In diesem Fall wird die Suche nur auf die neu hinzugekommenen Wavesets erweitert.

## 5.7 Waveset-Datenspeicher

Im Paket *WavesetStorage* wird die Speicherung von Wavesets mittels zweier Ringpuffer realisiert (siehe UML-Diagramm Abb. 4.7 auf Seite 47). Die beiden Ringpuffer werden in der Klasse *WsDataDualBuf* zusammengefasst und verwaltet. Ein Puffer für die Audiodaten, der andere für die Postionen der Wavesets mit Metadaten. Wie im Entwurf festgelegt, werden diese mit einer Template-Klasse realisiert. Die beiden Puffer werden bei der Initialisierung des Waveset-Speichers in den angegebenen SuperCollider-Puffern angelegt. Dafür wird jeweils über die gegebene Puffernummer ein Zeiger auf den Datenbereich sowie dessen Größe ermittelt. Innerhalb dieses Speicherbereichs erfolgt die folgende Aufteilung:

- Byte 1-4: magische Zahl (float), welche die Initialisierung des Ringpuffers markiert
- Byte 5-24: Instanz der Ringpuffer-Klasse
- restlicher Bereich ab Byte 25: Datenbereich des Ringpuffers

Aufgrund dieser speziellen Art der Speicherverwaltung wurde entschieden, auf keine bestehende Ringpuffer-Implementierung zurückzugreifen, sondern speziell für diese Anwendung einen eigenen Ringpuffer zu realisieren. Damit liegt die volle Kontrolle über Speicherverbrauch und Laufzeitverhalten beim Entwickler. Der Ringpuffer

besteht aus einer einfachen Steuerstruktur mit vier Variablen: Einem Zeiger auf den Datenbereich, der Größe des Datenbereichs, sowie der Position des ersten und letzten Elements als eindeutiger Index. Als Datentyp für die Position wurde als adäquater Datentyp ein 32-Bit integer gewählt, damit ist die Verwaltung von mehr als 13 Stunden Audiomaterial bei der üblichen Samplerate von 44,1 kHz möglich. Zum Schreiben und Lesen von Daten wird der eindeutige Index nach einer Überprüfung des gültigen Datenbereichs mit einer einfachen Modulo-Operation auf den Datenbereich des Ringpuffers projiziert.

# 6 Anwendung in Live-Performances

Die implementierte Echtzeit-Resynthese wurde von Till Bovermann[1] in zwei Setups für Live-Performances eingesetzt und überprüft, inwiefern sie sich hierfür eignet und welche Möglichkeiten sich dem Künstler bieten.

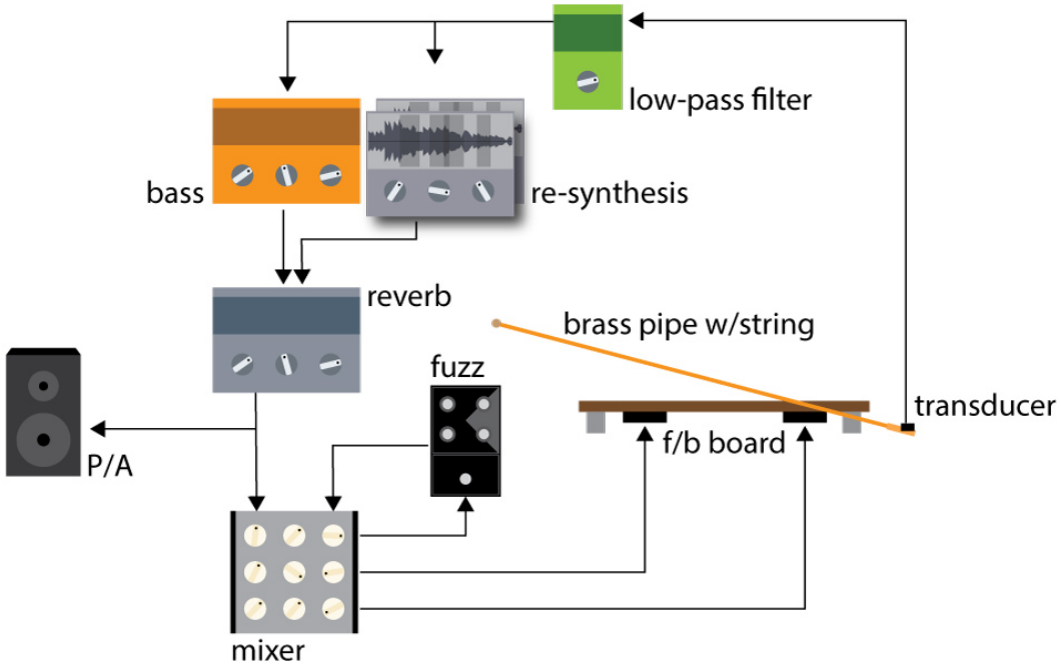


**Abbildung 6.1:** Komponenten und Signalfluss des *Half-Closed-Loop*-Setups [50]. Zentrales Element ist ein individuelles Instrument bestehend aus einem Messingrohr mit einer gespannten Saite im Innern.

**Half-Closed-Loop** ist ein Instrument zur geschichtenerzählenden Improvisation (engl. *storytelling improvisation*) [50]. Es basiert auf Rückkopplungen, in unterschiedlichen Ebenen gesteuert durch einen Mixer. Zentrale Komponente ist ein individuelles Instrument aus einem Messingrohr mit einer gespannten Saite im Innern und ein Holzbrett mit montiertem Körperschallwandler. Ist das Messingrohr in Kontakt mit

[1] `http://tai-studio.org`

dem Brett, wird es zusammen mit der Waveset-Resynthese Teil einer Feedbackschleife (siehe Abb. 6.1). Die durch die Rückkopplung auftretenden Instabilitäten unterstützen die Improvisation als Kommunikation zwischen KünstlerIn und Instrument.

