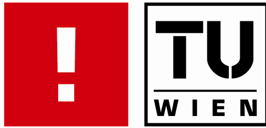
Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).



FAKULTÄT FÜR !NFORMATIK

Der Re-Compose Prototyp

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Wolfgang Deix
Matrikelnummer 9525896

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer:
Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer

Wien, 24. 10. 2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien Karlsplatz 13 Tel. +43/(0)1/58801-0 http://www.tuwien.ac.at

Abstract

Im Zuge dieser Diplomarbeit ist ein Prototyp einer VST-Plug-in-Software für die *Re-Compose GmbH* nach Vorgaben von Gründer und Geschäftsführer *Dr. Stefan Oertl* entwickelt worden. Der sogenannte *Re-Compose Prototyp* verbindet Audio-Softwareapplikationen mit einem von *DI Brigitte Rafael* für Re-Compose implementierten Musikanalysemoduls, der *Re-Compose Logik*, über die VST-Schnittstelle (*Virtual Studio Technology*). Der Musikdatenaustausch findet ausschließlich über das MIDI-Protokoll (*Musical Instrument Digital Interface*) statt. Die in Java implementierte *Re-Compose Logik* wird vom hier beschriebenen *Re-Compose Prototyp* über das *Java Native Interface* (JNI) eingebunden. Diese schriftliche Arbeit beschreibt die Planung und den implementierten Aufbau des Prototyps und gibt einen Ausblick auf folgende Entwicklungsarbeiten.

This diploma thesis has been set up in collaboration with Re-Compose GmbH *and its founder and managing director,* Dr. Stefan Oertl. *The main focus was the implementation of prototype software which connects audio software applications to a music analysis module developed by Re-Compose employee,* DI Brigitte Rafael. *An audio host sequencer and the former mentioned* Re-Compose Logic *are linked via the* Virtual Studio Technology *interface (VST). Music information is exchanged via the* Musical Instrument Digital Interface *protocol (MIDI). The prototype employs the* Java Native Interface *(JNI) to invoke the Re-Compose analysis module coded in the Java programming language. This paper describes the design and structure of the implemented Re-Compose prototype and gives a prospect of future challenges.*

Diese Diplomarbeit widme ich meiner Familie,

meinen Großeltern
Irene Deix
und
Anton Müller,
meinen Eltern *Waltraud* und *Rüdiger*,
meinen Geschwistern *Thomas*, *Stefan*, *Christina*
und meinem Neffen *David*.

Mein großer Dank gilt

meinen Eltern

Waltraud und *Rüdiger*,

die mir dieses Studium ermöglicht haben.

Dr. Stefan Oertl,

dessen Kreativität und Ideenvielfalt
diese Diplomarbeit ermöglicht hat.

Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer,

der mit Sorgfalt und Übersicht
diese Diplomarbeit betreut hat.

meiner Freundin *Alice*
für ihre Unterstützung.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wien, 18. November 2008 ________________________________

Unterschrift

Einleitung

Motivation

Musik ist ein uns ständig begleitendes Element. Wir hören und spielen Musik zur Entspannung, als Zeitvertreib, zur Motivation oder Ablenkung. Musik kann Gefühle und Gedanken kommunizieren oder gar beeinflussen.

Die qualitative Einteilung von Musik ist meist subjektiv, Kultur abhängig und kontrovers. *Dr. Stefan Oertl*, der Leiter der *Re-Compose GmbH*, einem privatwirtschaftlichen Unternehmen im Bereich der angewandten Forschung, betrachtet das Erleben von Musik aus der Sicht der psychologischen *Wirkungsforschung* als generalisierbaren Prozess, der darüber hinaus optimiert werden kann.

Aufbauend auf diese These erarbeitet *DI Brigitte Rafael* als Doktoratsstudentin an der Johannes Kepler Universität Linz und Mitarbeiterin der Re-Compose GmbH praktisch umsetzbare Algorithmen, um Musikstücke in Form von MIDI-Sequenzen oder -Dateien am Computer zu analysieren. An Hand der unterschiedlichen Analysen sollen automatisierte Umstrukturierungen der vorhandenen Sequenzen durchgeführt werden. Diese sogenannten *Resynthese*-Effekte helfen, das Erleben des Musikstücks weiter zu optimieren.

Ziele meiner Arbeit bei *Re-Compose* sind die Planung einer grafischen Darstellung der Analyseergebnisse, die Planung einer robusten Integration dieses Visualisierungsmoduls in bestehende Musik-Sequenzersoftware, beginnend mit einer ersten Umsetzung in Form des Re-Compose Prototyps, was gleichermaßen auch das Ziel dieser Diplomarbeit war.

Als Bindeglied zwischen Musik-Sequenzeranwendungen und dem von *DI Brigitte Rafael* in Java realisierten Logikmodul ist der Re-Compose Prototyp für unterschiedliche Schnittstellen implementiert worden. Über diese Schnittstellen werden Musikdaten zwischen *Host-*Sequenzer, Re-Compose *Prototyp* und Re-Compose *Logik* ausgetauscht.

Aufbau

Diese Diplomarbeit ist in zwei aufeinander aufbauende Sektionen unterteilt. Beginnend mit Teil I „Grundlagen“ werden die Grundlagen der Diplomarbeit präsentiert:

- Eine Vorstellung des Re-Compose Teams findet sich als Einstieg im Kapitel 1 „Re-Compose Überblick“ ab Seite 11.
- Für die Entwicklung eines ersten Prototyps als Bindeglied zwischen handelsüblichen Host-Sequenzern und der in Java realisierten Re-Compose Logik werden im Kapitel 2 „Schnittstellen“ ab Seite 13 die technischen Rahmenbedingungen vorgestellt.
- In den Kapiteln 3, 4, 5 und 6 werden grundlegende Zeitbegriffe der Musik und die verwendeten Schnittstellen näher erläutert.

Detaillierte Beschreibungen zu Planungs- und Entwicklungsarbeiten am Re-Compose Prototyp sind im Teil II ab Seite 38 zusammengefasst:

- Die Analyse der beteiligten externen Komponenten und die Festlegung der Entwicklungsumgebung werden im Kapitel 7 „Planung des Prototyps“ ab Seite 38 beschrieben.
- Ausgewählte Details der Implementierung des Prototyps finden sich im Kapitel 8 „Implementierungsdetails“ ab Seite 50.
- Planung und Design zur Darstellung der Analyseergebnisse der Re-Compose Logik werden im Kapitel 9 „Design der Visualisierung“ ab Seite 55 beschrieben.
- Abschließend werden im Kapitel 10 „Weiterführende Ideen“ ab Seite 59 mögliche Erweiterungen und zukünftige Arbeiten und Herausforderungen im Re-Compose Projekt aufgezählt.

Der Anhang enthält zur weiteren Übersicht ein *Abbildungsverzeichnis*, ein *Tabellenverzeichnis*, eine Liste von verwendeten Abkürzungen (*Akronyme*), ein *Glossar* und ein *Literaturverzeichnis*.

Inhaltsverzeichnis

Einleitung 5

Inhaltsverzeichnis 9

I. Grundlagen 10

1. Re-Compose Überblick 11
1.1. Dr. Stefan Oertl, Gründung & Leitung . . . 11
1.2. Stephan Dorfmeister, MBA, Finanzen . . . 11
1.3. Mag. Georg Furxer, Komposition . . . 12
1.4. DI Brigitte Rafael, IT . . . 12

2. Schnittstellen 13

3. Zeitbegriffe in der Musik 14
3.1. (Grund-) Schlag . . . 14
3.2. Takt . . . 14
3.3. Schlaglänge . . . 14
3.4. Notenlänge . . . 15
3.5. Taktbezeichnung . . . 15
3.6. Schläge pro Minute . . . 16
3.7. Impulse pro Viertelnote . . . 16
3.8. Samplerate/Sampleframes . . . 16
3.9. Zusammenfassung . . . 17

4. VST SDK (Version 2.4) 18
4.1. VST-Effekte . . . 18
4.2. VST-MIDI-Effekte . . . 18
4.3. VST-Instrumente . . . 18
4.4. Kommunikationsaufbau . . . 19
4.5. SDK-Aufbau . . . 22
4.6. Projektbeginn eines VST-Plug-ins . . . 22
4.7. Methode *processReplacing* . . . 25
4.8. VST-Parameter und VST-Programme . . . 25
4.9. Einbinden der eigenen Plug-in Klasse in das VST SDK . . . 26
4.10. Zusammenfassung . . . 26

5. MIDI **27**
5.1. MIDI-Protokoll . . . 27
5.2. Note On/Off Nachrichten . . . 28
5.3. Zusammenfassung . . . 29

6. JNI **30**
6.1. Einleitung . . . 30
6.2. Überblick . . . 30
6.3. Meta-Programmiersprache . . . 32
6.4. Basistypen . . . 32
6.5. Referenzen auf Arrays, Strings, Objekte und Methoden . . . 33
6.6. Arbeitsanweisungen . . . 33
6.7. Klassendeskriptor und Klassenreferenzen . . . 33
6.8. Feld- und Methodendeskriptoren . . . 34
6.9. Objektorientiertes JNI . . . 35
6.10. Natives Programmieren mit JNI . . . 35
6.11. Zusammenfassung . . . 36

II. Prototyp **37**

7. Planung des Prototyps **38**
7.1. Festlegung der Entwicklungsumgebung . . . 38
7.1.1. Windows DLL . . . 38
7.1.2. VST SDK Version 2.4 . . . 38
7.1.3. JNI und JDK Version 6.0 . . . 39
7.2. Identifikation der externen Komponenten . . . 39
7.3. Strukturierung des Prototyps . . . 39
7.3.1. Einteilung in Module . . . 39
7.3.2. Einteilung in Objektklassen . . . 42
7.4. Beschreibung der Objektklassen . . . 42
7.4.1. VstPlug . . . 42
7.4.2. Sequencer . . . 44
7.4.3. Data . . . 44
7.4.4. Logic . . . 45
7.4.5. JniTools . . . 47
7.4.6. Threading . . . 48
7.5. Zusammenfassung . . . 49

8. Implementierungsdetails **50**
8.1. VST-MIDI-Anbindung . . . 50
8.2. Handhabung der Zeiteinheiten . . . 50
8.2.1. VST Timing . . . 50
8.2.2. Re-Compose Timing . . . 50
8.2.3. Konversionsbeispiel . . . 51
8.3. Multithreading . . . 51
8.3.1. VST Thread vs. JNI Thread . . . 52

8.4. Prozessablauf 52
8.5. Zusammenfassung 53

9. Design der Visualisierung **55**
9.1. Einleitung 55
9.2. Hierarchische Analyse 55
9.2.1. Hierarchische Darstellung 55
9.3. Geometrische Analyse 56
9.3.1. Geometrische Darstellung 56
9.4. Zusammenfassung 57

10. Weiterführende Ideen **59**
10.1. Das „Multitrack-Problem“ 59
10.1.1. Master/Slave-Plug-ins 59
10.2. Interaktivität 60
10.3. Farbkodierung 60
10.4. Zusammenfassung 60

Anhang **61**

A. Abbildungsverzeichnis **61**

B. Tabellenverzeichnis **62**

C. Literaturverzeichnis **63**

Glossar **64**

Akronyme **66**

Teil I.

Grundlagen

Kapitel 1.

Re-Compose Überblick

1.1. Dr. Stefan Oertl, Gründung & Leitung

Die *Re-Compose GmbH* wurde im April 2007, damals als *Re-Compose Stefan Oertl & Partner GnbR* gegründet. Dabei legte Stefan Oertl die Schwerpunkte seiner eigenen Ausbildungslaufbahn, *Informationstechnologie* (abgeschlossenes Diplomstudium an der Fakultät für Informatik der technischen Universität Wien) und *Musikkomposition* (abgeschlossener Filmmusikausbildungszweig der *Thornton School of Music, Los Angeles, Kalifornien*) unter Betrachtung der *Wirkungspsychologie* (abgeschlossenes Doktoratsstudium an der Fakultät für Psychologie der Universität Wien) als Kernkompetenzen der Firma fest, die es zu kombinieren und verschmelzen gilt.

1.2. Stephan Dorfmeister, MBA, Finanzen

Ein *Partner* der ersten Stunde war und ist Stephan Dorfmeister, der sich mit seinen Erfahrungen in der Privatwirtschaft gemeinsam mit Stefan Oertl der finanziellen Angelegenheiten annimmt. Stephan Dorfmeister ist geschäftsführender Gesellschafter der *Dorfmeister Projektentwicklungs GmbH* und unter anderem auch erfolgreicher Manager des Labels *G-Stone Recordings*, einem Zusammenschluss bekannter Künstler der sogenannten „*Wiener Elektronik*“, denen beispielsweise sein Bruder Richard Dorfmeister und Peter Kruder aus der international berühmten und anerkannten Formation *Kruder & Dorfmeister* angehören.

Stephan Dorfmeister ist Obmann des staatlich geförderten Vereins *Austrian Music Ambassador Network (AMAN)*, dessen Ziel die internationale Vermarktung von in Österreich entstandener und produzierter Musik aus unterschiedlichen Genres wie *Electronic*, *World Music*, *Alternative*, *Rock* und *Pop* sowie *Jazz* ist.

1.3. Mag. Georg Furxer, Komposition

Georg Furxer ist als ausgebildeter Komponist, Arrangeur und Pianist (Universität für Musik und darstellende Kunst in Wien) eine Verstärkung für den musikalischen Forschungsbereich. Georg Furxer hat für seine Arbeit schon unterschiedliche Auszeichnungen und Stipendien entgegennehmen können (Quelle: Musikdokumentationsstelle des Landes Vorarlberg):

- 2002 Preisträger des Kulturpreises für Komposition der Stadt Feldkirch.
- 2003 1. Platz des Kompositionswettbewerbs des Quartett22.
- 2005 Preisträger des Theodor-Körner Fonds.
- 2005 1. Platz des Gustav Mahler Kompositionspreises der Stadt Klagenfurt.
- 2008 Österreichisches Staatsstipendium für Komposition.

1.4. DI Brigitte Rafael, IT

Brigitte Rafael ist im Re-Compose Team, wie zu Beginn schon beschrieben, für die angewandte Forschungsarbeit im IT-Bereich zuständig. In ihrem von Re-Compose unterstützten Doktoratsstudium an der Johannes Kepler Universität Linz plant sie gemeinsam mit Stefan Oertl die praktische Umsetzung seiner Optimierungstheorien aus dem Bereich der Wirkungspsychologie und implementiert diese in Java als Re-Compose Logikmodul.

Kapitel 2.

Schnittstellen

Für die Planung des Prototyps war es erforderlich, die Anforderungen von vier unterschiedlichen Schnittstellen zu berücksichtigen:

Audio-Sequenzersoftware steht zur Zusammenarbeit über die Virtual Studio Technology (VST) Schnittstelle bereit. Ein VST Software Development Kit (SDK) hilft bei der Entwicklung von *nativen* Prozessen (sogenannten *Plug-ins*), die über die VST-Schnittstelle mit einem *Host* Sequenzer kommunizieren können (siehe auch Kapitel 4 „VST SDK (Version 2.4)“).

Der Austausch von Musikdaten wird bei der VST Schnittstelle über das *Musical Instrument Digital Interface (MIDI)* abgewickelt (siehe auch Kapitel 5 „MIDI“).

Die Re-Compose Logik als Java Modul kann wiederum von *nativen* Prozessen am besten über das Java Native Interface (JNI) eingebunden werden: Dazu wird über JNI eine *virtuelle Java-Maschine* (JavaVM) gestartet. In dieser JavaVM wird die Re-Compose Logik instanziiert und betrieben (siehe auch Kapitel 6 „JNI“).

Der eigentliche Austausch der Musikdaten zwischen dem Prototyp und der Logik findet über eine eigens definierte, proprietäre und schlanke Schnittstelle statt. Über diese Schnittstelle werden Listen von einfachen Notenobjekten übermittelt (vgl. Abschnitt 7.3.2 „Einteilung in Objektklassen“).

Bei jeder Datenkommunikation findet ein Austausch von Musikdaten statt. Dafür ist es erforderlich, sich mit grundlegenden Zeitbegriffen der Musik vorab auseinander zu setzen. Im Kapitel 3 werden die wichtigsten Zeitgrößen im Überblick vorgestellt.

Alle Schnittstellen gemeinsam bilden ein Grundgerüst, auf dem die Planung und die Entwicklung des Re-Compose Prototyps aufbaut.

Kapitel 3.

Zeitbegriffe in der Musik

3.1. (Grund-) Schlag

Ein Schlag oder auch Puls (engl. *beat*) ist eine atomare Einheit in der hierarchischen Zeiteinteilung der Musik. Die Zeit zwischen zwei aufeinander folgenden Schlägen impliziert das Tempo eines Musikstücks (siehe auch Abschnitt 3.3 „Schlaglänge“ und vgl. Abschnitt 3.6 „Schläge pro Minute“).

Abbildung 3.1.: Ein Viertelnotenschlag

3.2. Takt

Die nächst größere Zeiteinheit in der Musik ist der Takt. Ein Takt definiert sich über seine Taktart, die eine definierte Anzahl an Noten mit gleicher Notenlänge zu einem Takt gehörig vorschreibt. Ein $\frac{4}{4}$ Takt gruppiert beispielsweise 4 Viertelnoten innerhalb eines Takts (siehe auch Abb. 3.2 „Vier Viertelnoten in einem $\frac{4}{4}$ Takt“ und vgl. Abschnitt 3.5 „Taktbezeichnung“).

3.3. Schlaglänge

Die Schlaglänge ist die Zeit in Sekunden zwischen zwei aufeinander folgenden Schlägen.

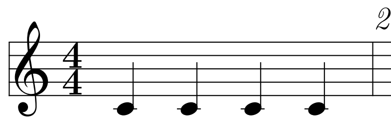


Abbildung 3.2.: Vier Viertelnoten in einem $\frac{4}{4}$ Takt

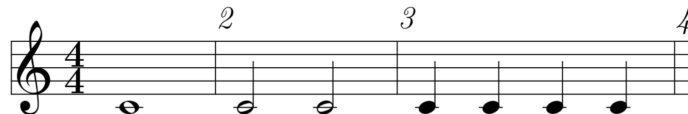


Abbildung 3.3.: Ganze-, Halbe- und Viertelnoten

3.4. Notenlänge

Unabhängig von der eigentlichen Schlaglänge werden in der Musik Noten in zueinander relativen Notenlängen angeschrieben. Eine *ganze Note* hat dabei in einem $\frac{4}{4}$ Takt eine Dauer von genau einer Taktlänge. Analog dazu haben zwei *halbe Noten* eine Dauer von einer ganzen Note, zwei *Viertelnoten* die Dauer einer *Halben* und so weiter (siehe auch Abb. 3.3 „Ganze-, Halbe- und Viertelnoten“).

3.5. Taktbezeichnung

Die Taktart definiert nun einerseits eine Anzahl an zusammengehörigen Noten mit andererseits einer vorgegebenen Notenlänge. Diese duale Information wird über die *Taktbezeichnung* wiedergegeben, die meist als Bruchzahl angeschrieben ist. Der *Zähler* des Bruchs definiert dabei die Anzahl der Noten pro Takt. Der *Nenner* legt die Notenlänge fest.

Wie in Abbildung 3.2 gezeigt, sind die Basis eines $\frac{4}{4}$ Takts vier Viertelnoten. Ein $\frac{3}{4}$ Takt kommt mit drei Viertelnoten pro Takt aus. Der in der volkstümlichen Musik weit verbreitete „kleine *Alla-breve*-Takt“ hat die Taktbezeichnung $\frac{2}{2}$.

Der *Nenner* in der Taktbezeichnung legt nicht notwendigerweise das Maximum des *Zählers*, also der möglichen Anzahl zusammengehöriger Noten fest. Beispielsweise ist das sehr populär gewordene Jazzstück *Take Five* (geschrieben 1959 von *Paul Desmond*), wie auch der Titel schon anklingen lässt, im $\frac{5}{4}$ Takt geschrieben.

3.6. Schläge pro Minute

Als ein Maß für das Tempo eines Musikstücks werden häufig die *Schläge pro Minute* (engl. *beats per minutes*) angegeben. Die Abkürzung des englischen Begriffes wird allgemein als Maßeinheit verwendet: *beats per minute (bpm)* .

Durch Umkehrung lässt sich aus dem angegebenen *bpm*-Wert die Schlagdauer s in Sekunden berechnen (Glg. 3.1).

$$s = 60 * \frac{1}{bpm} \tag{3.1}$$

3.7. Impulse pro Viertelnote

Für die zeitliche Positionierung innerhalb von Musikstücken ist vor allem im Bereich des Musical Instrument Digital Interface (MIDI) der Begriff *pulses per quarter (ppq)* (engl. *pulses per quarter*) von großer Bedeutung. Dabei werden Viertelnoten in eine definierte Anzahl von Impulsen weiter unterteilt.

Der standardisierte (minimale) MIDI-Zeitgeber arbeitet mit einer Auflösung von 24 pulses per quarter (ppq). D.h. zeitliche Ereignisse und Positionen können höchstens mit der Genauigkeit einer $\frac{1}{24}$ Viertelnote aufgelöst werden. MIDI-Sequenzer arbeiten heutzutage (zumindest intern) mit einem Vielfachen von 24 ppq. Moderne Sequenzer werden z.B. mit 960 ppq angegeben.

3.8. Samplerate/Sampleframes

Audioprogramme verarbeiten Audiosignale diskret bei einer definierten Samplerate. Ein diskreter Zeitpunkt einer digitalisierten Audiosequenz wird *Sampleframe* genannt. Sehr häufig werden von Audio-Bearbeitungssystemen Positionen innerhalb eines Musikstücks über *Sampleframe*-Nummern ausgetauscht. Somit bezeichnet das *Sampleframe* mit Nummer 44100 bei einer *Samplerate* von 44100 Hz die Position im Musikstück nach genau einer Sekunde.

Ganzzahlige *Sampleframe*-Nummerierungen haben vor allem bei der Computerverarbeitung den Vorteil, dass sie zu weniger numerischen Problemen wie etwa *Auslöschung* oder *Rundungsfehler* führen als etwa bei Fließkommawerten (vgl. *ppq* Abschnitt 3.7). Bei langen Musikstücken und hohen *Sampleraten* muss hingegen kontrolliert werden, dass es zu keinem Überlauf bei der *Sampleframe*-Nummerierung kommt.

3.9. Zusammenfassung

Diese kurze Übersicht beschreibt sehr knapp die für diese Diplomarbeit relevanten Zeitbegriffe der Musik. Bewusst wurden Begriffe wie *Rhythmik*, *Metrik*, *Akzentuierung* und weitere mehr vernachlässigt. Für eine detailliertere Auseinandersetzung mit der Thematik und für weiterführende Informationen sei „*The Rhythmic Structure of Music*“ [GWC60] als Referenz angeführt.

Kapitel 4.

VST SDK (Version 2.4)

VST von *Steinberg Media Technologies GmbH* ist eine Schnittstellendefinition für Austausch und Manipulation von Musikdaten zwischen Softwareanwendungen. VST sieht dafür eine hierarchische Struktur ähnlich einer Client/Server-Architektur vor. Eine Hauptanwendung (der sogenannte *VST-Host*) hat die Kontrolle inne und kann mehrere VST-konforme Clientprozesse (die sogenannten *VST-Plug-ins*) starten und einbinden.

4.1. VST-Effekte

Die übertragenen Musikdaten können ausschließlich aus Audiosignalen bestehen, welche vom Host an das Plug-in übergeben, durch das Plug-in verarbeitet/verändert und anschließend wieder zurück an den Host gesendet werden. Derartige Plug-ins werden als *VST-Effekte* bezeichnet. Klassische Effekte sind zum Beispiel *Hall*- oder *Delay*-Effekte oder auch Audiosignalfilter wie *Equalizer*.

4.2. VST-MIDI-Effekte

Ein Plug-in kann auch ausschließlich MIDI-Signale vom Host entgegennehmen, welche nach einem Verarbeitungsschritt, wie beispielsweise Ergänzen von Akkordtönen, geändert oder erweitert wieder als MIDI-Signale an den Host zurückgegeben werden. Diese Plug-ins werden *VST-MIDI-Effekte* genannt.

4.3. VST-Instrumente

Empfängt ein VST-Plug-in MIDI-Signale von der Host-Anwendung und gibt daraufhin Audiosignale entsprechend der erhaltenen MIDI-Daten zurück an diesen Host, spricht man von

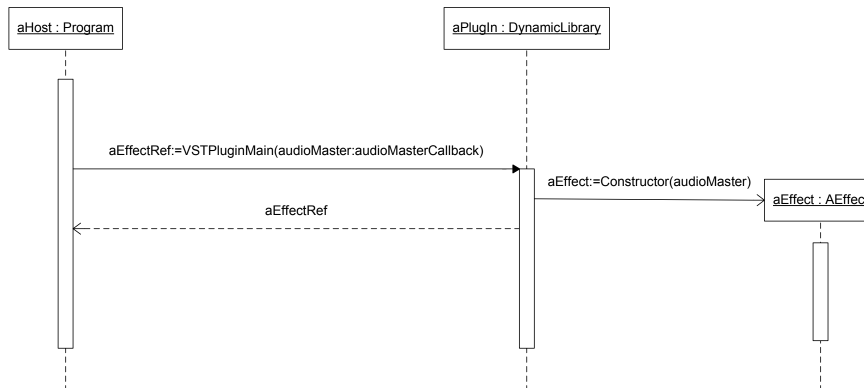


Abbildung 4.1.: UML-Sequenzdiagramm: VST-Initialisierung

VST-Instrumenten. Ein VST-Instrument erfüllt dabei dieselbe Funktion wie ein Software-Synthesizer.

4.4. Kommunikationsaufbau

Damit ein VST-Host passende VST-Plug-ins erkennen und ausführen kann, werden Plug-ins als dynamische Bibliotheken mit standardisierten Einsprungsmethoden *VSTPluginMain* entwickelt. In Windows werden diese Bibliotheken in Form von *multi-threaded Dynamic Link Libraries (dll)* gespeichert, für Mac OS X als sogenannte *Bundle* Dateien, für BeOS und SGI (MOTIF, UNIX) als *Shared Library* [Ste06]. Findet ein Host die Einsprungsmethode *VSTPluginMain* in einer Bibliothek, ruft er diese zur Initialisierung auf und übergibt als Parameter einen Funktionszeiger auf eine eigene Rückrufmethode vom Typ *audioMasterCallback*, über die in weiterer Folge das Plug-in Anfragen an den Host stellen kann.

Der Rückgabeparameter von *VSTPluginMain* des VST-Plug-ins ist wiederum die Referenz, über die der Host das Plug-in kontaktieren kann. Um dies zu gewährleisten ist der Rückgabewert ein Zeiger auf eine Instanz einer Klasse nach Schema der vordefinierten Struktur *AEffect* (siehe auch Abb. 4.1 „UML-Sequenzdiagramm: VST-Initialisierung“).

Der VST-Host kann nach dieser Initialisierungsphase das VST-Plug-in über diesen zurück-

Abbildung 4.2.: UML-Sequenzdiagramm: VST-Kommunikation

gegebenen Zeiger als eine der Struktur *AEffect* entsprechende Instanz ansprechen. Dazu sind bestimmte Verarbeitungsmethoden in *AEffect* vordefiniert, die der Host zu bestimmten Ereignissen aufruft. Die Hauptverarbeitungsmethode heißt *dispatcher*. Diese nimmt Verarbeitungscodes (*opcodes*) und Argumente von der Host-Anwendung für die Weiterverarbeitung entgegen (siehe auch Abb. 4.2 „UML-Sequenzdiagramm: VST-Kommunikation“ und Tab. 4.1 „Aufzählung: Plug-in-Verarbeitungscodes (VST SDK 2.4)“).

Das VST-Plug-in kann in eigener Sache den Host mit Hilfe der erhaltenen Rückrufmethode des Typs *audioMasterCallback* wiederum mit vordefinierten Verarbeitungscodes und dazugehörigen Argumenten kontaktieren (siehe auch Abb. 4.2 „UML-Sequenzdiagramm: VST-Kommunikation“ und Tab. 4.2 „Aufzählung: Host-Verarbeitungscodes (VST SDK 2.4)“).

Bei der mittlerweile großen Anzahl an verfügbaren Verarbeitungscodes wurden ab Version 2.0 manche Codes nur *optional* hinzugefügt. Host und Plug-in müssen für optionale Verarbeitungscodes keine Bearbeitungsprozeduren bereitstellen, jedoch muss auf Anfrage mit Code `effCanDo` bzw. mit `audioMasterCanDo` Auskunft gegeben werden, ob auf bestimmte optionale Verarbeitungscodes reagiert wird oder nicht.

Wie dieser wohlgemerkt sehr kurz gehaltene Einstieg in VST zeigt, ist eine VST-konforme Entwicklung ein komplexes Vorhaben. *Steinberg Media Technologies GmbH* hat aus diesem Grund ein Software Development Kit (SDK) entwickelt, um vor allem die Programmierung von VST-Plug-ins zu vereinheitlichen und zu vereinfachen. Dieses VST SDK ist mit von

Tabelle 4.1.: Aufzählung: Plug-in-Verarbeitungscodes (VST SDK 2.4)

(Version 1.x)	effGetProgramNameIndexed	effGetMidiProgramName
effOpen = 0	effGetInputProperties	effGetCurrentMidiProgram
effClose	effGetOutputProperties	effGetMidiProgramCategory
effSetProgram	effGetPlugCategory	effHasMidiProgramsChanged
effGetProgram	effOfflineNotify	effGetMidiKeyName
effSetProgramName	effOfflinePrepare	effBeginSetProgram
effGetProgramName	effOfflineRun	effEndSetProgram
effGetParamLabel	effProcessVarIo	*(Version 2.3)*
effGetParamDisplay	effSetSpeakerArrangement	effGetSpeakerArrangement
effGetParamName	effSetBypass	effShellGetNextPlugin
effSetSampleRate	effGetEffectName	effStartProcess
effSetBlockSize	effGetVendorString	effStopProcess
effMainsChanged	effGetProductString	effSetTotalSampleToProcess
effEditGetRect	effGetVendorVersion	effSetPanLaw
effEditOpen	effVendorSpecific	effBeginLoadBank
effEditClose	effCanDo	effBeginLoadProgram
effEditIdle	effGetTailSize	*(Version 2.4)*
effGetChunk	effGetParameterProperties	effSetProcessPrecision
effSetChunk	effGetVstVersion	effGetNumMidiInputChannels
(Version 2.0)	*(Version 2.1)*	effGetNumMidiOutputChannels
effProcessEvents	effEditKeyDown	
effCanBeAutomated	effEditKeyUp	
effString2Parameter	effSetEditKnobMode	

Tabelle 4.2.: Aufzählung: Host-Verarbeitungscodes (VST SDK 2.4)

(Version 1.x)	audioMasterOfflineRead
audioMasterAutomate = 0	audioMasterOfflineWrite
audioMasterVersion	audioMasterOfflineGetCurrentPass
audioMasterCurrentId	audioMasterOfflineGetCurrentMetaPass
audioMasterIdle	audioMasterGetVendorString
(Version 2.x)	audioMasterGetProductString
audioMasterGetTime	audioMasterGetVendorVersion
audioMasterProcessEvents	audioMasterVendorSpecific
audioMasterIOChanged	audioMasterCanDo
audioMasterSizeWindow	audioMasterGetLanguage
audioMasterGetSampleRate	audioMasterGetDirectory
audioMasterGetBlockSize	audioMasterUpdateDisplay
audioMasterGetInputLatency	audioMasterBeginEdit
audioMasterGetOutputLatency	audioMasterEndEdit
audioMasterGetCurrentProcessLevel	audioMasterOpenFileSelector
audioMasterGetAutomationState	audioMasterCloseFileSelector
audioMasterOfflineStart	

Steinberg Media Technologies GmbH vorgegebenen Lizenzbestimmungen frei erhältlich und in der Programmiersprache C++ für unterschiedliche Plattformen wie Macintosh, Windows oder BeOS erhältlich.