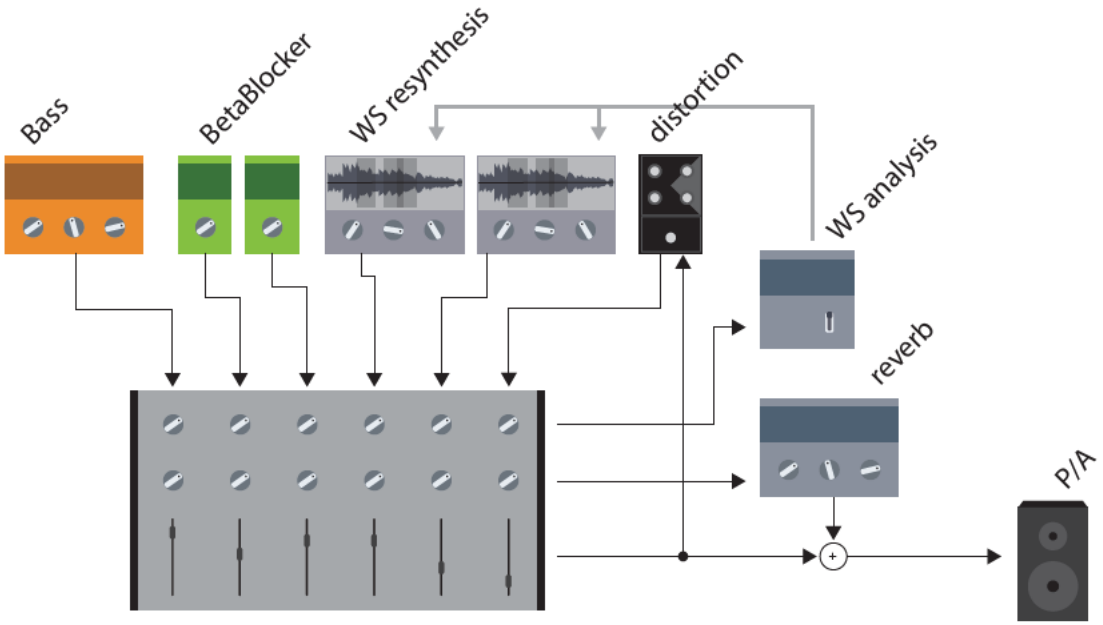


**Abbildung 6.2:** Komponenten und Signalfluss des *BBWS*-Setups [49]. Mit einem Bass-Synthesizer und *BetaBlocker*-UGens wird ein Basis-Signal erzeugt, das mittels Waveset-Resynthese und anderen Effekten modifiziert wird.

**BBWS** ist ein Setup in SuperCollider für eine Live-Performance[2] mit *Chip Interpretations*[3]) und einer Waveset-Resynthese [49]. Zwei *BetaBlocker*-UGens [8] erzeugen mittels Chip Interpretations zusammen mit einem Bass-Synthesizer ein Grundsignal (siehe Abb. 6.2). Dieses wird über zwei parallele Waveset-Resynthesen, einem Verzerrungsfilter und einem Hall-Effekt modifiziert. Mit dem (virtuellen) Mixer wird das Signal für die Resynthese, Hall-Effekt und Master-Ausgang frei aus den Eingängen zusammengestellt. Je nach Abmischung entsteht eine mehr oder weniger starke Rückkopplung über die Resynthese. Für die Resynthese wird das jeweils

[2]Eine Audioaufzeichnung ist unter `https://archive.org/details/BBWSLiveAtSpektrumBerlin` verfügbar.

[3]Bei *Chip Interpretations* wird ein Audiosignal von einer virtuellen Rechenheinheit (CPU) erzeugt. Diese interpretiert beliebige Eingabedaten als eine Sequenz von definierten Befehlen (8-Bit Elemente) und interpretiert diese als Programm. Ergebnis ist ein vom Programm abhängiges, kontinuierliches Audiosignal. (mehr dazu in [8]

letzte Waveset mit den vorhergehenden 1 bis 40 Wavesets zu einer Gruppe vereint. Diese werden mit der ereignisbasierten Synthese gleichmäßig angeordnet. Die Gruppe wird je nach Einstellung 1-10 mal wiederholt bei einer Dichte von 0.5 bis 100Hz Wiedergabevorgängen pro Sekunde. Die Wiedergabegeschwindigkeit lässt sich aus einer vorgegebenen Menge von Werten (0.125, 0.25, 0.5, 0.75, 1, 2) auswählen. Das eingesetzte SC-Programm für die Waveset-Resynthese ist im Anhang A.1 zu finden.

Allgemein lassen sich mit der Waveset-Resynthese neue Strukturen schaffen, die auf dem Ausgangsmaterial basieren, z.B. metrische Rhythmen mit einer synchronen Synthese oder statische Töne mit einer kontinuierlichen Synthese. Grundsätzlich haben die Wavesets einen charakteristischen „digitalen“ Klang, der dennoch gewisse Klangfarben des Ausgangsmaterials aufweist. Letzteres überrascht nicht, da Wavesets aus der gleichen Wellenform bestehen und die Resynthese als eine Restrukturierung des Ausgangsmaterials betrachtet werden kann. In der Regel bleibt bei längeren Waveset-Gruppen mehr von der Klangfarbe des Ausgangsmaterials erhalten, während bei kürzeren Gruppen der typische „künstliche“ Klang der Wavesets stärker in den Vordergrund tritt. Im *Half-Closed-Loop*-Setup bieten die Wavesets mit ihrem digitalen Klang einen sehr interessanten Kontrast zu den ansonsten organischen Klängen.

Die Wiedergabegeschwindigkeit bei der Synthese lässt sich nutzen um Ober- und Untertöne zu erzeugen. Entspricht die Geschwindigkeit einem Vielfachen (2, 3, 4, ...) entstehen harmonische Obertöne, ist sie ein Bruchteil ($\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, ...) harmonische Untertöne. Durch die Rückkopplung im BBWS-Setup in Kombination mit unterschiedlichen Abspielgeschwindigkeiten lassen sich Ober- und Untertöne mit treppenartiger Struktur erzeugen. Wird zum Beispiel die doppelte Abspielgeschwindigkeit verwendet, entstehen Obertöne doppelter Frequenz. Werden diese Wiederrum in die Resynthese geleitet, entstehen zeitlich versetzt zusätzliche Obertöne vierfacher Frequenz, achtfacher Frequenz und so weiter.