4.5. SDK-Aufbau

Das VST SDK in der Version 2.4 beinhaltet vier wesentliche Komponenten, die vor allem bei der Entwicklung von VST-Plug-ins sehr nützlich sind:

- *Pluginterface* - C-konforme Definition der Plug-in-Schnittstelle.
- *SDK Source* - C++ Implementierung von Basisklassen der Schnittstelle, die als Ausgangspunkt jeder Plug-in-Entwicklung dienen.
- *VstGui* - C++ Paket zur plattformübergreifenden Entwicklung von grafischen Benutzeroberflächen.
- *SDK Examples* - Beispielprojekte zur Veranschaulichung der unterschiedlichen Einsatzmöglichkeiten der *Virtual Studio Technology*.

4.6. Projektbeginn eines VST-Plug-ins

Das VST SDK enthält keine Laufzeit- oder Code-Bibliotheken der Basisklassen, wodurch einerseits Versionskonflikte vermieden werden und andererseits die Entwicklung für unterschiedliche Plattformen erleichtert wird. Das hat zur Folge, dass jedes neue Plug-in-Projekt auch die Quelldateien dieser Basisklassen referenzieren, kompilieren und binden muss. In Tabelle 4.3 sind jene SDK-Dateien mit ihren jeweiligen Klassen und Strukturen aufgelistet, die einen guten Ausgangspunkt zur Entwicklung von VST-Plug-ins darstellen.

Für ein minimales VST-Plug-in unter Windows, welches mit Visual Studio als *dynamic link library* erstellt wird, werden die beschriebenen VST-SDK-Basisdateien zu einem Bibliotheksprojekt hinzugefügt. Dabei ist zu beachten, dass das Projekt eine *Multithread*-Bibliothek erzeugt (siehe auch Abb. 4.3 „Struktur der VST-SDK-Basisdateien“).

Visual Studio muss so konfiguriert werden, dass eine Virtual Studio Technology (VST)-konforme dynamische Bibliothek mit Einstiegsmethode *VSTPluginMain* erzeugt wird. Dafür wird eine *export* Definitionsdatei (mit Dateiendung *.def*) zum Projekt hinzugefügt. In diese Definitionsdatei wird die Haupteinstiegsmethode *main* der Bibliothek festgelegt (siehe auch Lst. 4.1 „Visual Studio Export-Definitionsdatei“).

Tabelle 4.3.: SDK-Quelldateien eines VST-Basisprojektes

DATEI	FUNKTION	BESCHREIBUNG
`aeffect.h`	Interface	Basisdefinitionen für jeweilige Entwicklungsplattform und Definition der grundlegenden Struktur `struct AEffect`.
`aeffectx.h`	Interface	VST 2.x Erweiterung: Definition von erweiterten Schnittstellen-Basisstrukturen und -Aufzählungen.
`vstfxstore.h`	Interface	Definitionen von Strukturen und Aufzählungen für ein einheitliches VST-Parametersystem (siehe auch Abschnitt 4.8 „VST-Parameter und VST-Programme“).
`audioeffect.h` `audioeffect.cpp`	*AudioEffect*	Virtuelle Basisklasse *AudioEffect* als Grundgerüst für Effekt-Plug-ins der Version 1.x. siehe `audioeffect.h`.
`audioeffectx.h` `audioeffectx.cpp`	*AudioEffectX*	Virtuelle Basisklasse *AudioEffectX* abgeleitet von Klasse *AudioEffect* für Plug-ins der Version 2.x . siehe `audioeffectx.h`.
`aeffeditor.h`	*AEffEditor*	Virtuelle Klasse *AEffEditor* zur Erweiterung eines Plug-ins um eine eigene grafische Benutzeroberfläche.
`vstplugmain.cpp`	*VSTPluginMain*	Vordefinierte Einsprungsmethode *VSTPluginMain* für eine automatisierte Initialisierung der Kommunikation von Host und Plug-in.

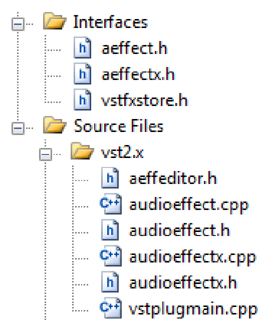


Abbildung 4.3.: Struktur der VST-SDK-Basisdateien

Listing 4.1: Visual Studio Export-Definitionsdatei

```
EXPORTS
  VSTPluginMain
  main=VSTPluginMain
```

Das minimale Plug-in wird nun als Klasse *MiniPlug* von der Basisklasse *AudioEffectX* abgeleitet (siehe auch Lst. 4.2 „MiniPlug Beispiel“).

Listing 4.2: MiniPlug Beispiel

```
class MiniPlug : public AudioEffectX
{
public:
  MiniPlug( audioMasterCallback audioMaster );
  ~MiniPlug();

  virtual void processReplacing(
    float** inputs,
    float** outputs,
    VstInt32 sampleFrames);

  virtual void setParameter( VstInt32 index, float value );
  virtual float getParameter( VstInt32 index );
  virtual void getParameterLabel( VstInt32 index, char* label );
  virtual void getParameterDisplay( VstInt32 index, char* text );
  virtual void getParameterName( VstInt32 index, char* text );

  virtual void setProgram( VstInt32 program );
  virtual void setProgramName( char* name );
  virtual void getProgramName( char* name );
  virtual bool getProgramNameIndexed(
    VstInt32 category,
    VstInt32 index,
    char* text);

  virtual bool getEffectName( char* name );
  virtual bool getVendorString( char* text );
  virtual bool getProductString( char* text );
  virtual VstInt32 getVendorVersion();
  virtual VstInt32 canDo( char* text );
}
```

Die Klassendeklaration (Lst. 4.2) von *MiniPlug* zeigt vier wichtige Teilbereiche eines VST-Plug-ins auf: Die polymorphe Methode `void processReplacing(...)`, zwei Methodengruppierungen zum Verarbeiten von VST-Parametern und (Parameter-) Programmen und eine Gruppe von Methoden, die Informationen über das Plug-in bereitstellen.

4.7. Methode *processReplacing*

Die Methode *processReplacing* ist für die eigentliche Verarbeitung der Audiosignale der jeweiligen Plug-ins zuständig. Der Parameter *sampleFrames* gibt Auskunft über die Anzahl der zu verarbeitenden Sampleframes. Pro übertragener Sampleframe-Nummer ist ein Frequenzwert eines bestimmten diskreten Zeitpunktes gespeichert (siehe auch Abschnitt 3.8 „Samplerate/Sampleframes“). Die Argumente *inputs* und *outputs* halten jeweils *float* Speicherbereiche für genau diese angegebene Anzahl an Sampleframes bereit. Ein einfacher *„Durchlaufeffekt“* (engl. *pass through*), der die eingehenden Audiosignale unverändert an den Host zurück gibt, wird durch eine Iteration über die Werte des *inputs* Parameters und dabei gleichzeitiger Zuweisung an *outputs* erzeugt (siehe auch Lst. 4.3 „Durchlaufeffekt“).

Listing 4.3: Durchlaufeffekt

```
void MiniPlug::processReplacing(
  float** inputs,
  float** outputs,
  VstInt32 sampleFrames)
{
  // Stereo setup with 2 inputs and 2 outputs
  float* in1 = inputs[0];
  float* in2 = inputs[1];
  float* out1 = outputs[0];
  float* out2 = outputs[1];

  // iterate over all sample frames
  while (--sampleFrames >= 0)
  {
    // pass-through
    (*out1++) = (*in1++);
    (*out2++) = (*in2++);
  }
}
```

4.8. VST-Parameter und VST-Programme

VST stellt eine eigene Verarbeitungsmöglichkeit zum Einstellen, Darstellen und Verwalten von VST-Plug-in-Parametern zur Verfügung. Über die Methoden *getParameterName()*, *getParameterDisplay()*, *getParameterLabel()* und *setParamter()* ist es Host und Plug-in gleichermaßen möglich, Parameterwerte zu verändern und korrekt anzuzeigen.

Für eine einfachere Verwaltung können Parameterwertebelegungen als *Programme* abgespeichert werden. So ist es auch möglich, Voreinstellungen in Form von Programmen den Benutzern von Plug-ins anzubieten. Analog zu Parametern implementieren Plug-ins dafür die Methoden *getProgramName()*, *setProgramName()*, *getProgramNameIndexed()* und *setProgram()*.

4.9. Einbinden der eigenen Plug-in Klasse in das VST SDK

Abschließend gilt es, das eigene Plug-in so einzubinden, dass es durch die vorgegebenen VST-Software Development Kit (SDK)-Verarbeitungsroutinen instanziiert und gestartet werden kann. Dazu ist eine globale Methode *createEffectInstance* vorgesehen, die im VST SDK als *external* deklariert ist und vom Plug-in-Entwickler zu definieren ist (siehe Lst. 4.4 „Definition von *createEffectInstance* in `vstplugmain.cpp`“ und Lst. 4.5 „Deklaration von *createEffectInstance* (Bsp. *MiniPlug*)“).

Listing 4.4: Definition von *createEffectInstance* in `vstplugmain.cpp`

```
1 //-------------------------------------------------------------------------
2 /** Must be implemented externally. */
3 extern AudioEffect* createEffectInstance( audioMasterCallback audioMaster );
```

Listing 4.5: Deklaration von *createEffectInstance* (Bsp. *MiniPlug*)

```
1 //-------------------------------------------------------------------------
2 AudioEffect* createEffectInstance( audioMasterCallback audioMaster )
3 {
4   return new MiniPlug( audioMaster ); // erzeuge Plug-in Instanz
5 }
```

4.10. Zusammenfassung

In diesem Kapitel wurden die grundsätzlichen Konzepte und die Funktionsweise der *Virtual Studio Technology* Schnittstelle (VST) von *Steinberg Media Technologies GmbH* vorgestellt. Des Weiteren wurde für die Entwicklung eines VST-konformen *Plug-ins* der Aufbau des VST Software Development Kit (SDK) präsentiert und an Hand einiger Beispiele mit Quellcode verdeutlicht.

Kapitel 5.

MIDI

Die ersten Entwürfe eines offenen, allgemeinen Standards für digitale Musikschnittstellen gehen zurück in die Jahre 1982, 1983. Das erste Musical Instrument Digital Interface (MIDI)-fähige Instrument wurde zu Beginn von 1983 am Markt eingeführt. Ein Konsortium von japanischen und amerikanischen Synthesizerherstellern veröffentlichte das Musical Instrument Digital Interface in Version 1.0 im August 1983 [Roa96].

Der einfache Ansatz von MIDI, Musik als simple *Note On/Note Off* Steuersignale unabhängig vom eigentlichen Klangbild zu übertragen, war gleichzeitig dessen große Stärke. Die Bearbeitung des Protokolls stellt keine großen Anforderungen an die Rechenleistung und konnte schon damals mit erschwinglichen Rechnerkomponenten realisiert werden.

Ein typisches MIDI-System aus dieser Zeit bestand in der Regel aus elektronischen Instrumenten oder anderen speziell entwickelten Eingabegeräten, welche MIDI-Ereignisse (*events*) erzeugen, während auf ihnen gespielt wird. Ein MIDI-Sequenzer empfängt diese MIDI-Signale und kann sie intern im zeitlichen Kontext und Ablauf abspeichern (*MIDI Recording*) und zu einem späteren Zeitpunkt wiedergeben. Ein MIDI-Synthesizer interpretiert MIDI-Signale von MIDI-Eingabegeräten oder MIDI-Sequenzern und erzeugt daraus wieder dementsprechende Audiosignale durch Frequenzmodulation.

MIDI-Sequenzer definieren Zeitpunkte von Ereignissen in pulses per quarter (ppq), also „Impulse pro Viertelnote“ als Zeiteinheit. Die ersten MIDI-Sequenzer verarbeiteten Zeitpunkte mit einer maximalen Auflösung von 24 pulses per quarter (ppq). Das bedeutet $\frac{1}{24}$ einer Viertelnote war die kürzest mögliche Note, die nach einer Aufnahme wieder korrekt wiedergegeben werden konnte (siehe auch Kapitel 3 „Zeitbegriffe in der Musik“).

5.1. MIDI-Protokoll

MIDI-Nachrichten haben keine bestimmte Empfängeradresse, d.h. jeder MIDI-Bus-Teilnehmer kann eine gesendete MIDI-Nachricht lesen. Für eine mögliche Differenzierung und Unterscheidung von Nachrichten wurden für MIDI eigene Kanäle (*MIDI Channels*) definiert.

Tabelle 5.1.: MIDI-Protokoll: Bytes 1..4

Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24
OP3	OP2	OP1	OP0	CH3	CH2	CH1	CH0

OP Verarbeitungscode (*Opcode*)
CH Kanalnummer (*MIDI Channel*)

Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16
0	D1_6	.	.	.	.	.	D1_0

D1 1. Parameter

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
0	D2_6	.	.	.	.	.	D2_0

D2 2. Parameter

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	D3_6	.	.	.	.	.	D3_0

D3 3. Parameter

So ist es möglich, mehrere MIDI-Instrumente auf unterschiedlichen Kanälen am selben Bus zusammen zu führen, ohne dass sich ihre Nachrichten gegenseitig behindern. Eine MIDI-Nachricht besteht aus 4 Byteblöcken (32 Bit). Der erste Block beinhaltet einen MIDI-Verarbeitungscode (*high word*, Bits 7-4) und einen von 16 MIDI-Kanälen (*low word*, Bits 3-0). Die nächsten 3 Byteblöcke können maximal 3 Parameter für den gewählten Verarbeitungscode darstellen, wobei jeweils das höchste Datenbit (*MSB* für *Most Significant Bit*) eines Datenbytes fix auf „0“ gesetzt ist. Dadurch ergibt sich (mit den jeweils verbleibenden Datenbits 6-0) ein Wertebereich der 3 Parameter von 0 bis 127 (siehe auch Tab. 5.1 „MIDI-Protokoll: Bytes 1..4“).

5.2. Note On/Off Nachrichten

Nach den beschriebenen Belegungen der einzelnen Bits lassen sich unterschiedliche Standardnachrichten von MIDI zusammenstellen. Die Verarbeitungscodes für *Note On* und *Note Off* sind „1001“ (0x9 Hex) respektive „1000“ (0x8 Hex). Der erste Parameter bei beiden Verarbeitungscodes gibt die Notenhöhe (*pitch*) an. Der zweite Parameter steht für die Geschwindigkeit in der z.B. eine Klaviertaste angeschlagen wurde und wird daher als Lautstärke interpretiert. Der dritte Parameter wird bei dieser Art von MIDI-Nachrichten nicht verwendet.

5.3. Zusammenfassung

Das Musical Instrument Digital Interface (MIDI) besticht durch seine einfache Art und Weise, Musikinformationen zu übertragen und ist nach über 20 Jahren nach wie vor ein wesentlicher Bestandteil der computerunterstützten, elektronischen Verarbeitung von Musik. MIDI-Nachrichten können selbst auf unterster Ebene einfach als 32-Bit-Wort zusammengestellt werden. Dazu ist lediglich das Wissen aller existierenden Verarbeitungscodes und ihren dazugehörigen Parametern erforderlich, welche übersichtlich in der MIDI-Spezifikation der *MIDI Manufacturer's Association* zusammengefasst sind [MID01].

Kapitel 6.

JNI

6.1. Einleitung

Mit der zunehmenden Popularität von Java häuften sich Situationen, in denen Entwickler gefordert waren, laufende Prozesse oder gar vollständige Applikationen mit Java-Klassen und -Objekten interkommunikativ zu verbinden. Ursachen dafür können vom einfachen Austauschen von Daten, Einbinden von bestehenden Bibliotheken aus anderen Programmiersprachen wie C oder C++ bis hin zum Ersetzen oder Erweitern von zeitkritischen Funktionen, welche spezialisiert und optimiert für bestimmte Plattformen als sogenannte *native* Prozesse entwickelt wurden, sein.

Die größten Hürden stellen dabei einerseits Javas automatisierte dynamische Speicherverwaltung und abstrahierte Datentypen und auf der anderen Seite die eingeschränkten Interaktionsmöglichkeiten mit nativen Prozessen dar. Die Java-Schnittstelle Java Native Interface (JNI) definiert und ermöglicht einen *koordinierten* und *standardisierten* Zusammenschluss von Java und nativen Anwendungen. JNI verfolgt dabei einen rudimentären Ansatz auf niedriger Stufe, wodurch es auf Softwareebene als „*low level*“ Schnittstelle kategorisiert werden kann.