Die Länge der Wavesets bei der Analyse passt sich an die Beschaffenheit des Ausgangsmaterials an. Dies kann im BBWS-Setup aktiv genutzt werden, um die Resynthese zu beeinflussen. Mit dem Bass-Synthesizer kann die dominante Grundfrequenz bestimmt werden, die sich dann direkt auf die Waveset-Länge auswirkt.

# 7 Zusammenfassung und Ausblick

## Ergebnisse

In dieser Arbeit wurde eine SuperCollider-Erweiterung geschaffen, die eine flexibel einsetzbare Waveset-Resynthese für Live-Anwendungen bietet. Im Gegensatz zu bisherigen Waveset-Implementierungen wird sowohl die Analyse als auch die Synthese in Echtzeit ausgeführt. Bei der Synthese werden zwei grundlegende Typen unterstützt. Zum einen die kontinuierliche Synthese, welche Wavesets nahtlos aneinander reiht. Zum anderen die ereignis-basierte Synthese, welche eine freie Anordnung, gesteuert durch einen Trigger, erlaubt. Letztere unterstützt die parallele Wiedergabe bei zeitlich überlappenden Wavesets. Beide Varianten können über die Parameter Wiedergabegeschwindigkeit, Gruppengröße sowie Anzahl der Wiederholungen beeinflusst werden. In der Praxis zeigte sich die Flexibilität der ereignisbasierten Synthese von Vorteil. Sie ermöglicht ebenso synchrone wie asynchrone Anordnungen. Über die Wiedergabegeschwindigkeit lassen sich Ober- und Untertöne erzeugen, was insbesondere in Kombination mit einer Rückkopplung interessante Ergebnisse liefert. Allgemein hat sich gezeigt, dass sich Synthesemethoden, die für aufgezeichnetes Material entwickelt wurden, oft nicht direkt in eine Live-Resynthese übertragen lassen. Die Live-Resynthese ist hier begrenzt durch ihre Vorwärtsbewegung in der Zeit (vgl. [35, S. 191]) mit einer fortlaufend veränderlichen Menge verfügbarer Wavesets. Die Wavesets der Zukunft sind unbekannt und die der Vergangenheit nur begrenzt verfügbar. Für die Auswahl von Wavesets wurde ein automatisiertes Verfahren anhand von berechneten Merkmalen entwickelt. Dabei werden Sollwerte für die Merkmale angegeben und das Waveset mit der kleinsten Abweichung ausgewählt. Der Zahl der betrachteten Wavesets der Vergangenheit kann dabei begrenzt werden. Durch die modularen Synthesebausteine kann auf die Auswahl individuell Einfluss genommen werden, etwa mit mit stochastischen Zufallswerten, falls ein weniger statisches Ergebnis erwünscht ist.

Für die Umsetzung wurden die Anforderungen schrittweise erfasst, analysiert und in Zwischenversionen implementiert. Die dabei eingesetzten objektorientierten Methoden bei der Anforderungsanalyse und dem Softwareentwurf begünstigten eine gutes,

objektorientiertes Design der Software. Die Schnittstelle für die Implementierung von Erweiterungen in SuperCollider ist einfach gehalten, aber ihre Struktur entspricht nicht dem, was heute unter einem guten Softwaredesign verstanden wird. Mit etwas Zusatzaufwand konnte die Schnittstelle aber auf eine moderne, objektorientierte Anwendung mit dynamischer Speicherverwaltung adaptiert werden.

## Limitierungen

Die derzeitige Implementierung unterliegt gewissen Limitierungen, welche die Einsatzmöglichkeiten einschränken können. Etwa bei der Auswahl von Wavesets basierend auf Merkmalen. In Kombination mit längeren Gruppen bei der Synthese ergibt sich eine geringere Aussagekraft der Merkmale. Wird ein Waveset anhand von Merkmalen ausgewählt und bei der Synthese mit den umgebenden Wavesets zu einer Gruppe zusammengefasst, kann die Gruppe als Ganzes beliebig von den Auswahlkriterien abweichen. Abhilfe würde eine Berücksichtigung der Gruppen sowohl bei der Auswahl als auch der Merkmalsberechnung schaffen. Eine weitere Einschränkung stellt die geringe Anzahl berechneter Merkmale dar. Zusätzliche Merkmale könnten das Potential der Auswahl noch deutlich erweitern. Insbesondere Merkmale im Frequenzbereich wie der spektrale Schwerpunk oder eine Tonhöhenerkennung. Diese lassen sich aber nur sinnvoll in Verbindung mit der genannten Merkmalsberechnung für Gruppen einsetzen, da einzelne meist sehr kurze Wavesets nicht für die Berechnung eines aussagekräftigen Fequenzspektrums ausreichen.

Bei der Implementierung hat der Einsatz von abstrakten Schnittstellen in C++ ohne Zweifel den Vorteil geringerer Abhängigkeiten. Allerdings bringt die Realsierung der Schnittstellen als abstrakte Klassen gewisse Performanceinbußen mit sich, die sich bei der Verarbeitung von Audiodaten mit hohen Sampleraten durchaus bemerkbar machen können. Eine blockweise Verarbeitung, auch intern, könnte diesen Faktor verringern. Bei der Verwaltung der Waveset-Daten hat das derzeit eingesetzte zweigliedrige Speicherkonzept gewisse Nachteile. Die beiden Puffer müssen in der Größe aufeinander abgestimmt werden und je nach Waveset-Länge kann die genaue Anzahl gespeicherter Wavesets schwanken. Hier sollte geprüft werden, ob die Umstellung auf dynamischen Speicher mit einer festen Anzahl vorgehaltener Wavesets sinnvoll ist.

## Ausblick

Für eine zukünftige Weiterentwicklung ergeben sich bereits einige Ansätze aus den diskutierten Limitierungen der derzeitigen Implementierung. Es gibt auch auch mehrere neue Funktionen, die einen Mehrwert bieten würden.

Um die räumliche Darstellung von Mehrkanalsystemen zu nutzen, kann das Summensignal der Synthese mittels Panning nachträglich auf die Kanäle verteilt werden. Bei der ereignisbasierten Synthese ergibt sich hier aber die Einschränkung, dass parallele Wiedergabevorgänge nicht einzeln positioniert werden können. Hierzu wäre eine Erweiterung der Synthese sinnvoll, die das Panning der Teilsignale intern vornimmt und ein Mehrkanalsignal ausgibt.