JNI trägt der objektorientierten Struktur von Java Rechnung und erlaubt die entfernte Verwaltung und Bearbeitung von Objektklassen und Objektinstanzen (inklusive deren Eigenschaften und Methoden) über Prozessgrenzen. Dies ermöglicht eine Umsetzung der meisten erdenklichen Szenarien, wird jedoch mit erhöhtem Programmieraufwand wie Komplexitätsgrad erkauft.

6.2. Überblick

Für die folgende Darstellung der JNI-Grundkonzepte und -Basisfunktionalitäten werden im weiteren Verlauf unterschiedliche Komponenten der Programmiersprachen Java und C/C++

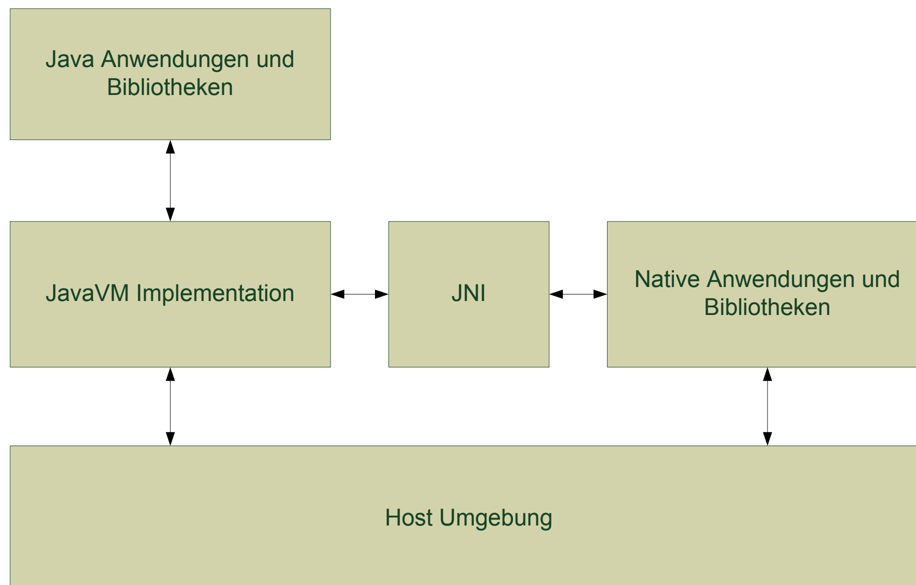


Abbildung 6.1.: Übersicht *Java*, *JNI* und *native* Anwendungen

referenziert. Für einen besseren Überblick werden die wichtigsten Komponenten in ihrem semantischen Zusammenhang vorgestellt (sinngemäß ins Deutsche übersetzt aus [Lia99]):

*Die **Java-Plattform** ist eine Programmierumgebung, die aus einer **virtuellen Java-Maschine (JavaVM)** und einer Programmierschnittstelle, dem **Java Application Programming Interface (API)**, besteht. Java Anwendungen werden in der **Java-Programmiersprache** geschrieben und in ein „maschinenunabhängiges“ Klassen-Binärformat dem **Java-Bytecode** kompiliert. Eine Klasse im Bytecode kann auf jeder Implementierung einer virtuellen Java-Maschine (JavaVM) ausgeführt werden. Das Java Application Programming Interface (API) besteht aus einer Menge vordefinierter Java-Klassen. Für jede Implementierung der Java-Plattform wird die Unterstützung der Java-Programmiersprache, einer virtuellen Java-Maschine und des APIs garantiert.*

*Eine **Host-Umgebung** ist eine Kombination aus einem Host-Betriebssystem, einer Menge an nativen Bibliotheken und einem Prozessorbefehlssatz. **Native** (systemeigene) Anwendungen werden in **nativen Programmiersprachen** wie C oder C++ geschrieben, in einen Host-spezifischen Binärcode kompiliert und an native Bibliotheken verlinkt. Native Applikationen und native Bibliotheken sind typischerweise abhängig von einer bestimmten Host-Umgebung. Eine C-Anwendung eines bestimmten Systems muss nicht notwendigerweise auch auf anderen Betriebssystemen lauffähig sein.*

Tabelle 6.1.: JNI-Typdefinitionen für Basistypen [Lia99]

Java Typ	Nativer Typ	Beschreibung
`boolean`	`jboolean`	vorzeichenlose 8 Bits
`byte`	`jbyte`	8 Bits mit Vorzeichen
`char`	`jchar`	vorzeichenlose 16 Bits
`short`	`jshort`	16 Bits mit Vorzeichen
`int`	`jing`	32 Bit mit Vorzeichen
`long`	`jlong`	64 Bit mit Vorzeichen
`float`	`jfloat`	32 Bit Fließkomma
`double`	`jdouble`	64 Bit Fließkomma

6.3. Meta-Programmiersprache

Aus der Sicht von nativen Programmiersprachen ist JNI eine Art *Meta-Programmiersprache* für Java-Objektklassen. JNI stellt der nativen Sprache eigene Strukturen und Methoden zur Verfügung, um Java-Objekte in einer JavaVM-Implementierung zu instanziieren und mit diesen Instanzen zu interagieren. Darüber hinaus können sogar neue Java-Objektklassen über Methoden von JNI in einer *beschreibenden* Art und Weise erzeugt werden.

Realisiert wird die Meta-Programmierung fast ausschließlich über *Referenzen* auf verwendete Daten und Typen. Diese Referenzen werden über eindeutige Identifikationsnummern (*Identifier*) angesprochen. Einzige Ausnahme stellen *Basistypen*[1] wie `int`, `char` und `float` dar.

6.4. Basistypen

JNI bietet für fast alle gängigen Basistypen der nativen Programmiersprache eigene, korrespondierende Typdefinitionen an, die in der C-Header-Datei `jni.h` zu finden sind (siehe auch Tab. 6.1 „JNI-Typdefinitionen für Basistypen [Lia99]“). Die Verwendung der vordefinierten Typen hat gleichzeitig einen zukunftsorientierten Aspekt, da die eigentliche Implementierung, also die Darstellung der Werte im Speicher, gekapselt wird. Ändert sich beispielsweise beim Wechsel von 32-Bit auf 64-Bit Architekturen die Implementierung der Java-Basistypen, ist nur eine Anpassung des JNI, nicht jedoch der mit JNI entwickelten Programme notwendig.

[1] Werte von Basistypen können meist direkt vom Prozessor bearbeitet und in Register abgelegt werden, daher werden sie in vielen Programmiersprachen als *value types* bezeichnet.

6.5. Referenzen auf Arrays, Strings, Objekte und Methoden

Komplexere Datenstrukturen wie *Arrays*, *Strings*, *Objekte* und *Methoden* können nicht so einfach zwischen Java-Anwendungen und nativen Anwendungen ausgetauscht werden. Ein wichtiger Grund dafür ist die dynamische Speicherverwaltung von Java, die automatisch ohne Wissen des Benutzers oder des Programmierers ihre Arbeit verrichtet. Um diese dynamische Speicherverwaltung möglichst effizient implementieren zu können, gibt es keine Garantien für fixe Speicherpositionen von Datenobjekten. Das bedeutet, die physikalische Speicherposition eines Objekts kann sich im Laufe der Anwendung kontinuierlich ändern, wenn dies für die Speicherverwaltung der virtuellen Java-Maschine von Vorteil ist. Daher ist es nicht ausreichend, der nativen Anwendung einfache Speicherreferenzen auf Java-Objekte zu übergeben. Vielmehr übernimmt JNI als Schnittstelle die Aufgabe, eine korrekte *Objektreferenzierung* zwischen der Java-Anwendung und der nativen Anwendung zu gewährleisten.

6.6. Arbeitsanweisungen

Eine native Anwendung kann folglich Verarbeitungsaufgaben wie das Erzeugen oder das Ändern von komplexen Datenstrukturen über JNI nur beantragen aber nicht selbst durchführen. So kann eine in C geschriebene Anwendung mit dem Befehl `NewIntArray` über JNI ein Array vom Typ `int` in der virtuellen Java-Maschine erzeugen, erhält jedoch nur eine symbolische Objektreferenz in Form eines int-Array *Identifiers* von `NewIntArray` zurück.

Um in weiterer Folge Array Elementen innerhalb der nativen Anwendung Werte zuweisen zu können, sind weitere JNI-Befehle notwendig. Auch hier wird wieder unterschieden zwischen der Bearbeitung von Arrays mit Basistypen und Arrays mit komplexeren Datenstrukturen. Im gewählten Beispielfall sind die Arrayelemente vom Basistyp `int`. Dadurch ist es möglich, mit dem Befehl `SetIntArrayRegion` das gesamte Array mit Werten aus einem übergebenen `int` Buffer zu initialisieren.

6.7. Klassendeskriptor und Klassenreferenzen

Werden benutzerdefinierte Java-Klassen verwendet, muss es für JNI möglich sein, aus der nativen Anwendung angeforderte Klassen eindeutig zu erkennen und zuordnen zu können. Die hierarchische Einteilung von Java, in der Pakete, Module und Objektklassen durch Punkte getrennt werden, findet unter JNI eine Entsprechung mit Schrägstrichen „/“. Somit ist für die die Objektklasse „`java.lang.String`“ unter JNI der *Klassendeskriptor* „`java/lang/String`“. Der Klassendeskriptor ermöglicht, auch Arrays von Klassen zu beschreiben, indem der Klassenbeschreibung eine eckige Klammer „`[`“ vorangestellt wird, z.B. „`[java.lang.String`“.

Tabelle 6.2.: JNI-Felddeskriptoren

Felddeskriptor	Java Typ
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

Zur Demonstration soll ein Array mit Elementen des benutzerdefinierten Typs *BeispielKlasse* aus dem Paket *BeispielPaket* erzeugt werden. Bevor nun ein Befehl für die Erzeugung eines benutzerdefinierten Elementarrays aufgerufen werden kann, muss zuerst eine *Klassenreferenz* für *BeispielKlasse* erfragt werden. JNI stellt die Funktion `FindClass` bereit, die mit der Zeichenkette „`BeispielPaket/BeispielKlasse`" die gewünschte Klassenreferenz zurück liefert. Mit dieser Referenz als Parameter kann die JNI Funktion `NewObjectArray` in der virtuellen Java-Maschine das gewünschte *BeispielKlasse* Array erzeugen. Soll hingegen ein Array von verschachtelten Arrays von *BeispielKlasse* Elementen erzeugt werden, wird `FindClass` als Array-Klassendeskriptor die Zeichenkette „`[BeispielPaket/BeispielKlasse`" übergeben.

6.8. Feld- und Methodendeskriptoren

Zur Interaktion mit Objektklassen und Objektinstanzen fehlt nur noch die Möglichkeit, Referenzen auf Objektfelder und Objektmethoden über JNI zu erfragen. Die Schwierigkeit dabei ist wieder eine eindeutige Beschreibung oder Benennung der gewünschten Felder und Methoden. Bei der Entwicklung von JNI wurde festgelegt, Felder und Methoden über ihren *Namen* in Kombination mit ihrem *Typ* eindeutig auswählen zu können. Da ein Feldtyp wiederum eine komplexe Datenstruktur sein kann, werden Feldtypen und Methodentypen ähnlich wie bei Klassenreferenzen über Zeichenketten beschrieben. Im Falle von Feldtypen werden einzelne Großbuchstaben für Basistypen (sogenannte *Felddeskriptoren*) verwendet. Für komplexe Typen werden die Klassendeskriptoren (siehe oben) mit vorangestelltem Großbuchstaben „`L`" verwendet (siehe Tab. 6.2 „JNI-Felddeskriptoren" und Abschnitt 6.10 „Natives Programmieren mit JNI").

Bei *Methodendeskriptoren* müssen über eine Zeichenkette der Rückgabetyp und die Typen der Parameter der Methode angegeben werden. Per Definition besteht ein Methodendeskriptor aus den Parametertypen angegeben in 2 runden Klammern und dem Rückgabetyp am Ende. Als Typen können wieder Feld- oder Klassendeskriptoren verwendet werden. Zu beachten ist, dass Klassendeskriptoren zur Begrenzung mit einem Semikolon „`;`" abgeschlossen werden müssen.

Als Beispiel wird eine Methode *method* mit Rückgabetyp `String[]` und den Parametern `int i` und `BeispielKlasse beispiel` deklariert:

```
String[] method( int i, BeispielKlasse beispiel );
```

Der dazu passende Methodendeskriptor ist:

```
"(ILBeispielPaket/BeispielKlasse;)Ljava/lang/String;"
```

6.9. Objektorientiertes JNI

Das JNI schafft den großen Spagat, eine Schnittstelle für objektorientiertes Java und ANSI C zu bilden. Dies wurde möglich, da in JNI intern objektorientiert gearbeitet wird und die C-Funktionen der Schnittstelle immer in einem objektorientierten Kontext interpretiert werden. Dieser Umstand ist gut an der Schnittstellenstruktur *JNIEnv* erkennbar. *JNIEnv*-Instanzen speichern Zustandsdaten und Objektreferenzen einer aktiven JNI-Umgebung und sind damit die Basis aller nativen JNI-Programme. Aufgerufene JNI-Befehle sind immer als Methoden einer Instanz von *JNIEnv* zu sehen.

Für objektorientiertes C++ ist diese Eigenschaft ohne Probleme umgesetzt worden, wie folgendes Quellcode-Beispiel demonstriert:

```
1 JNIEnv* environment;
2 ...
3 jintArray arr = environment->NewIntArray( 3 );
```

Unter ANSI C werden, wie im nächsten Beispiel gezeigt, Methoden mit ihren zugehörigen *JNIEnv* Instanzen als erster Parameter zur Definition des objektorientierten Kontextes aufgerufen:

```
1 JNIEnv* environment;
2 ...
3 jintArray arr = (*environment)->NewIntArray( environment, 3 );
```

6.10. Natives Programmieren mit JNI

Wie die vorangegangenen Abschnitte gezeigt haben, ist die Programmierung mit JNI mit einem erheblichen Mehraufwand verbunden. Eine einfache Zuweisung an eine Eigenschaft einer Objektinstanz kann in bestimmten Fällen vier Arbeitsschritte erfordern, wie das Quellcodebeispiel 6.1 zeigt.

Listing 6.1: JNI Programmieren in C++

```
void resetCounter( JNIEnv* env, jobject object )
{
  jfieldID fieldID; // speichert den Identifier einer Objekteigenschaft
  jint counter;     // int Basistyp von jni.h

  // Abfragen der Klassenreferenz des erhaltenen Objekts
  jclass objectClass = env->GetObjectClass( object );

  // Abfragen des Identifiers der Eigenschaft 'Counter' vom Typ int
  fieldID = env->GetFieldID( objectClass, "Counter", "I" );

  // Auslesen des Wertes der Eigenschaft 'Counter'
  counter = env->GetIntField( object, fieldID );

  counter = 0;

  // Zuweisen des geänderten Wertes an 'Counter'
  env->SetIntField( object, fieldID, counter );
}
```

6.11. Zusammenfassung

Das JNI löst die Problematik, objektorientierte Strukturen inklusive ihrer Eigenschaften und Methoden über „*low level*“ Schnittstellen auszutauschen. Im Fall von JNI ist die Schnittstelle auf Programmierebene für Java-Applikationen und native Anwendungen entwickelt worden. Vor allem im Bereich des Internets sind in den letzten Jahren ähnliche Herausforderungen häufig bearbeitete Wissenschaftsthemen gewesen und sind es teilweise noch heute. Allgemeine Protokolle wie das Simple Object Access Protocol (SOAP), das auf der eXtensible Markup Language (XML) als Austauschmedium basiert, ermöglichen einen Austausch von Objekten ähnlich einem JNI. Diese *Web-Protokolle* bieten jedoch meist zusätzliche spezielle Entwicklungstools, die den eigentlichen Aufwand beim praktischen Einsatz der Protokolle stark reduzieren können.