Die Wavesets haben einen charakteristischen, künstlichen bzw. digital empfundenen Klang. Bei Anwendungen in denen dies nicht erwünscht ist, könnte eine optionale Modellierung der Wavesets mit einer Hüllkurve den Klang beeinflussen. Damit würde die Waveset-Synthese näher an eine konventionelle Granularsynthese heranrücken und ein breiteres Anwendungsspektrum abdecken.

Es ist geplant, die RTWavesets-Erweiterung quelloffen für die SuperCollider-Gemeinschaft zur Verfügung zu stellen. Für die Zukunft ist auch eine Portierung auf andere Plattformen denkbar. Eine neuere Entwicklung im Bereich modularer Audioanwendungen sind Smartphone bzw. Tablet Apps, die sich inzwischen auch untereinander verbinden lassen. Etabliert ist hier die App *Audiobus*[1], welche unter Apples iOS eine Verbindung parallel laufenden Audio-Apps in Echtzeit ermöglicht. Aus dem schnell wachsenden Repertoire unterstützter Audio-Apps können einfache Patches aus Signalgeneratoren, Filtereffekten und Samplern zusammengesetzt werden. In der Flexibilität sind solche Systeme zwar nicht mit SuperCollider vergleichbar, aber dafür bieten sie einen einfachen Einstieg und so könnte die Waveset-Resynthese eine breitere Zielgruppe erreichen.

[1] `https://audiob.us`

# Quellenverzeichnis

## Literatur


[1] Harold Abelson und Gerald Jay Sussman. *Structure and interpretation of computer programs*. 1983.

[2] Martin Atkins et al. *The Composers' Desktop Project*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 1987.

[3] Helmut Balzert. *Lehrbuch der Software-Technik. [2]. Softwaremanagement*. ger. 2. Aufl. Heidelberg ua: Spektrum Akademischer Verlag, 2008. ISBN: 978-3-8274-1161-7.

[4] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. de. Heidelberg: Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-1705-3 978-3-8274-2247-7.

[5] Helmut Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. de. Heidelberg: Spektrum Akademischer Verlag, 2011. ISBN: 978-3-8274-1706-0 978-3-8274-2246-0.

[6] Ross Bencina. „Implementing real-time granular synthesis“. In: *Audio Anecdotes III*. A K Peters, 2006. ISBN: 978-1-56881-215-1.

[7] Tim Blechmann. „Supernova, a Multiprocessor-Aware Synthesis Server for Supercollider“. In: *Proceedings of the Linux Audio Conference*. 2005, S. 141–146.

[8] Till Bovermann und Dave Griffiths. „Computation As Material in Live Coding“. In: *Computer Music Journal* 38.1 (2014), S. 40–53. ISSN: 0148-9267. DOI: 10.1162/COMJ_a_00228.

[9] Till Bovermann et al. „3DMIN—Challenges and Interventions in Design, Development and Dissemination of New Musical Instruments“. In: *Proceedings of the 40th International Computer Music Conference (ICMC)*. 2014, S. 1–5.

[10] Gilbert Brands. *Das C++ Kompendium: STL, Objektfabriken, Exceptions*. ger. 2. Aufl. Berlin: Springer, 2010. ISBN: 978-3-642-04786-2.

[11] William H. Brown, Raphael C. Malveau und Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. 1998.

[12] A. de Campo. „Microsound“. In: *The SuperCollider Book*. The MIT Press, 2011.

[13] Sergio Cavaliere und Aldo Piccialli. „Granular synthesis of musical signals“. In: *Musical Signal Processing* (1997), S. 155–186.

[14] Floris Cohen. *Die zweite Erschaffung der Welt: Wie die moderne Naturwissenschaft entstand*. de. Campus Verlag, Aug. 2010. ISBN: 978-3-593-39134-2.

[15] Carl De Boor. *A practical guide to splines*. Bd. 27. Springer-Verlag New York, 1978.

[16] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[17] Dennis Gabor. „Theory of communication. Part 1: The analysis of information“. In: *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering* 93.26 (1946), S. 429–441.

[18] Dennis Gabor. „Acoustical quanta and the theory of hearing“. In: *Nature* 159.4044 (1947), S. 591–594.

[19] Dietmar Herrmann. *Effektiv Programmieren in C und C++: Eine aktuelle Einführung mit Beispielen aus Mathematik, Naturwissenschaft und Technik*. de. Springer-Verlag, März 2013. ISBN: 978-3-322-94365-1.

[20] Amelie Hinrichsen et al. „PushPull. Reflections on Building a Musical Instrument Prototype“. In: *Proceedings of ICLI 2014*. 2014.

[21] Ralf Hinze. „List-Comprehensions“. de. In: *Einführung in die funktionale Programmierung mit Miranda*. Vieweg+Teubner Verlag, 1992, S. 123–137. ISBN: 978-3-519-02287-9 978-3-322-93090-3.

[22] Olaf Hochherz. „SPList, a Waveset synthesis library and its usage in the composition"draussen“. In: *Proceedings of Linux Audio Conference 2008*. 2008.

[23] IEEE. „Systems and software engineering – Life cycle processes –Requirements engineering“. In: *ISO/IEC/IEEE 29148:2011(E)* (2011), S. 1–94. DOI: `10.1109/IEEESTD.2011.6146379`.

[24] Donald Ervin Knuth. *The art of computer programming. 1. Fundamental algorithms*. eng. 3. ed. Upper Saddle River, NJ ; Munich ua: Addison-Wesley, 1997. ISBN: 978-0-201-89683-1.

[25] Terry Alan Lee. *GranCloud - A New SuperCollider Class for Real-time Granular Synthesis*. University of North Texas, 2009.

[26] Terry Alan Lee. *Grancloud: A Real-Time Granular Synthesis Application and Its Implementation in the Interactive Composition Creo*. University of North Texas, 2009.

[27] Alexander Lerch. *An introduction to audio content analysis: applications in signal processing and music informatics*. eng. Piscataway, NJ: IEEE ua, 2012. ISBN: 978-1-118-26682-3.

[28] James McCartney. „SuperCollider: a new real time synthesis language“. In: *Proceedings of the International Computer Music Conference*. 1996.

[29] James McCartney. „Rethinking the computer music language: SuperCollider“. In: *Computer Music Journal* 26.4 (2002), S. 61–68.

[30] Timothy Opie. „Granular synthesis: Experiments in live performance“. In: *Proceedings of the Australasian Computer Music Conference 2002*. 2002, S. 97–102.

[31] Timothy Tristram Opie. *Creation of a real-time granular synthesis instrument for live performance*. Queensland University of Technology, 2003.

[32] Klaus Pohl. *Requirements engineering: Grundlagen, Prinzipien, Techniken*. ger. 2., korrigierte Aufl. Heidelberg: dpunkt-verl, 2008. ISBN: 978-3-89864-550-8.

[33] Curtis Roads. „Automated granular synthesis of sound“. In: *Computer Music Journal* (1978), S. 61–62.

[34] Curtis Roads. *The computer music tutorial*. MIT press, 1996.

[35] Curtis Roads. *Microsound*. MIT press, 2004.

[36] Suzanne Robertson und James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012.

[37] Mayank Sanganeria und Kurt James Werner. „GrainProc: a real-time granular synthesis interface for live performance“. In: *Proceedings of the International Conference on New Interfaces for Musical Expression 2013*. 2013.

[38] M. Schönfinkel. „Über die Bausteine der mathematischen Logik“. deu. In: *Mathematische Annalen* 92.3 (1924), S. 305–316. ISSN: 0025-5831. DOI: 10.1007/BF01448013.

[39] Ian Sommerville. *Software engineering*. eng. 9. ed., Internat. ed. Boston, Massua: Pearson, 2011. ISBN: 978-0-13-705346-9.

[40] D. Stowell. „Writing Unit Generator Plugins“. In: *The SuperCollider Book*. Hrsg. von S. Wilson, D. Cottle und N. Collins. Cambridge, MA: MIT Press, 2011.

[41] Bjarne Stroustrup. *Die C++ -Programmiersprache: aktuell zum C++ 11-Standard. The C++ programming language <dt.>* ger. München: Hanser, 2015. ISBN: 978-3-446-43961-0.

[42] Barry Truax. *Real-time granular synthesis with the DMX-1000*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 1986.

[43] Barry Truax. *Real-time granulation of sampled sound with the DMX-1000*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 1987.

[44] Scott Wilson, David Cottle und Nick Collins. *The SuperCollider Book*. The MIT Press, 2011.

[45] Trevor Wishart. *Audible design: a plain and easy introduction to practical sound composition*. Orpheus the Pantomime, 1994.

[46] Matthew Wright. „Open Sound Control: an enabling technology for musical networking“. In: *Organised Sound* 10.03 (2005), S. 193–200.

[47] Iannis Xenakis. *Formalized music*. en. Indiana University Press, 1971. ISBN: 978-0-253-32378-1.

## Online-Quellen

[48] *3DMIN Website*. URL: `http://www.3dmin.org/` (besucht am 23. 01. 2016).

[49] Till Bovermann. *BBWS | TAI Studio*. URL: `http://tai-studio.org/index.php/projects/instrument-making/chipinterpretations/bbws/` (besucht am 16. 03. 2016).

[50] Till Bovermann. *Half-closed loop | 3DMIN*. 2016. URL: `http://www.3dmin.org/research/open-development-and-design/half-closed-loop/` (besucht am 15. 03. 2016).

[51] *sc3-plugins - GitHub*. URL: `https://github.com/supercollider/sc3-plugins` (besucht am 05. 01. 2016).

[52] *SuerpCollider Website: scsynth*. URL: `http://supercollider.github.io/development/scsynth-source-guide` (besucht am 19. 03. 2016).

[53] *SuperCollider Documentation: GrainIn*. URL: `http://doc.sccode.org/Classes/GrainIn.html` (besucht am 25. 07. 2015).

[54] *SuperCollider Documentation: Impulse*. URL: `http://doc.sccode.org/Classes/Impulse.html` (besucht am 04. 02. 2016).

[55] *SuperCollider Documentation: Using Extensions*. URL: `http://doc.sccode.org/Guides/UsingExtensions.html` (besucht am 11. 02. 2016).

[56] *SuperCollider Documentation: Writing Classes*. URL: `http://doc.sccode.org/Guides/WritingClasses.html` (besucht am 05. 01. 2016).

[57] *SuperCollider Documentation: Writing Unit Geneartors*. URL: `http://doc.sccode.org/Guides/WritingUGens.html` (besucht am 26. 01. 2016).

[58] *SuperCollider Mailing List: Does anybody have GranCloud, developed by Terry A. Lee?* URL: `http://new-supercollider-mailing-lists-forums-use-these.2681727.n2.nabble.com/Does-anybody-have-GranCloud-developed-by-Terry-A-Lee-td7579201.html#a7579324` (besucht am 10. 03. 2016).

[59] *SuperCollider Mailing List: overloading new and delete in plug-ins.* URL: http://new-supercollider-mailing-lists-forums-use-these.2681727.n2.nabble.com/overloading-new-and-delete-in-plug-ins-td4833499.html#a4834559 (besucht am 26. 02. 2016).

[60] *Supercollider Quarks: Wavesets.* URL: https://github.com/supercollider-quarks/Wavesets (besucht am 28. 04. 2015).

[61] *SuperCollider Website.* URL: http://supercollider.github.io/ (besucht am 08. 10. 2015).

[62] Trevor Wishart. *Computer Sound Transformation.* 2000. URL: http://www.trevorwishart.co.uk/transformation.html (besucht am 28. 10. 2015).

# Glossar

**Analyse (Granularanalyse)**

Extrahierung von Grains aus einem Signal (=Granulierung).

**Asynchrone Synthese**

Synthese, bei der die Grains nicht linear angeordnet, sondern ungleichmäßig gestreut werden (siehe 2.1.2).

**Audio-Rate**

Signale bzw. Signalverarbeitungsprozesse in SuperCollider, die im Takt der Audio-Abtastrate berechnet werden, z.B. 44,1 kHz (siehe 2.3.3).