Ein Ausblick diesbezüglich wäre eine Adaption von JNI in Richtung SOAP. Der inhärente, unter anderem auch durch XML verursachte Mehraufwand von SOAP beim Austausch von Objekten ist jedoch nicht zu vernachlässigen. Eine einfachere, kurzfristige Möglichkeit zur Erleichterung der Anwendung von JNI wäre die Entwicklung von JNI-spezifischen Quellcodepräprozessoren. Nach dem Vorbild der Web-Protokoll-Tools kann dem Entwickler die Programmierung von Standardkonstrukten abgenommen oder erleichtert werden.

Teil II.

Prototyp

Kapitel 7.

Planung des Prototyps

7.1. Festlegung der Entwicklungsumgebung

7.1.1. Windows DLL

Beim Design des Re-Compose Prototyps waren die meisten Rahmenbedingungen von Beginn an eindeutig gesetzt. Der Haupteinsatzzweck des Prototyps war als VST-Plug-in für beliebige VST-Host-Anwendungen vorgegeben, somit ist das Re-Compose Modul als dynamische Bibliothek mit speziellen VST-Vorgaben zu entwickeln gewesen (vgl. Abschnitt 4.4 „Kommunikationsaufbau“). Windows wurde als einzige Zielplattform gewählt, um den Mehraufwand einer Multiplattformlösung zu Gunsten einer priorisierten Entwicklung in den Bereichen der *VST*, *JNI* und *Re-Compose Logik*-Anbindungen einzusparen. Somit war als System-Programmierschnittstelle das Windows Application Programming Interface (API) *Win32* vorgegeben.

7.1.2. VST SDK Version 2.4

Obwohl zu Beginn des Prototypdesigns *Steinberg Media Technologies GmbH* eine für MIDI-Anwendungen passendere VST-Schnittstelle mit Namen *VST Module Architecture (MA)* oder auch eine neuere Version ihres VST SDKs mit Versionsnummer 3.0 angeboten hatte, ist letztendlich das VST SDK in Version 2.4 die einzige Alternative für eine zweckmäßige Umsetzung des Prototyps geblieben. Die neueren Technologien hätten zwar eine modernere Implementierung ermöglicht, jedoch fehlen in der Praxis noch die geeigneten VST-Hosts, die diese Technologien unterstützen.

7.1.3. JNI und JDK Version 6.0

Das JNI ist Bestandteil des Java Development Kits (JDK) seit Version 4.0. Für die Entwicklung des Prototyps konnte ohne Restriktionen das JDK von *Sun Microsystems, Inc.* in Version 6.0 verwendet werden. Der JNI-Teil des Prototyps ist dahingehend auf virtuellen Java-Maschinen (JavaVM) ab Version 4.0 lauffähig.

7.2. Identifikation der externen Komponenten

Der Re-Compose Prototyp ist als VST-Plug-in ein Teil eines *VST-Hosts*. Der VST-Host wird über die *Win32 API* auf einem *Windows-Host-System* ausgeführt. Der Prototyp initialisiert über das *JNI* eine *virtuelle Java-Maschine*, in der die Re-Compose Logik gestartet und betrieben wird. Die Abbildungen 7.1 „Übersicht Prototypdesign JNI“ und 7.2 „Übersicht Prototypdesign VST“ als Ergebnis des Designprozesses veranschaulichen den Zusammenhang des Prototyps und dessen identifizierten externen Komponenten in Form von Schichtenmodellen.

7.3. Strukturierung des Prototyps

7.3.1. Einteilung in Module

Nach der Identifikation aller externen Komponenten und deren Rollen im Gesamtsystem kristallisierten sich zwei mögliche Strukturen für den internen Aufbau des Prototyps heraus:

- Eine *serviceorientierte* Architektur *(SOA)*, bei der Module aus benötigten abgeschlossenen Diensten oder Funktionen abgeleitet werden (Beispielsweise ein Modul für das Empfangen von VST-Nachrichten).

- Eine klassische *objektorientierte* Architektur *(OOA)*, die passend zu den erkannten externen Komponenten abhängig vom Komplexitätsgrad eine oder mehrere Objektklasse(n) vorsieht. Diese Klassen werden nach dem Vorbild der externen Komponenten über deren Methoden miteinander verbunden.

Bei Planungsbeginn der inneren Struktur des Prototyps konnten die externen Komponenten sehr gut in mehrere Einzelmodule eingeteilt werden. Darauf aufbauend wurde für die Prototypstruktur eine *objektorientierte* Architektur gewählt. Das Hauptaugenmerk wurde dabei auf eigenständige, von einander unabhängige Module gelegt. Abgeleitet von den beschriebenen externen Komponenten konnten somit folgende benötigte Module im Kontext ihrer Zusammenarbeit definiert werden (siehe auch Abb. 7.3 „Übersicht Prototypmodule“):

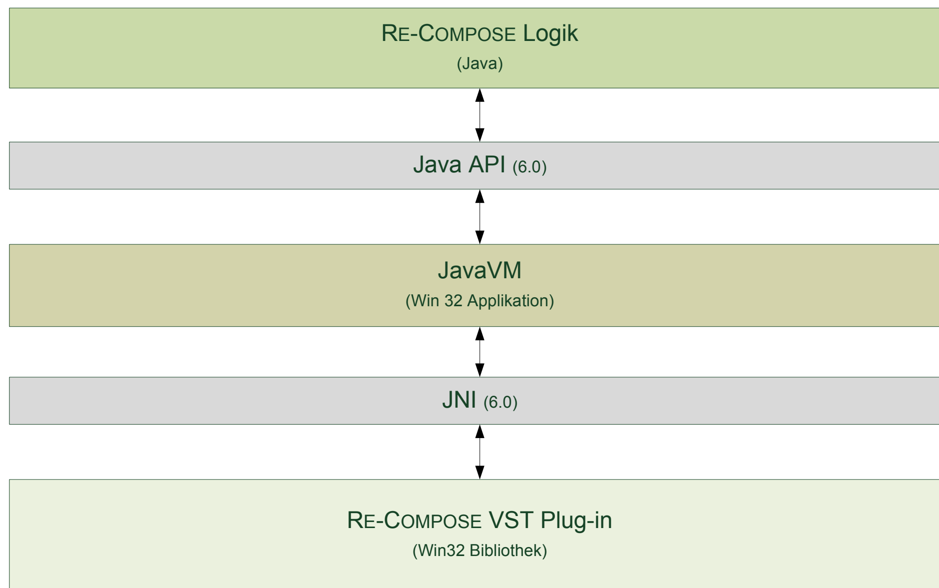


Abbildung 7.1.: Übersicht Prototypdesign JNI

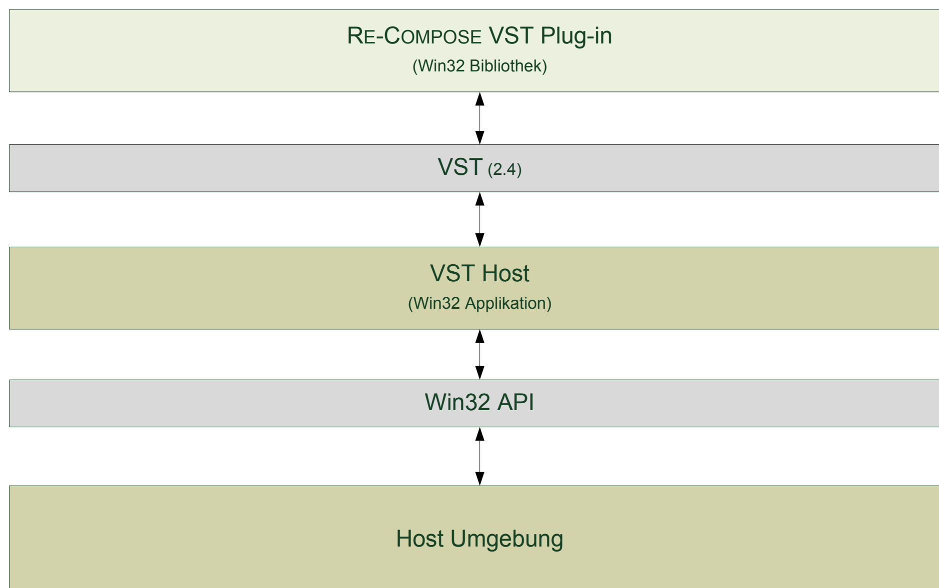


Abbildung 7.2.: Übersicht Prototypdesign VST

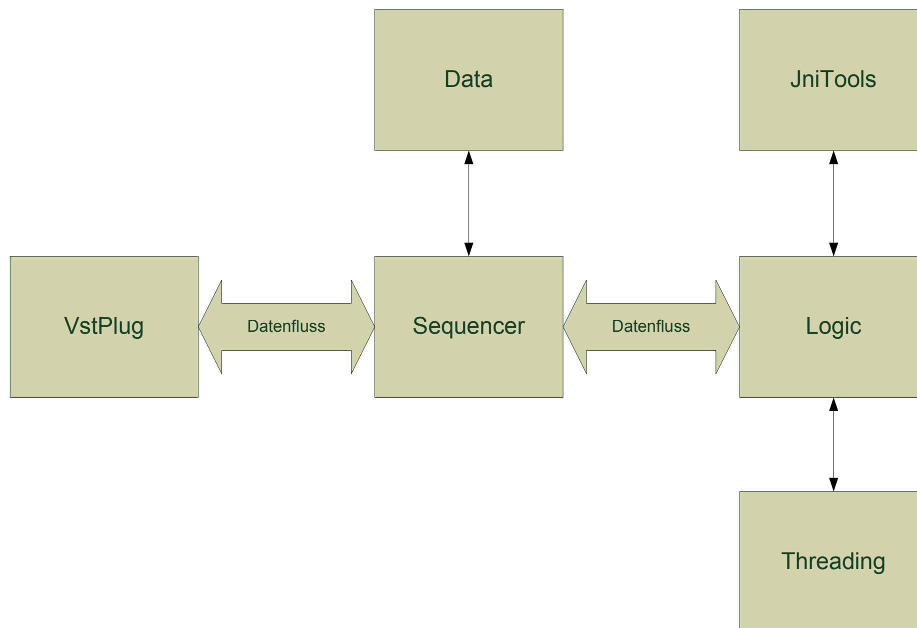


Abbildung 7.3.: Übersicht Prototypmodule

VstPlug ist hauptverantwortlich für die Implementierung der VST-Plug-in-Schnittstelle und wickelt den Datenaustausch mit den VST-Hosts ab.

Data stellt Strukturen zum Speichern von Musikdaten in unterschiedlichen Zeitangaben[1] zur Verfügung und kann Konversionen von einem Zeitformat in beliebig andere durchführen.

Sequencer speichert eine Folge von Musikdaten in ihrem zeitlichen Zusammenhang. Die Musiksequenz kann als eine Datenstruktur bearbeitet oder in einem zeitlichen Kontext korrekt wiedergegeben werden.

Logic kapselt die Kommunikation mit der Re-Compose Logik über das JNI.

JniTools stellt Hilfsmethoden unter anderem zur Instanziierung einer *virtuellen Java-Maschine* bereit.

Threading definiert objektorientierte Klassen, in denen *Multithreading*- und *Critical-Section*-Funktionalitäten des *Win32 APIs* gekapselt werden.

[1] Zeitangaben siehe auch Kapitel 3 „Zeitbegriffe in der Musik“.

7.3.2. Einteilung in Objektklassen

Die beschriebenen Module sind nach deren Definition in weitere Hierarchiestufen verfeinert worden und dadurch in passende Objektklassen unterteilt worden. Für Module, deren Funktionalitäten gut überschaubar und einigermaßen atomar waren, genügte meist die Planung einer einzigen Klasse. Abstraktere Module bekamen hingegen eine Aufteilung in mehrere Objektklassen. Der Bereich der Re-Compose Logikanbindung stellte den komplexesten Teil des Prototyps dar, insbesondere da dieser als *Multithread*-System konzipiert werden musste. In den nächsten Abschnitten werden alle Klassen geordnet nach Modulzugehörigkeit aufgezählt und kurz beschrieben. Eine Gesamtübersicht des zu Grunde liegenden Klassenmodells gemäß UML-Notation findet sich in Abbildung 7.4.

7.4. Beschreibung der Objektklassen

7.4.1. VstPlug

VstPlug::VstPlug

- Die Klasse *VstPlug::VstPlug* ist gleichermaßen von *AudioEffectX* des VST SDK und von *Threading::Object* abgeleitet (*multiple inheritance*).
- *VstPlug* übernimmt die komplette Implementierung der VST-Plug-in-Schnittstelle und ist somit für die Kommunikation mit dem VST-Host zuständig.
- *VstPlug::VstPlug* instanziiert die Klassen *Sequencer::Sequencer*, *Data::Note*, *Data::Timing* und erbt Funktionen zur Synchronisation von *Threading::Object*.
- Die Objektklasse *VstPlug::VstPlug* empfängt über die VST-Methode *processEvents* die laufenden *MIDI-Events* und gibt diese zur Weiterverarbeitung an *Sequencer::Sequencer* weiter.
- *VstPlug::VstPlug* tauscht dazu mit *Sequencer::Sequencer* eine Liste von *Data::Note* Instanzen aus.

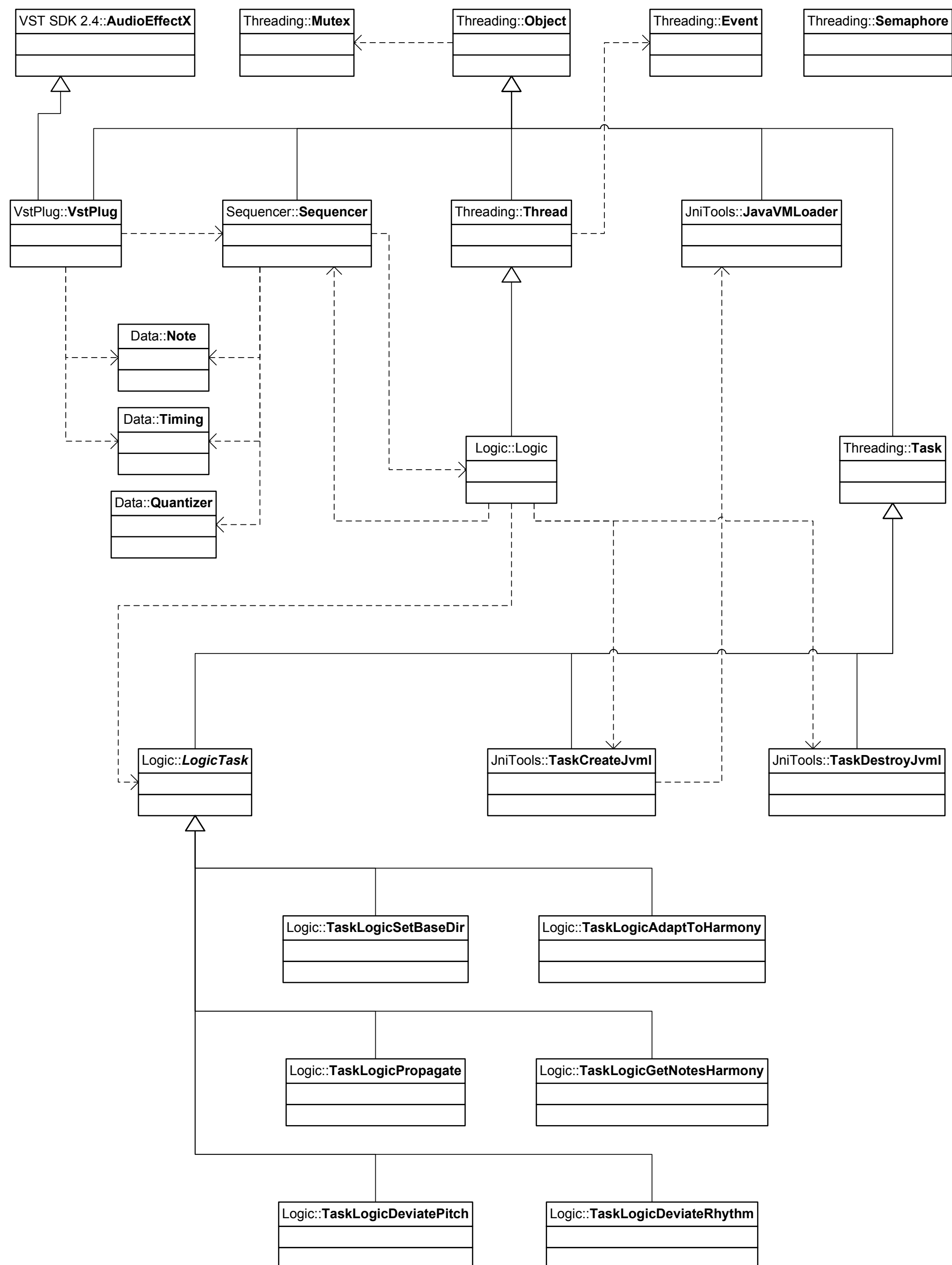


Abbildung 7.4.: UML-Klassendiagramm des Prototyps

7.4.2. Sequencer

Sequenzer::Sequencer

- Die *Sequencer::Sequencer* Klasse implementiert eine grundlegende Version eines Sequenzers.
- Eine Folge von Noten wird zeitlich korrekt abgespeichert, wiedergegeben oder als Datenstruktur im proprietären Musikdatenformat des Re-Compose Logikmoduls exportiert.
- *Sequencer::Sequencer* instanziiert die Klassen *Logic::Logic*, *Data::Note*, *Data::Timing*, *Data::Quantizer* und erbt Funktionen zur Synchronisation von *Threading::Object*.
- Je nach Benutzerinteraktion ruft *Sequencer::Sequencer* Methoden seiner *Logic::Logic*-Instanz zum Bearbeiten/Verändern der aktuellen Notensequenz auf.
- *Sequencer::Sequencer* tauscht dazu mit *Logic::Logic* eine Liste von *Data::Note* Instanzen aus.