**Control-Rate**

Signale bzw. Signalverarbeitungsprozesse für Steueraufgaben in SuperCollider, die mit einem Bruchteil der Audio-Abtastrate berechnet werden (siehe 2.3.3).

**Demand-Rate**

Signale bzw. Signalverarbeitungsprozesse in SuperCollider, die bei Bedarf berechnet werden (siehe 2.3.3).

**Grain**

Sehr kurzes (digitales) Klangfragment (englisch für Korn oder Quäntchen).

**Granulierung**

Extrahierung von Grains aus einem Signal (=Analyse).

**Objektorientierte Analyse (OOA)**

Objektorientierte Variante der Anforderungsanalyse. Ergebnis der OOA ist ein abstraktes Produktmodell in Form eines objektorientierten Analyse-Modells (siehe 2.4.2).

**Objektorientiertes Design (OOD)**

Objektorientierte Variante des Systementwurfs. Das bei der Anforderungsanalyse erstellte Produktmodell wird weiterentwickelt zu einem Systementwurf (siehe 2.4.3).

**Resynthese**

Zerlegung eines Signals in Grains (Analyse) und erneutes Zusammensetzen zu einem neuen Signal (Synthese).

**SCLang**

Programmiersprache von SuperCollider bzw. dessen Interpreter (siehe 2.3.3).

**SCSynth**

Syntheseserver von SuperCollider (siehe 2.3.2).

**SuperCollider**

Eine Software und Programmiersprache für Echtzeit-Klangsynthese und algorithmische Komposition (siehe 2.3).

**Synchrone Synthese**

Synthese, bei der die Grains in gleichmäßigen Abständen angeordnet werden (siehe 2.1.2).

**Synthese (Granularsynthese)**

Zusammensetzen eines neuen Signals aus kleinen Fragmenten, Grains.

**Transformation (Waveset-Transformation)**

Bearbeitung eines Signals durch Neuanordnung und/oder Modifikation von Wavesets (siehe 2.2).

**UGen (Unit Generator)**

Synthesebausteine, aus denen Signalverarbeitungsprozesse in SuperCollider zusammengesetzt werden können (siehe 2.3.3).

**Unified Modeling Language (UML)**

Grafische Modellierungssprache zur Spezifikation und Dokumentation von Software in Diagrammen.

**Waveset**

Teil eines Audiosignals zwischen einem negativ-positiv Nulldurchgang und dem nächsten (siehe 2.2). Kann als eine spezielle Art der Grains verstanden werden.

# A Quellcode

## A.1 Supercollider

**Listing A.1:** Definition der RTWaveset-UGens in SuperCollider mit Ein- und Ausgabesignalen.

```
WavesetData {
    var <audioBufSize, <wsBufSize;
    var <server;
    var <audioBuf, <wsBuf;

    *new{ arg audioBufSize=1000000, wsBufSize=10000, server(Server.default);
        ^super.newCopyArgs(audioBufSize, wsBufSize, server).init
    }

    init {
        audioBuf = Buffer.alloc(server, audioBufSize);
        wsBuf = Buffer.alloc(server, wsBufSize);
    }
}

WavesetAnalysis : MultiOutUGen {
    *ar { arg wsData, in, active=1, minWSLen=0.0005;
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, in, active, minWSLen)
    }

    init { arg ... theInputs;
        inputs = theInputs;
        outputs = [
            OutputProxy(rate, this, 0),
            OutputProxy(rate, this, 1)
        ];
        ^outputs
    }
}

WavesetSelector : UGen {
    *ar { arg wsData, dur=(-1), rms=(-1), peaks=(-1),lookBackLimit=(-1);
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, dur, rms, peaks, lookBackLimit)
    }
}

WavesetSynthTriggered : UGen {
    *ar { arg wsData, trig, idx, rate=1, groupSize=1, repeat=1;
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, trig, idx, rate, groupSize, repeat)
    }
}

WavesetSynthContinuous : UGen {
```

```
    *ar { arg wsData, idx, rate=1, groupSize=1, repeat=1;
        ^this.multiNew('audio', wsData.audioBuf, wsData.wsBuf, idx, rate, groupSize, repeat)
    }
}

WavesetGetFeature : UGen {
    *kr { arg wsData, feature, idx;
        var featureID = switch(feature,\dur , 0, \rms, 1, \peaks, 2);
        ^this.multiNew('control', wsData.audioBuf, wsData.wsBuf, featureID, idx)
    }
}
```

**Listing A.2:** SC-Programm für die Waveset-Resynthese in der BBWS Performance [49].

```
// this uses RTWaveSet and MeTA
// http://3dmin-code.org/instrument-sketching/MeTA
// http://3dmin-code.org/instrument-sketching/RTWaveSets // closed source for now


// the data resource
// goes into 3_helperNdefs/wsData.scd
m.helpers[\wsData] = WavesetData(server: m.server);


// waveset recorder
// goes into 3_efx/aux/wsAux.scd
Ndef(\ws).addSpec(\active, ControlSpec(0, 1, 'linear', 1, 0));
Ndef(\ws).addSpec(\leak, ControlSpec(0.9, 1, 'exp', 0, 0));

m.aux[\ws] = Ndef(\ws, {
    var in = \in.ar(0!m.config.numChans).sum;
    in = LeakDC.ar(in, \leak.kr(0.995));
    WavesetAnalysis.ar(m.helpers[\wsData], in, \active.kr(1))//.poll
});


// goes into 3_generators/wsResynth.scd
Ndef(\ws); // return value

(
Ndef(\wsResynth).ar(2);

// specs
// useful ranges for controls
Ndef(\wsResynth).addSpec(\trig, [0.5, 100, \exp]);
Ndef(\wsResynth).addSpec(\rateIdx, [0, 5, \lin, 1]);
Ndef(\wsResynth).addSpec(\gSize, [1, 40, \lin, 1]);
Ndef(\wsResynth).addSpec(\repeat, [1, 10, \lin, 1]);
Ndef(\wsResynth).addSpec(\lpFreq, [100, 20000, \exp]);
Ndef(\wsResynth).addSpec(\amp, [0, 1]);        // amplitude
Ndef(\wsResynth).addSpec(\on, [0, 1, \lin, 1]); // on/off

Ndef(\wsResynth, {
```

```
    var snd;
    var pan;

    // slow automated panning
    pan = LPF.kr(BrownNoise.kr, 1);


    snd = WavesetSynthTriggered.ar(

        // data input
        m.helpers[\wsData],

        // regular trigger,
        // rate changed each time rChange is called from the language
        // 5 * (0.125, 0.25 .. 2) // << possible rates
        // slowly adapting from previous level within 5 seconds
        Impulse.ar(
            \trig.kr(5)
            * TRand.kr(0.125, 2, \rChange.tr(0))
              .round(0.125).lag(5)
        ),

        // just the last index
        WavesetSelector.ar(m.helpers[\wsData]),

        // playback rate fixed to a bunch of values
        // [0.125, 0.25, 0.5, 0.75, 1, 2] that are slightly off
        Select.kr(\rateIdx.kr(4), [0.125, 0.25, 0.5, 0.75, 1, 2])
        * LFNoise0.kr(0.2).range(0.95, 1.05),

        // group size between 1..40
        \gSize.kr(1),

        // repeat between 1..10
        \repeat.kr(1));

    // oh yeah, this is my personal phase-panning
    snd = m.utils[\phasePan].value(snd, pan);

    // turn-on-the-sound-mechanism
    (snd * \amp.kr(1)).tanh * \on.kr(0, 0.2);

}).play;

);


////////////// MAPPING //////////////////////

// on/off
Ndef(\wsResynth).addHalo(\offFunc, {
    Ndef(\wsResynth).set(\on, 0);
});
```

```
Ndef(\wsResynth).addHalo(\onFunc, {
    Ndef(\wsResynth).set(\on, 1);
});


// knobs used to set rate, groupSize and repeat
Ndef(\wsResynth).addHalo(\sndA, {|value|
    Ndef(\wsResynth).setUni(\rateIdx, value)
});
Ndef(\wsResynth).addHalo(\sndB, {|value|
    Ndef(\wsResynth).setUni(\gSize, value)
});
Ndef(\wsResynth).addHalo(\pan, {|value|
    Ndef(\wsResynth).setUni(\repeat, value)
});

// button triggers new arbitrary rate
Ndef(\wsResynth).addHalo(\fcs, {|value|
    Ndef(\wsResynth).set(\rChange, 1.postln)
});
```

## A.2 C++

### A.2.1 Analyse

**Listing A.3:** C++ Implementierung des Analysis-UGens. Die Eingangssignale werden entgegengenommen und der *active*-Parameter interpretiert. Ist die Analyse aktiv wird das Signal zur Verarbeitung an die Klasse *WsAnalysis* weitergegeben.

```
#include "WavesetAnalysis.h"

/**
 * @brief WavesetAnalysis_Ctor Constructor for the WavesetAnalysis UGen.
 * @param unit
 */

void WavesetAnalysis_Ctor( WavesetAnalysis *unit ) {

    WavesetBase_Ctor(unit);
    new (&unit->wsAnalysis) WsAnalysis(unit->wsData);

    // set the calculation function.
    SETCALC(WavesetAnalysis_next);

    // calculate one sample of output.
    WavesetAnalysis_next(unit, 1);
}

/**
 * @brief WavesetAnalysis_Dtor Destructor.
 * @param unit
 */

void WavesetAnalysis_Dtor( WavesetAnalysis *unit ) {
    unit->wsAnalysis.~WsAnalysis();
    WavesetBase_Dtor(unit);
}


/**
 * @brief WavesetAnalysis_next Process the next block of audio samples.
 * @param unit
 * @param inNumSamples
 */

void WavesetAnalysis_next( WavesetAnalysis *unit, int inNumSamples ) {
```

```
    float *in = IN(2);
    float *out1 = OUT(0);
    float *out2 = OUT(1);
    float analysisOn = IN0(3);
    float inMinWavesetLengthSec = IN0(4);
    int minWsLen = int (inMinWavesetLengthSec * unit->mWorld->mFullRate.mSampleRate + 0.5f
        ↪ ); // convert in samples and round

    // Input Processing
    for ( int i=0; i<inNumSamples; ++i)
    {

        if(analysisOn > 0.0) // analysis is running...
        {
            unit->wsAnalysis.nextInputSample(in[i],minWsLen);
        }
        else
        {
            unit->wsAnalysis.reset();
        }

        out1[i] = (float) unit->wsAnalysis.getFirstWsIdx();
        out2[i] = (float) unit->wsAnalysis.getLastWsIdx();
    }

}
```

**Listing A.4:** Implementierung der Klasse *WsAnalysis*. Nimmt die Eingangssignale vom Analysis-UGen entgegen und führt die Waveset-Analyse durch.

```
#ifndef WSANALYSIS_H
#define WSANALYSIS_H

#include "WsStorage.h"
#include "WavesetBuilder.h"

class WsAnalysis : public ScObject
{
private:
    WsStorage* wsData;
    WavesetBuilder* wsBuilder;
    float lastIn;

public:
    WsAnalysis(WsStorage* wsData);
    ~WsAnalysis();
```

```
    void nextInputSample(float val, int minWSLen);
    void reset();
    int getFirstWsIdx();
    int getLastWsIdx();
};

#endif // WSANALYSIS_H
```

```
#include "WsAnalysis.h"

/**
 * @brief Constructur, just set the wsData variable.
 * @param wsData
 */

WsAnalysis::WsAnalysis(WsStorage *wsData)
{
    this->wsData = wsData;
    this->wsBuilder = this->wsData->createWavesetBuilder();
    reset();
}

/**
 * @brief WsAnalysis::~WsAnalysis
 */

WsAnalysis::~WsAnalysis()
{
    delete this->wsBuilder;
}

/**
 * @brief Reset the running analysis.
 */

void WsAnalysis::reset()
{
    this->wsBuilder->startNewWaveset();
    this->lastIn = NAN;
}

/**
 * @brief receive Audio Input.
 * @param audioIn
 * @param minWSLen minimal waveset length in samples
 */
```

```
void WsAnalysis::nextInputSample(float audioIn, int minWSLen)
{
    this->wsBuilder->addAudio(audioIn);

    // look for a -/+ zero crossing
    if(this->lastIn <= 0.0 && audioIn > 0.0)
    {
        // get WS Length
        int len = wsBuilder->getRunningLegnth();

        // Perform actions depending on WS Length
        if(len>=minWSLen)
        {
            // long enouph, save it!
            wsBuilder->saveAndStartNew();
        }
        else if(len<0)
        {
            // no WS started so far, start a new one!
            wsBuilder->startNewWaveset();
        }
    }

    this->lastIn = audioIn;
}

/**
 * @brief Get the idx of the newes waveset in the storage.
 * @return
 */

int WsAnalysis::getFirstWsIdx()
{
    return wsData->getFirstWsIdx();
}

/**
 * @brief Get the idx of the oldest waveset in the storage.
 * @return
 */

int WsAnalysis::getLastWsIdx()
{
   return wsData->getLastWsIdx();
}
```