7.4.3. Data

Data::Timing

- Die *Data::Timing* Klasse kann mit Zeitangaben aus unterschiedlichen Zeitzählsystemen der Musik initialisiert werden und liefert die dazugehörigen konvertierten Werte aller implementierten Zeitzählsysteme zurück.

Data::Quantizer

- Die Klasse *Data::Quantizer* implementiert eine einfache Quantifizierung von Noten. Das bedeutet, dass der zeitliche Beginn einer Note gemäß einer gewählten minimalen Auflösung auf das nächstgelegene ganzzahlige Vielfache der Auflösung verschoben wird (siehe auch Gloss. Quantizer).

Data::Note

- *Data::Note* speichert die nötigsten Daten einer Musiknote: *Tonhöhe*, *Startzeit*, *Endzeit* und optional die *Schlaglänge*, um die Notenlänge im Takt bestimmen zu können.

7.4.4. Logic

Logic::Logic

- Die Klasse *Logic::Logic* implementiert den eigentlichen Kommunikationsablauf mit dem Re-Compose Logikmodul.
- *Logic::Logic* ist von der Klasse *Threading::Thread* abgeleitet und erbt davon die Funktionalität in einem eigenen *Thread* abzulaufen.
- *Logic::Logic* instanziiert die Klasse *JniTools::TaskCreateJvml*, mit der gemeinsam eine *virtuelle Java-Maschine* innerhalb des separierten *Threads* gestartet wird.
- Die Klasse *Logic::Logic* fungiert als *Dispatcher* für abgeleitete Instanzen der abstrakten Basisklasse *Logic::LogicTask*.
- Von *Logic:LogicTask* abgeleitete Klassen beschreiben Aufgaben, die von der Logik zu bearbeiten sind (siehe auch *Logic:LogicTask*).
- *Logic::Logic* empfängt diese Aufgaben in ihrem eigenen *Thread* und führt daraufhin im *first in - first out (FIFO)* Verfahren die korrespondierenden Schnittstellenmethoden der Re-Compose Logik in der *virtuellen Java-Maschine* aus.

Logic::LogicTask

- Die *abstrakte* Klasse *Logic::LogicTask* implementiert allgemeine Funktionen, die für die Kommunikation von Arbeitsbeschreibungen an den eigenen Logik-*Thread* erforderlich sind.
- *Logic::LogicTask* ist von der ebenfalls abstrakten Klasse *Threading::Task* abgeleitet.
- Für die konkreten Kindklassen von *Logic:LogicTask* ist es erforderlich, die virtuelle Methode *process* von *Threading::Task* zu implementieren.

Logic::TaskLogicAdaptToHarmony

- *Logic::TaskLogicAdaptToHarmony* ist abgeleitet von *Logic::LogicTask*.
- *Logic::TaskLogicAdaptToHarmony* Instanzen werden vom *Dispatcher* von *Logic::Logic* übernommen und aktivieren den Re-Compose Prozess „Adapt to Harmony“ im Re-Compose Logikmodul.

Logic::TaskLogicDeviatePitch

- *Logic::TaskLogicDeviatePitch* ist abgeleitet von *Logic::LogicTask*.
- *Logic::TaskLogicDeviatePitch* Instanzen werden vom *Dispatcher* von *Logic::Logic* übernommen und aktivieren den Re-Compose Prozess „Deviate Pitch“ im Re-Compose Logikmodul.

Logic::TaskLogicDeviateRhythm

- *Logic::TaskLogicDeviateRhythm* ist abgeleitet von *Logic::LogicTask*.
- *Logic::TaskLogicDeviateRhythm* Instanzen werden vom *Dispatcher* von *Logic::Logic* übernommen und aktivieren den Re-Compose Prozess „Deviate Rhythm“ im Re-Compose Logikmodul.

Logic::TaskLogicGetNotesHarmony

- *Logic::TaskLogicGetNotesHarmony* ist abgeleitet von *Logic::LogicTask*.
- *Logic::TaskLogicGetNotesHarmony* Instanzen werden vom *Dispatcher* von *Logic::Logic* übernommen und erfragen die Akkordnoten zu gegebenen Harmonien bei der Re-Compose Logik.

Logic::TaskLogicPropagate

- *Logic::TaskLogicPropagate* ist abgeleitet von *Logic::LogicTask*.
- *Logic::TaskLogicPropagate* Instanzen werden vom *Dispatcher* von *Logic::Logic* übernommen und aktivieren den Re-Compose Prozess „Propagate“ im Re-Compose Logikmodul.

Logic::TaskLogicSetBaseDir

- *Logic::TaskLogicSetBaseDir* ist abgeleitet von *Logic::LogicTask*.

- *Logic::TaskLogicSetBaseDir* Instanzen werden vom *Dispatcher* von *Logic::Logic* übernommen und geben dem Re-Compose Logikmodul ein zu setzendes Basisverzeichnis vor.

7.4.5. JniTools

JniTools::JavaVMLoader

- Instanzen von *JniTools::JavaVMLoader* erzeugen *virtuelle Java-Maschinen* im aktuell laufenden *Thread*. Dazu wird in der Windows-Registrierung des Hostsystems eine Referenz der installierten Java-Laufzeitumgebung gesucht und deren Versionsnummer auf Gültigkeit überprüft.

- Die Klasse *JniTools::JavaVMLoader* wird von der Klasse *Threading::Object* abgeleitet und erbt deren Funktionen zur Synchronisation.

JniTools::TaskCreateJvml

- *JniTools::TaskCreateJvml* ist von der abstrakten Klasse *Threading::Task* abgeleitet.

- *JniTools::TaskCreateJvml* instanziiert in der virtuellen, ererbten Methode *process* die Klasse *JniTools::JavaVMLoader* und erzeugt eine *virtuelle Java-Maschine*.

JniTools::TaskDestroyJvml

- *JniTools::TaskDestroyJvml* ist von der abstrakten Klasse *Threading::Task* abgeleitet.

- *JniTools::TaskDestroyJvml* wird von *Logic::Logic* mit einer Referenz auf eine laufende *virtuelle Java-Maschine* instantiiert.

- *JniTools::TaskDestroyJvml* implementiert die virtuelle, ererbte Methode *process*, in der die übergebene *virtuelle Java-Maschine* heruntergefahren und geschlossen wird.

7.4.6. Threading

Threading::Semaphore

- Die Klasse *Threading::Semaphore* kapselt *Win32-API*-Anweisungen zum Setzen und Zurücksetzen von Semaphoren.

Threading::Mutex

- Die Klasse *Threading::Mutex* kapselt *Win32-API*-Anweisungen zum Setzen und Zurücksetzen von Mutex-Werten.
- *Threading::Mutex* wird instanziiert in der Klasse *Threading::Object*. *Threading::Objekt* realisiert damit eine Synchronisation der eigenen Instanzen, die an alle abgeleiteten Klassen von *Threading::Object* weitervererbt wird.

Threading::Event

- Die Klasse *Threading::Event* kapselt *Win32-API*-Anweisungen zum Senden und Empfangen von System*events*.
- *Threading::Event* wird instanziiert von *Threading::Thread*. Dadurch ist es Instanzen von *Threading::Thread* möglich, sich gegenseitig bestimmte Ereignisse zu signalisieren, wie zum Beispiel das Eintreffen von neuen *Threading::Task* Instanzen, die abgearbeitet werden sollen.

Threading::Task

- Die *abstrakte* Klasse *Threading::Task* stellt einen Arbeitsauftrag für *Threading::Thread* Instanzen dar.
- Die virtuelle, abstrakte Methode *process* muss von abgeleiteten Klassen implementiert werden.
- *Threading::Thread* Instanzen nehmen abgeleitete *Threading::Task* Instanzen entgegen und führen deren *process* Methode aus, wenn der dementsprechende *Task* aktiv wird.

Threading::Thread

- Die Klasse *Threading::Thread* kapselt *Win32-API*-Anweisungen zum Starten eines losgelösten *Threads* (genannt *child process*).

- Ein *Thread* ist in einem Ruhezustand und wartet, solange er keine Aufträge bekommt. Erhält ein *Thread* eine von *Threading::Task* abgeleitete Instanz, reiht er diese in eine Liste von empfangenen Aufträgen ein.

- Wird ein Thread durch Betriebssystem und Prozessor aktiv, überprüft die Thread-Instanz die eigene Auftragsliste und arbeitet die erhaltenen Aufgaben der Reihe nach im *first in - first out (FIFO)* Verfahren ab (siehe auch 7.4.4 „Logic::Logic“).

7.5. Zusammenfassung

Der in der Designphase erarbeitete und dargestellte Zusammenhang des Re-Compose Prototyps mit seinen externen Komponenten konnte gut in eine korrespondierende innere Struktur eingeteilt in Module *VstPlug*, *Data*, *Sequencer*, *Logic* und *JniTools* umgesetzt werden. Die Objektklassen der einzelnen Module sind dabei in einem überschaubaren Rahmen geblieben. Zur weiteren Veranschaulichung des Aufbaus des Re-Compose Prototyps wurden schematische Designdiagramme und ein UML-Klassendiagramm präsentiert.

Kapitel 8.

Implementierungsdetails

8.1. VST-MIDI-Anbindung

Ab VST Version 2.0 können MIDI-Ereignisse von VST-Hosts an VST-Plug-ins übermittelt werden. Für diesen Vorgang wurde der Plug-in-Verarbeitungscode „`effProcessEvents`" definiert. Das VST SDK deklariert in der abstrakten Klasse *AudioEffectX* die Servicemethode *processEvents*, die bei Hostnachrichten mit Verarbeitungscode `effProcessEvents` aufgerufen wird. Im Prototyp wird diese virtuelle Methode überschrieben, um den ankommenden MIDI-Datenstrom auf Notenereignisse (*Note On/Note Off*) zu filtern (siehe auch Abschnitt 5.1 „MIDI-Protokoll").

8.2. Handhabung der Zeiteinheiten

8.2.1. VST Timing

Bei der Verarbeitung von MIDI-Ereignissen ist die Art und Weise zu berücksichtigen, wie VST deren Zeitangaben handhabt. In der VST-Datenstruktur *VstMidiEvent* ist das Feld `deltaFrames` für die zeitliche Positionierung des Ereignisses vorgesehen. Dabei ist `deltaFrames` eine *relative* Zeitangabe, die in Zusammenhang mit der aktuellen (globalen) Position (*samplePos*) gebracht werden muss.

8.2.2. Re-Compose Timing

Die Re-Compose Logik empfängt und verarbeitet intern die Musikdaten in der Zeiteinheit *pulses per quarter (ppq)*. Daher ist im Datenfluss des Prototyps eine Konversion zwischen den Zeitangaben *Sampleframe* und *ppq* erforderlich. Zur Veranschaulichung sei folgendes Beispiel

zur Konvertierung von *Sampleframe* in *ppq* präsentiert (siehe auch Kapitel 3 „Zeitbegriffe in der Musik“).

$$LaengeFrames = SamplePos2 - SamplePos1 + deltaFrames2 - deltaFrames1 \quad (8.1)$$

$$LaengePPQ = \frac{LaengeFrames * BPMs}{60s * PPQRes * Samplerate} \quad (8.2)$$

8.2.3. Konversionsbeispiel

1. Die Geschwindigkeit des Musikstücks beträgt 120 *BPMs*. Die Samplerate beträgt 44100 *Hz*.

2. Die globale Position *SamplePos1* des VST-Hosts beträgt 66150 *Sampleframes*.

3. Die Methode `processEvents` übermittelt 1 MIDI-Ereignis: *Note On* mit `deltaFrames` gleich 0 (d.h. die Note beginnt mit *Sampleframe*-Nummer 66150).

4. Die globale Position ändert sich auf *SamplePos2* gleich 85769 *Sampleframes*.

5. Die Methode `processEvents` übermittelt 1 MIDI-Ereignis: *Note Off* mit `deltaFrames` gleich 2431 (d.h. die Note endet mit *Sampleframe*-Nummer 88200).

6. Aus beiden Ereignissen soll die *Notenlänge* der gesendeten Note ermittelt werden. Dies entspricht einer Konversion in ppq mit einer *Pulses-Per-Quarter*-Auflösung (*PPQRes*) gleich 1, das bedeutet 1 Impuls = 1 Viertelnote.

7. In Gleichung 8.2 eingesetzt ergibt die Beispielkonversion als Ergebnis genau eine Viertelnote.

8.3. Multithreading

Wie in den vorangegangenen Abschnitten aufgezeigt, ist der Re-Compose Prototyp als *Multithread*-Applikation implementiert und verwendet dazu Mechanismen zur Prozess-Synchronisation gekapselt in eigenen Objektklassen. Für VST-Plug-ins ist es erforderlich, in Form einer *Multithread*-Bibliothek entwickelt zu werden, damit sie in eigenen Threads parallel zur Host-Anwendung ablaufen und den Host in seiner Ausführung nicht behindern. Dennoch ist es meistens ausreichend, VST-Plug-ins intern als einfache Einzelprozesse anzusehen und dementsprechend simpel umzusetzen. Beim Re-Compose Prototyp konnte diese vereinfachte Sichtweise nicht angewandt werden. Der Zusammenschluss von VST und JNI machte eine Implementierung verteilt auf Threads inklusive Synchronisationsstrukturen erforderlich. Dieser Umstand soll im nächsten Abschnitt genauer betrachtet werden.

8.3.1. VST Thread vs. JNI Thread

VST-Host-Implementierungen haben beim Einbinden von VST-Plug-ins freie Hand, in welche Prozess-Struktur die Plug-ins integriert werden. Im idealen Fall erzeugen Hosts für jedes Plug-in einen eigenen Thread für dessen Abarbeitung. Dass auch in der Praxis immer der *selbe* Thread[1] für ein und dasselbe Plug-in zuständig ist, muss im Gegensatz dazu nicht garantiert werden. Es ist daher genauso gut möglich, dass Hosts nur eine limitierte Anzahl an Threads in einer Art Ringpuffer betreiben, die je nach Auslastung zum Abarbeiten von mehreren aktiven Plug-ins zuständig sind.

Eine über JNI instanziierte *virtuelle Java-Maschine (JavaVM)* ist grundsätzlich an den Thread, in dem sie erzeugt worden ist, gebunden. Eine JavaVM-Instanz verweigert den Zugriff aus „fremden“ Threads. Aus diesem Grund müssen „fremde“ Threads erneut eine Referenz auf die laufende *virtuelle Java-Maschine* erfragen, bevor sie diese verwenden können.