### A.2.2 Synthese

**Listing A.5:** C++ Implementierung des *SynthTriggered*-UGens. Nimmt die Eingangssignale entgegen und interpretiert den Trigger. Die Berechnung der Synthese wird von der Klasse *SynthParallel* umgesetzt.

```
#include "WavesetSynthTriggered.h"

void WavesetSynthTriggered_Ctor(WavesetSynthTriggered *unit){

    // init UGen
    WavesetBase_Ctor(unit);
    new (&unit->wsSynth) SynthParallel(unit->wsData);
    unit->prevTrig = 1.0;


    SETCALC(WavesetSynthTriggered_next);
    WavesetSynthTriggered_next(unit, 1);
}


/**
 * @brief Process the next block of audio samples.
 * @param unit
 * @param inNumSamples
 */

void WavesetSynthTriggered_next(WavesetSynthTriggered *unit, int inNumSamples){

    // ^this.multiNew('audio', audioBuf, wsBuf, trig, idx, rate, groupSize, repeat)

    // Inputs:
    float *trig = IN(2);
    float *idxInFloat = IN(3);
    float rate = IN0(4);
    float groupSize = IN0(5);
    float repeat = IN0(6);
    // Outputs:
    float *out = OUT(0);

    // Waveset Playback
    for ( int i=0; i<inNumSamples; ++i) {

        // get Index Input
        int idxIn = (int) idxInFloat[i];
```

```
        // check for Trigger and valid idx input
        if(trig[i]>0.0 && unit->prevTrig<=0.0 && idxIn >= 0)
        {
            // we have a trigger, star new playback
            unit->wsSynth.startNewPlayback(idxIn,groupSize,rate,repeat);

        }


        // get Output
        float outSum = unit->wsSynth.getNextOutput();

        out[i] = outSum;

        unit->prevTrig=trig[i];

    }


}


void WavesetSynthTriggered_Dtor(WavesetSynthTriggered *unit){

}
```

**Listing A.6:** Implementierung der Klasse SynthParallel, welche die Geschäftslogik der ereignisbasierten Synthese umsetzt. Sie nimmt die Eingangssignale vom SynthParalelel-UGen entgegen und berechnet das Ausgabesignal der Synthese.

```
#ifndef SYNTHTRIGGERED_H
#define SYNTHTRIGGERED_H

#include "Synth.h"


/**
 * @brief Implements event-based waveset synthesis with a limited number of parallel
     ↪ wavesets.
 */

class SynthParallel : public Synth
{

public:
    SynthParallel(WsStorage* wsData);
    void startNewPlayback(int idxIn,int groupSize,float rate,int repeat);
    float getNextOutput();

    /** max number of possible parallel playbacks */
    static const int numPlayers = 512;

private:

    /** Players for parallel WS Playbacks */
    WsPlayer wsPlayers[numPlayers];

    /** idx of last playing iterator in the wsPlayers array */
    int lastActiveIteratorIdx;

};

#endif // SYNTHTRIGGERED_H
```

```
#include "SynthParallel.h"

SynthParallel::SynthParallel(WsStorage* wsData) : Synth(wsData)
{
    this->lastActiveIteratorIdx = -1;
}

/**
 * @brief Initiliazize a new parallel waveset playback.
```

```
 * The collected playbacks will be executed calling SynthParallel::getNextOutput().
 * @param wsIdx
 * @param groupSize
 * @param rate
 * @param repeat
 */

void SynthParallel::startNewPlayback(int wsIdx, int groupSize, float rate, int repeat)
{

    // We have a Trigger, get Waveset and set Iterator:
    // look for a free WavesetIterator
    for(int playIdx=0;playIdx<numPlayers;playIdx++)
    {
        WsPlayer* player = &this->wsPlayers[playIdx];
        if(player->endOfPlay()){
            // got a free Iterator: start Playback and exit loop

            this->initPlayback(player, (int) repeat,(int) groupSize,wsIdx ,rate);

            if(this->lastActiveIteratorIdx < playIdx){
                this->lastActiveIteratorIdx = playIdx;
            }

            break;
        }

        if(playIdx==numPlayers-1)
        {
            printf("SynthTriggered␣Warning:␣Max␣number␣of␣parallel␣Waveset␣playback␣
                ↪ exceeded!\n");
        }

    }

}

/**
 * @brief Get the next audio sample of the synthesis.
 * All parallel playbacks will be continued for one step and the outputs are summed up.
 * @return
 */

float SynthParallel::getNextOutput()
{
```

```
        // Play Wavesets from Iterators
        float outSum = 0.0;
        int lastPlayedIterator=-1;
        for(int playIdx=0;playIdx<=lastActiveIteratorIdx;playIdx++)
        {
            WsPlayer* player = &this->wsPlayers[playIdx];
            try
            {
                // play parallel Wavesets from Iterators
                if(!player->endOfPlay())
                {
                    outSum += player->nextSample();
                    lastPlayedIterator = playIdx;
                }
            }
            catch(...)
            {
                printf("Waveset␣playback␣failed!␣(unknown␣exception)\n");
                *player = WsPlayer(); // stop playback by resetting
            }
        }

        this->lastActiveIteratorIdx = lastPlayedIterator;


        return outSum;
}
```