Durch die Kombination beider Schnittstellen innerhalb des Re-Compose Prototyps ergibt sich im schlechtesten Fall die Möglichkeit, dass das interne VST-Plug-in-Modul durch stetig wechselnde Verarbeitungsthreads aktiviert wird und sich in weiterer Folge ohne adäquate Vorbereitungsschritte nicht mit einer bereits laufenden JavaVM-Instanz austauschen kann.

Um den Anforderung beider Schnittstellen zu genügen wird die Klasse *Logic::Logic* von *Threading::Thread* abgeleitet. *Logic::Logic* kann dadurch für den Betrieb einer *virtuellen Java-Maschine* einen eigenen *Logic*-Thread erstellen und verwalten. So kann die Klasse *Logic::Logic* immer innerhalb desselben Threads direkt mit dieser *virtuellen Java-Maschine* über JNI kommunizieren. Somit kommt es auch zu keinen Problemen mehr, wenn das interne VST-Modul eigentlich durch unterschiedliche Threads des VST-Hosts aktiviert wird.

Diese Auslagerung des JNI-Teils von *Logic::Logic* in einen eigenen Thread erfordert die Implementierung eines asynchronen Nachrichtensystems, um Arbeitsaufträge an diesen Thread übermitteln zu können. Nach dem *Producer/Consumer*-Prinzip werden an *Logic::Logic* über die von *Threading::Thread* ererbte Methode `dispatch` Arbeitsbeschreibungen in Form von *Threading::Task*-Kindobjekten gesendet. Die abstrakte Klasse *Threading::Task* vererbt die virtuelle, abstrakte Methode `process`, die von ihren Nachfahren mit den jeweiligen Arbeitsschritten implementiert wird.

8.4. Prozessablauf

Ein UML-Sequenzdiagramm eines Beispielprozesses soll den Ablauf und das Zusammenspiel der eigenständigen Threads genauer illustrieren. Das Beispiel zeigt die Re-Compose Prototypmodule *Sequencer*, *Logic* und den von *Logic* erzeugten Kindprozess *LogicJniThread*. Im

[1] Threads werden in diesem Zusammenhang eineindeutig über ihre Prozessnummer identifiziert.

Beispiel wird *Logic* von *Sequencer* mit einer „Noten-zu-Harmonien“-Adaption beauftragt. Diese Anfrage ist die erste an *Logic*, somit muss von *Logic* zuerst eine *virtuelle Java-Maschine* im *LogicJniThread* gestartet werden. *Logic* erzeugt dazu eine Instanz von *TaskCreateJvml*, übergibt diesen *Task* mit `dispatch` an *LogicJniThread* und wartet, bis dieser fertig bearbeitet wurde. Nach Beendigung von *taskCreateJvml* ist auch der Start der *virtuellen Java-Maschine* abgeschlossen. Nun erzeugt *Logic* den Arbeitsauftrag *taskAdaptToHarmony* und übergibt diesen wieder an *LogicJniThread*. Ist *taskAdaptToHarmony* fertig abgearbeitet, kann *Logic* das Ergebnis mit `getResult` abfragen und schließlich an *Sequencer* zurückgeben (siehe auch Abb. 8.1 „UML-Sequenzdiagramm des Prototyps“).

8.5. Zusammenfassung

Das Zusammenspiel von VST und JNI erforderte die Implementierung eines Kindprozesses für die Ausführung der *virtuellen Java-Maschine*. Dadurch mussten Methoden zur Prozess-Synchronisation entwickelt werden, die im hinzugefügten Modul *Threading* zusammengefasst wurden. Zur weiteren Veranschaulichung der Funktionsweise des Re-Compose Prototyps wurde ein UML-Sequenzdiagramm präsentiert.

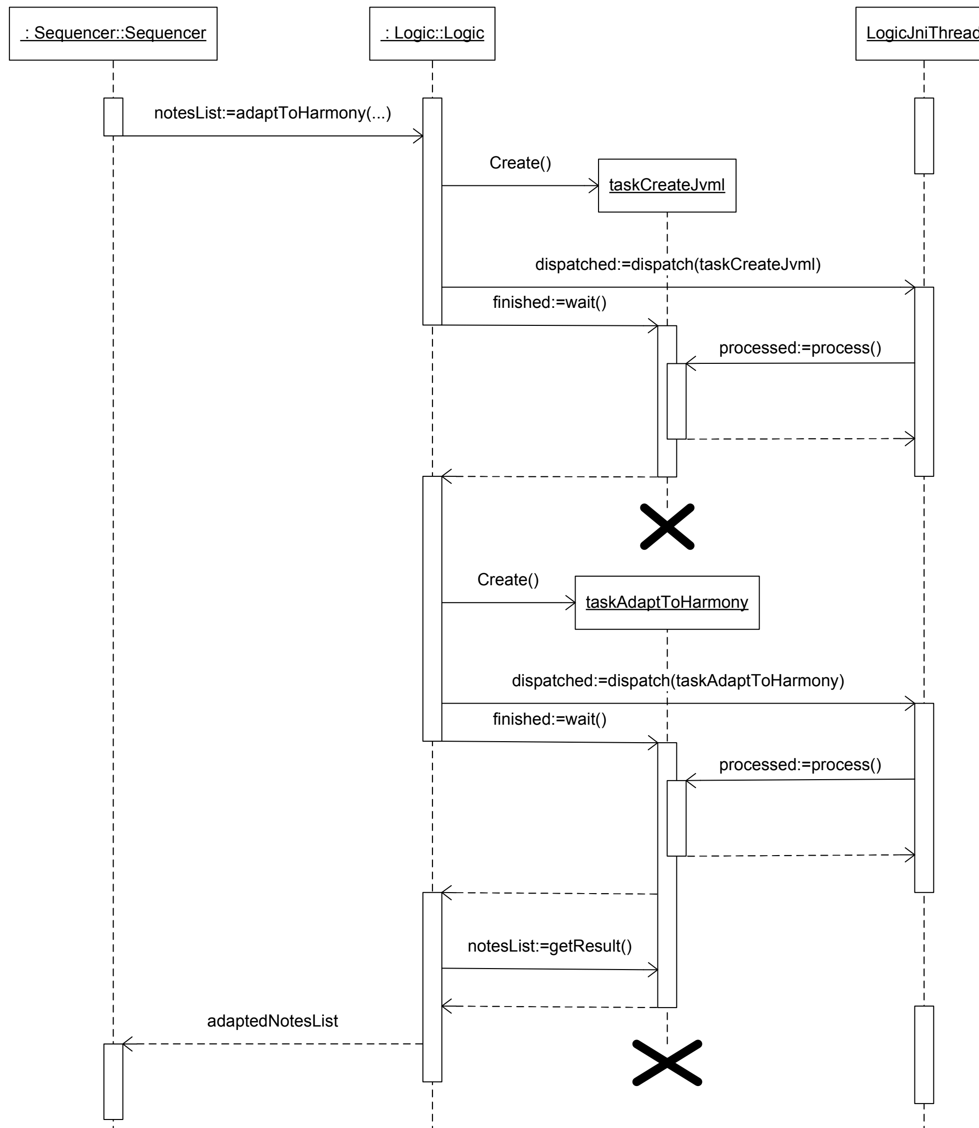


Abbildung 8.1.: UML-Sequenzdiagramm des Prototyps

Kapitel 9.

Design der Visualisierung

9.1. Einleitung

Mit der Entwicklung des Re-Compose Prototyps ist der erste Grundstein gelegt worden, die in Java entwickelte Re-Compose Logik über ein VST-Plug-in als Schnittstelle mit VST-Host-Anwendungen zu verbinden. Dadurch soll es ermöglicht werden, ganze Lieder oder Teile von Musikstücken in die Re-Compose Logikkomponente zu übertragen und auf hierarchische Strukturen analysieren zu lassen. Im weiteren Verlauf des gesamten Re-Compose Projektes wird es daher erforderlich, dem Benutzer die Ergebnisse der Logikanalyse in geeigneten Darstellungen zu präsentieren und ihm als weiteren Schritt Werkzeuge zur interaktiven Bearbeitung dieser Visualisierungen bereit zu stellen. In den folgenden Abschnitten finden sich erste Planungen und Überlegungen zur Realisierung der grafischen Präsentation.

9.2. Hierarchische Analyse

Eine Kernkomponente des Re-Compose Logikmoduls ist die *hierarchische Analyse*. Diese Analyse erstellt für eine Einzelspur eines Musikstücks eine hierarchische Struktur basierend auf vordefinierte, wiederkehrende Muster.

9.2.1. Hierarchische Darstellung

Bei der *hierarchischen Analyse* ist die klare Hauptanforderung an eine geeignete Visualisierung die intuitive Repräsentation der jeweiligen Hierarchiestrukturen. Die größte Herausforderung dabei ist eine einheitliche und übersichtliche Darstellung für Musiksequenzen mit ganz unterschiedlichen Sequenzlängen. Als erster Ansatz wird die hierarchische Struktur dem Benutzer in einer *normierten* Anzeige präsentiert. Es soll damit die hierarchische Information einer Musikspur im Gegensatz zur ihrer quantitativen Länge hervorgehoben werden.

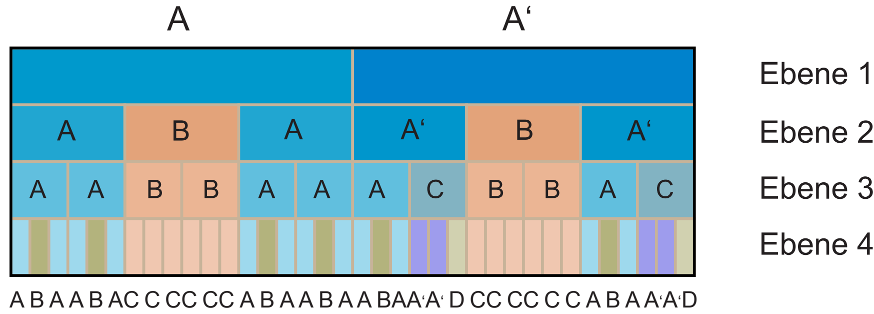


Abbildung 9.1.: Darstellung der Hierarchie (beschriftet)

In Abbildung 9.1 wird eine schematische Anzeige einer möglichen Hierarchiestruktur gezeigt. In der darauf folgenden Abbildung wird die Möglichkeit des interaktiven *Brushings* demonstriert. Eine Selektion innerhalb der analysierten Musikspur wird abhängig von ihrer Selektionslänge in den Kontext einer bestimmten Hierarchiestufe gestellt und dort dementsprechend hervorgehoben (siehe auch Abb. 9.2 „Hierarchie-Interaktion: *Brushing*“).

9.3. Geometrische Analyse

Die *geometrische Analyse* der Re-Compose Logik erstellt Hierarchiestrukturen für den Verbund von mehreren Musikspuren eines Musikstücks. Wie bei der *hierarchischen Analyse* werden die Musikspuren auf definierte, wiederkehrende Muster untersucht.

9.3.1. Geometrische Darstellung

Eine Darstellung der *geometrischen Analyse* visualisiert Abhängigkeiten und Beeinflussungen von mehreren oder allen Spuren eines Musikstücks zueinander. Ähnlich der *hierarchischen* Darstellung liegt die Herausforderung der *geometrischen* Darstellung bei der Integration von Musikspuren beliebiger Anzahl. Auch hier wird in der Planung ein *normierter* Ansatz gewählt, bei dem die Größe der Einzelspuranzeige abhängig von der Gesamtanzahl der Musikspuren ist. Es ist klar, dass bei dieser Festlegung Möglichkeiten zur Adaption der Anzeige zur Verfügung gestellt werden müssen. Beispielsweise kann durch individuelles Aktivieren oder Deaktivieren von Einzelspuren der vorhandene Anzeigebereich besser an jeweilige Situationen angepasst werden. In einer *zweidimensionalen* Darstellung der Ergebnisse der geometrischen Analyse ist im Gegensatz zur hierarchischen Darstellung nur eine einzige Hierarchiestufe gleichzeitig darstellbar. Für eine Auswahl der aktuell dargestellten Hierarchiestufe ließe sich die Anzeige an das *Brushing* der hierarchischen Struktur koppeln: Somit

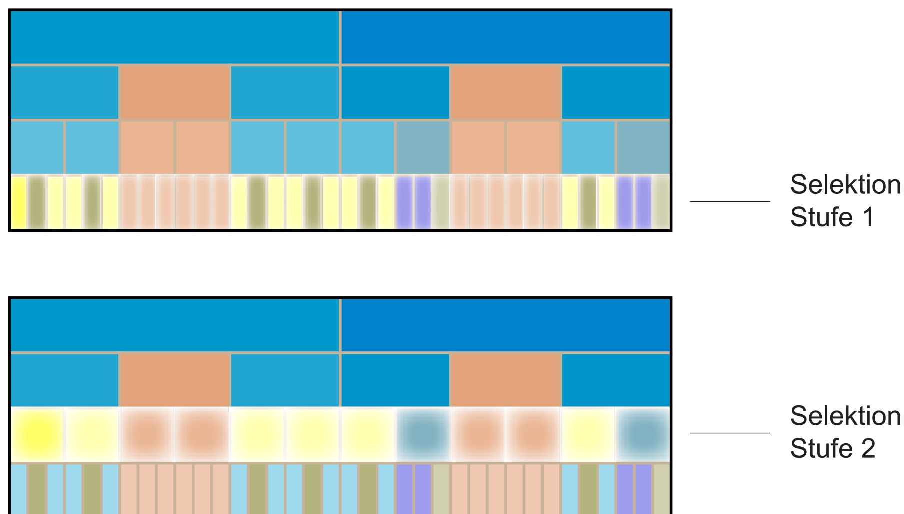


Abbildung 9.2.: Hierarchie-Interaktion: *Brushing*

kann die aktuell gewählte und hervorgehobene Hierarchiestufe der hierarchischen Darstellung automatisch die Hierarchiestufe der geometrischen Anzeige vorgeben (siehe auch Abb. 9.3 „Geometriebeispiel mit *Rhythmus*, *Melodie*, *Begleitung* und *Bass*“ und Abb. 9.4 „Gekoppelte geometrische und hierarchische Darstellungen“).

9.4. Zusammenfassung

Bei der Planung von unterschiedlichen Darstellungen der Ergebnisse von *hierarchischer Analyse* und *geometrischer Analyse* der Re-Compose Logikkomponente wurde ersichtlich, dass die erkannten Strukturen in zweidimensionalen Visualisierungen angezeigt werden können, wenn auch mit gewissen Einschränkungen. Die *geometrische* Darstellung beleuchtet eigentlich Strukturen im dreidimensionalen Raum. Mit Bindung der dritten Achse an einen Wert, der mittels *Brushing* in der hierarchischen Darstellung gesetzt werden kann, lässt sich diese Einschränkung lösen.

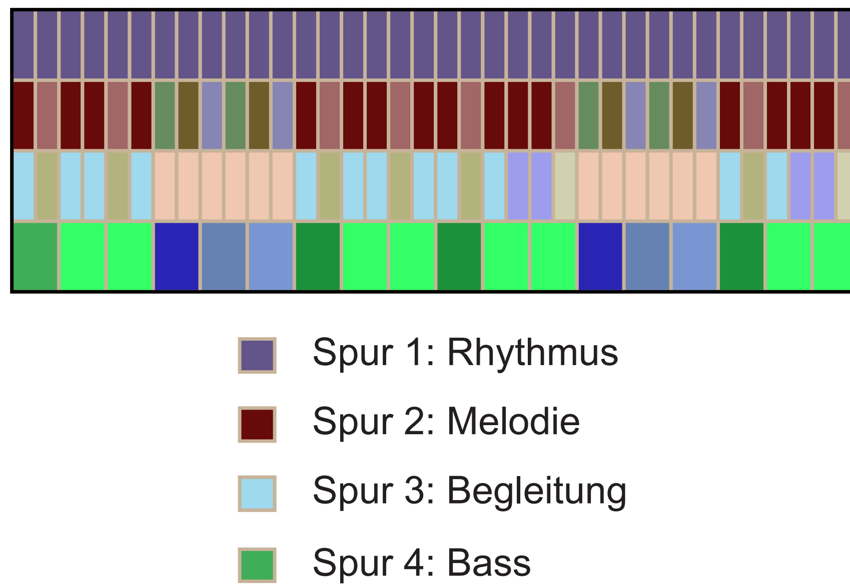


Abbildung 9.3.: Geometriebeispiel mit *Rhythmus*, *Melodie*, *Begleitung* und *Bass*

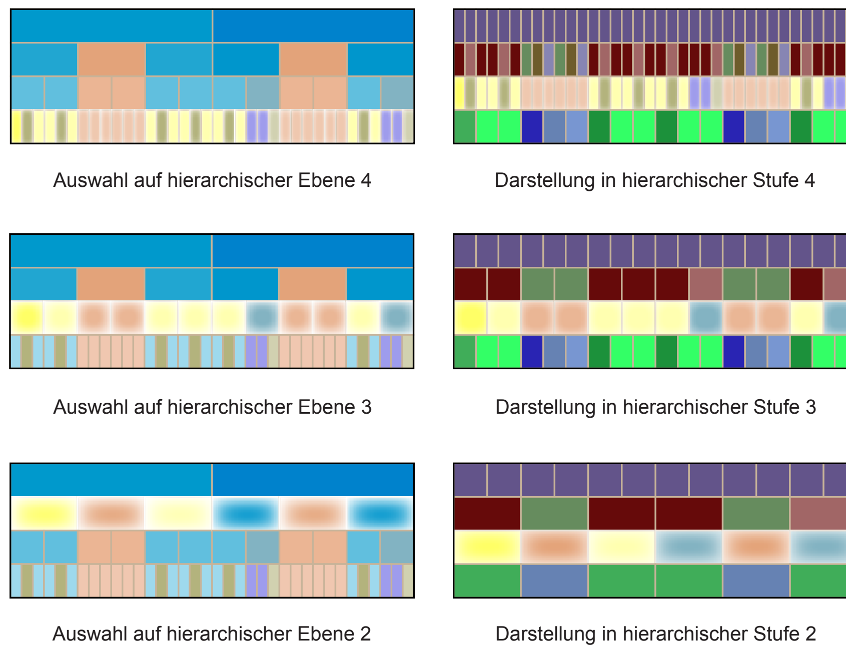


Abbildung 9.4.: Gekoppelte geometrische und hierarchische Darstellungen

Kapitel 10.

Weiterführende Ideen

10.1. Das „Multitrack-Problem“

Der Re-Compose Prototyp erfüllt seine Aufgabe im Aufzeigen der notwendigen Schritte zum Erfassen und Bearbeiten einer *einzelnen* Musikspur eines Musikstücks innerhalb der Java Re-Compose Logik. Am Beispiel der *dynamischen geometrischen Analyse* wird ersichtlich, dass in Zukunft ganze Systeme von Musikspuren kontextgetreu erfasst und an die Re-Compose Logik weitergegeben werden müssen - eine Anforderung, die mehrere Herausforderungen mit sich bringt.

VST-Plug-ins sind gemäß ihrer Konzeption atomare Automaten. Unbeeinflusst von ihrer Umgebung, also eventuell existierenden benachbarten Plug-ins, empfangen sie einen Musikdatenstrom ihres VST-Hosts, den sie nach ihrer Bearbeitung wieder an diesen zurück geben. Zur Erfassung eines ganzen Systems an Musikspuren müssen die Plug-ins dieser Spuren auf eine bestimmte Art und Weise miteinander kollaborieren können.

Als zukünftige Arbeit wird zur Lösung ein *Master/Slave*-Konzept vorgeschlagen, welches gut in die aktuelle Prototypstruktur, die schon in mehrere Threads aufgeteilt wurde, eingepasst werden kann.

10.1.1. Master/Slave-Plug-ins

Die Grundidee sieht vor, dass Re-Compose Plug-in-Instanzen zu Beginn ein *Master*-Plug-in aufsuchen und sich an dieses als *Slave* binden. Kann eine Instanz keinen *Master* finden, ernennt sie sich selbst zu einem *Master*-Plug-in. Nur ein *Master*-Plug-in kann mit der Re-Compose Logik kommunizieren. Dafür startet und betreibt der *Master* den (einzigen) Kindprozess mit *virtueller Java-Maschine* und darauf laufender Re-Compose Logikkomponente. *Slave*-Plug-ins übermitteln und erhalten ihre Daten ausschließlich vom *Master*-Plug-in.

Die weiteren Herausforderungen für dieses Konzept sind offensichtliche Probleme bezüglich

Gleichzeitigkeit und Auffindbarkeit (*concurrency* und *identifiability*):

- Zwei Plug-ins könnten zur gleichen Zeit erfolglos nach einem *Master* suchen und sich daraufhin zu *Master* Plug-ins ernennen.

- Pro Musikstück wird ein *Master*-Plug-in benötigt. Ein Verfahren zum Veröffentlichen und Koordinieren aller aktiven *Master*-Plug-ins muss dafür erarbeitet werden.

10.2. Interaktivität

Ein wichtiger Punkt bei der grafischen Darstellung der *hierarchischen* und *geometrischen* Strukturen ist die Interaktion mit diesen angezeigten Ergebnissen. Eine weitere Idee ist eine Beeinflussung des Musikstücks per *drag & drop* ganzer Strukturblöcke. Dabei könnte ein Block innerhalb der *hierarchischen Darstellung* vom Benutzer verschoben werden, während die Re-Compose Logik dazu gehörende oder abhängige Strukturen automatisch adaptiert.

10.3. Farbkodierung

Die in Kapitel 9 präsentierten *hierarchischen* und *geometrischen* Strukturdarstellungen verwenden unterschiedliche Farben für eine intuitive Präsentation der aktuellen Strukturen. Die Farben werden zufällig gewählt. Hierbei ist eine Überlegung, die Farbe beziehungsweise ihre Opazität an weitere Parameter der Re-Compose Logikanalysen zu koppeln. Beispielsweise könnte die Häufigkeit von erkannten Strukturen bei der hierarchischen Analyse als weiterer Darstellungsparameter eingebunden werden.

10.4. Zusammenfassung

Im Re-Compose Projekt gilt es nach der Fertigstellung des Prototyps, auch in Zukunft anspruchsvolle Herausforderungen zu lösen. Nach der erfolgreichen Verbindung von Audio-Sequenzerprogrammen mit dem Re-Compose Logikmodul über den Prototyp als *ein* VST-Plug-in kann *eine* Musikspur eines Stücks bearbeitet werden. In Zukunft soll auch die Bearbeitung von mehreren Spuren durch die Zusammenarbeit von mehreren Plug-ins möglich werden. Dies könnte mit einer *Master*/*Slave* Struktur und Einteilung der Plug-ins und dafür geeigneten Synchronisationsprotokollen umgesetzt werden. Die Visualisierungen der Analyseergebnisse sollen in Zukunft mit interaktiven Elementen und werteabhängigen Farbkodierungen noch bessere Hilfestellung beim Komponieren und Arrangieren von Musikstücken geben können.

Anhang A.

Abbildungsverzeichnis

3.1. Ein Viertelnotenschlag 14
3.2. Vier Viertelnoten in einem $\frac{4}{4}$ Takt 15
3.3. Ganze-, Halbe- und Viertelnoten 15

4.1. UML-Sequenzdiagramm: VST-Initialisierung 19
4.2. UML-Sequenzdiagramm: VST-Kommunikation 20
4.3. Struktur der VST-SDK-Basisdateien 23

6.1. Übersicht *Java*, *JNI* und *native* Anwendungen 31

7.1. Übersicht Prototypdesign JNI 40
7.2. Übersicht Prototypdesign VST 40
7.3. Übersicht Prototypmodule 41
7.4. UML-Klassendiagramm des Prototyps 43

8.1. UML-Sequenzdiagramm des Prototyps 54

9.1. Darstellung der Hierarchie (beschriftet) 56
9.2. Hierarchie-Interaktion: *Brushing* 57
9.3. Geometriebeispiel mit *Rhythmus*, *Melodie*, *Begleitung* und *Bass* 58
9.4. Gekoppelte geometrische und hierarchische Darstellungen 58

Anhang B.

Tabellenverzeichnis

4.1. Aufzählung: Plug-in-Verarbeitungscodes (VST SDK 2.4) 21
4.2. Aufzählung: Host-Verarbeitungscodes (VST SDK 2.4) 21
4.3. SDK-Quelldateien eines VST-Basisprojektes 23

5.1. MIDI-Protokoll: Bytes 1..4 28

6.1. JNI-Typdefinitionen für Basistypen [Lia99] 32
6.2. JNI-Felddeskriptoren 34

Anhang C.

Literaturverzeichnis

Die Literaturangaben sind alphabetisch nach den Namen der Autoren sortiert. Bei mehreren Autoren wird nach dem ersten Autor sortiert.

[GWC60] Grosvenor W. Cooper, Leonard B. Meyer: *The Rhythmic Structure of Music.* University of Chicago Press, Chicago, Illinois, 1960.

[Lia99] Liang, Sheng: *The Java Native Interface - Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[MID01] MIDI Manufacturer's Association: *Complete MIDI 1.0 Detailed Specification.* 2001.

[Roa96] Roads, Curtis: *The Computer Music Tutorial.* The MIT Press, 1996.

[Ste06] Steinberg Media Technologies GmbH: *Virtual Studio Technology Software Development Kit 2.4 Documentation.* 2006.

Glossar

Equalizer

Ein Audio *Equalizer* filtert Audiosignale, wobei er je nach Einstellung verschiedene Frequenzbereiche verstärkt oder abschwächt. Der häufig verwendete *Grafik-Equalizer* hat als Benutzerschnittstelle meist eine interaktive Darstellung der Parametrisierung von zu verstärkenden oder abzusenkenden Audiofrequenzen. (18)

Java

Java ist eine von *Sun Microsystems, Inc.* entwickelte objektorientierte Programmiersprache, deren Compiler einen abstrakten Bytecode erzeugen, der plattformunabhängig in Java-Laufzeitumgebungen mit Hilfe sogenannter *virtuellen Java-Maschinen* ausführbar ist (siehe auch Gloss. JavaVM). (5, 6, 12, 13, 30–32, 35, 36, 47, 55, 59)

JavaVM

JavaVM (oder auch JVM abgekürzt) ist die Bezeichnung der *virtuellen Java-Maschine* einem Kernmodul einer Java Laufzeitumgebung. Eine *virtuellen Java-Maschine* interpretiert den plattformunabhängigen Java-Bytecode und führt die entsprechenden Anweisungen auf der eingesetzten Plattform (d.h. dem eigentlichen Anwendungssystem) aus. (13, 31, 38, 52, 64)

Mutex

Mutex ist eine Abkürzung für *mutual exclusion* und ist ein Konstrukt in der Informatik zur Synchronisation von kritischen Bereichen (*critical sections*). Nur **einem** Teilnehmer wird erlaubt, einen kritischen Bereich einzunehmen. Zukünftige Teilnehmer müssen warten, bis sie an der Reihe sind, den kritischen Bereich zu betreten. *Mutex* kann auch als Spezialfall einer *Semaphore* mit maximaler Teilnehmerzahl gleich 1 gesehen werden (vgl. Semaphore). (48, 65)

Quantizer

Quantizer oder Größenwandler werden in der Musik zur Anpassung von Noten an eine vorgegebene minimale Zeiteinheit verwendet. Für MIDI gibt es *Quantizer* in Hardware-

und Softwareausführung. Meistens werden *Quantizer* verwendet, um zeitlich unpräzise Musikspuren, z.B. nach einer Aufnahme per *MIDI Recording* zu präzisieren. (44)

Semaphore

Eine *Semaphore* bezeichnet in der Informatik ein Konstrukt zur Synchronisation von kritischen Bereichen (*critical sections*) mit beschränkten Teilnehmerplätzen. Ist die maximale Anzahl an Teilnehmern eines kritischen Bereichs erreicht, müssen zukünftige Teilnehmer warten, bis entsprechende Plätze wieder frei sind (vgl. Mutex). (47, 64)

Sequenzer

Ein Software-Musik*sequenzer* ist ein Programm zur Aufnahme und gleichzeitigen Wiedergabe von mehreren Musik*spuren* (engl. *tracks*). Früher bedienten Sequenzer hauptsächlich MIDI-Spuren. Heutzutage verarbeiten Sequenzer meist MIDI-Spuren gemeinsam mit Audiospuren (d.h. digitalisierten Audiosignalen). (5, 6, 13, 16, 27, 42, 60, 66)

Synthesizer

Ein Audio*synthesizer* erzeugt elektronische Audiosignale und versucht, die Charakteristika von realen Instrumenten so gut wie möglich nachzubilden. Ein *Synthesizer* wird oft über eine MIDI-Schnittstelle angesprochen und erzeugt zu den empfangenen MIDI-Noten Audiosignale in den entsprechenden Tonhöhen. (18, 27, 66)

Akronyme

API

Application Programming Interface, Schnittstelle zur Anwendungsprogrammierung. Ein API wird hauptsächlich auf Quelltextebene definiert und beschrieben (vgl. SDK, Bsp. *OpenGL API* und *DirectX SDK*). (31, 38, 66)

bpm

Die Abkürzung von *beats per minute* (Schlägen pro Minute) wird allgemein als Maßeinheit für die Geschwindigkeit eines Musikstücks angegeben. Bsp.: *120 bpm* für 120 Schläge pro Minute. (15)

JNI

Das *Java Native Interface (JNI)* ist eine von Sun Microsystems, Inc. entwickelte Schnittstelle zur Anbindung und Kopplung von Java-Applikationen (innerhalb einer virtuellen Java-Maschine) an *native* Prozesse des Anwendersystems. (13, 30–36, 38, 39, 41, 51–53)

MIDI

Das *Musical Instrument Digital Interface* ist eine digitale Schnittstelle für die Übertragung von Musikdaten. *MIDI* beschränkt sich dabei hauptsächlich auf die Übertragung von Steuersignalen wie Tonhöhe, Tonlänge usw. Das eigentliche Audiosignal wird nicht übermittelt. Ein MIDI-Sequenzer kann empfangene Steuersignale zeitlich geordnet aufnehmen und anschließend wiedergeben. Ein MIDI-Synthesizer kann die empfangenen Steuersignale interpretieren und *synthetische* Audiosignale erzeugen. (13, 16, 18, 27, 28, 38, 50, 51, 64, 65)

ppq

Die Anzahl von Impulsen pro Viertelnote (*pulses per quarter*) wird oft bei MIDI-Sequenzern als Qualitätsmerkmal angegeben und beschreibt dabei den Maximalwert für die Abtast- und Wiedergabeauflösung zeitlicher Ereignisse. (16, 27, 51)

SDK

Software Development Kits (SDK) dienen in der Regel zur Umsetzung von Softwareaufgabestellungen innerhalb vorgegebener thematischen Bereiche oder auch zur Entwicklung von Ergänzungen und Erweiterungen zu bestehenden Anwendungen. Sie beinhalten dazu Sammlungen von Quellcode-Modulen, (Laufzeit-) Bibliotheken, entsprechenden Hilfe- & Beispieldateien und manchmal auch ergänzenden Tools und Editor Applikationen. (vgl. API, Bsp. *DirectX SDK* und *OpenGL API*). (13, 20, 22, 25, 26, 38, 42, 50, 66)

SOAP

Das *Simple Object Access Protocol* ist eine Schnittstellendefinition und Protokollbeschreibung für den Austausch von Objekten in objektorientierten Umgebungen. Das beinhaltet sowohl den Zugriff auf Eigenschaften der Objekte (*field members* oder *properties*) wie auch die Einbindung (*invocation*) von deren Methoden. (36)

VST

Virtual Studio Technology (VST) - eine von Steinberg Media Technologies entwickelte Schnittstelle zur Erweiterung von Audio-Softwareanwendungen (*hosts*). Dazu zählen Sound- (*instrument plug-ins*) und Effektprozessoren (*filter plug-ins*). (13, 18–20, 22, 24–26, 38, 39, 42, 50–53, 55, 59, 60)

XML

Die *eXtensible Markup Language* ist als Weiterentwicklung von *HTML* die Definition einer hierarchischen, objektorientierten und streng formalisierten Sprache für den Austausch von Informationen in einer idealerweise auch für Menschen lesbaren Form. (36)