



Folkwang
Universität der Künste

BACHELORPROJEKT INTEGRATIVE KOMPOSITION

mutwo: eine Ereignis zentrierte Umgebung
zur Formalisierung zeitbasierter Künste

AUTOR: Levin Eric Zimmermann
EMAIL: levin-eric.zimmermann@folkwang-uni.de
MATRIKELNUMMER: 2332991
BETREUUNG: Prof. Dr. Michael Edwards

19. September 2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum

Hanoi, 19.09.2022

Unterschrift

A handwritten signature in black ink, consisting of a stylized 'G' followed by a flourish.

Inhaltsverzeichnis

I	Einleitung	I
I.1	Komposition und Werkzeuge (I)	I
I.2	Komposition und Werkzeuge (II)	2
2	<i>mutwo</i>	4
2.1	Motivation und Absicht	4
2.2	Softwarearchitektur	6
2.2.1	Komponenten und Beziehungen	6
2.2.2	Ereignisse	7
2.2.3	Parameter	9
2.2.4	Übersetzer	9
2.2.5	Generatoren	10
2.3	Entwicklungsstrategien	10
2.3.1	Programmiersprache	10
2.3.2	Strukturierung des Quellcodes	11
2.3.3	Abstrakte Basisklassen	14
2.3.4	<i>Konvention vor Konfiguration</i>	15
2.3.5	Globale Voreinstellungen	16
2.3.6	Dokumentation öffentlicher Schnittstellen	17
2.3.7	Unspezifische Ereignisse; dynamische Akzessoren	17
2.3.8	Annotation von Typen im Quellcode	20
2.3.9	Konsistente Namenskonventionen	21
2.4	Limitierungen und Grenzen	21
2.5	Fallbeispiele	23
2.5.1	<i>thanatos trees for Tim Pauli</i>	23
2.5.2	<i>ohne Titel (2)</i> und <i>ohne Titel (3)</i>	28
3	Schlusswort	33
3.1	Zusammenfassung	33
3.2	Aussichten	33

Abbildungsverzeichnis

1	Beziehungen zwischen verschiedenen Komponenten <i>mutwos</i>	7
2	Exemplarische Verschachtlung von Ereignissen	8
3	Schematische Darstellung eines Parameter und seine unterschiedliche Teil- klassen. Die abgeleiteten Klassen ermöglichen unterschiedliche Perspektiven derselben Entität. Sie werden mit unterschiedlichen Argumenten initialisiert.	15
4	<i>thanatos trees for Tim Pauli</i> im LTK ₄ in Köln.	24
5	Skizze zum Aufbau der Installation.	25
6	Schematischer Aufbau der Programmstruktur von <i>thanatos trees for Tim Pauli</i> . Gelb markierte Elemente repräsentieren Ereignisse, grün markierte Elemente sind Übersetzer.	27
7	Exemplarisches <i>time-bracket</i> der Violinstimme von <i>ohne Titel (3)</i>	28
8	Schematischer Aufbau der Programmstruktur von <i>ohne Titel (2)</i> und <i>ohne Titel (3)</i> . Gelb markierte Elemente repräsentieren Ereignisse, grün markierte Elemente sind Übersetzer.	30

Tabellenverzeichnis

1	Kernereignisse in <i>mutwo</i>	8
2	Liste exemplarischer Ein-Wert-Parameter	9
3	Exemplarische Übersetzer	9
4	Exemplarische Generatoren	10
5	Moduldefinitionen	11
6	Submoduldefinitionen	12
7	Exemplarische Module	13
8	Präzision der Liste exemplarischer Ein-Wert-Parameter	14

I Einleitung

I.1 Komposition und Werkzeuge (I)

Komposition ist von einer unbestimmten Menge Werkzeuge bedingt. Die Menge umfasst Kulturtechnologien wie Notation oder Stimmungen, Handwerk wie Instrumentenbau, Architekturen wie Konzerthäuser und *pendapa*, soziale Strukturen des Musizierens oder mathematische und logische Denkmodelle.

Letztgenannte Teilmenge umfasst Stimmführungsregeln oder 12-Ton Reihen. Sie können als eine geordnete Sequenz diskreter Handlungsschritte beschrieben werden, die Eingangswerte in Ausgangswerte transformieren (i.e. Algorithmen) [Cormen u. a. 1990, S. 3]. Algorithmische Komposition bezeichnet Komposition, die diese Werkzeuge verwendet [Nierhaus 2009, S. 1].

Mit dem Aufkommen der Computer wurden traditionelle Werkzeuge digitalisiert. Lejaren Hiller und Leonard Isaacson sind als erste Personen bekannt, die Algorithmen in einem Computersystem zum Zweck der Komposition implementierten [Nierhaus 2009, S. 63]. Auf sie folgten weitere. Mit fortschreitender Entwicklung wuchs die Notwendigkeit generische Programmbestandteile zu entwickeln, die in unterschiedlichsten Arbeiten wieder verwendet werden können (Bibliotheken oder Rahmen) [Gerzso 1992, S. 78].

Im Herbst 2020 kann ich eine Vielzahl von Softwarebibliotheken für algorithmische Komposition finden. Unzufrieden mit bestehenden Lösungen beginnen Tim Pauli und ich eine autonome Lösung zu entwickeln. Fast zwei Jahre später, im Sommer 2022, umfasst das resultierende *mutwo* Ökosystem über 22000 Zeilen Quellcode und 430 Tests. Seit initialer Entwicklung sind mithilfe des Projektes sechs Kompositionen entstanden.

In vorliegender Arbeit möchte ich *mutwo* dokumentieren. Quelloffen und mit der GPL-3.0 Lizenz veröffentlicht ist *mutwo* für Dritte zugänglich. Wirkliche Zugänglichkeit ist aber nur mit ausreichender Dokumentation gewährleistet. Meine kompositorische Arbeit ist durch die Bemühungen unzähliger Menschen möglich, die Freie Software veröffentlichen¹. Mit

¹Hier ist *Frei* im Sinne der Definition der *Free Software Foundation* (FSF) verstanden. Die FSF bezeichnet eine Software als Freie Software, falls sie von Benutzer:innen geteilt, gelesen und verändert werden darf [*What is Free Software?* o. D.].

der Dokumentation *mutwos* möchte ich einen Teil in die Gemeinschaften Freier Software zurückgeben.

1.2 Komposition und Werkzeuge (II)

“It would seem axiomatic that any music [...] reveals the philosophic attitude of its creator. It also seems self-evident that if his attitude is vigorous and individualistic, his practical requirements are not necessarily satisfied by the traditions he was born to; they may even require direct antitheses.” [Partch 1949, S. 3]

Was Partch mit “praktische Notwendigkeiten” bezeichnet, begreife ich als kompositorische Werkzeuge. Das Zitat impliziert, dass diese mitnichten neutral sind, sondern sich in einem engen Austausch mit den inneren Vorstellungen der Werkschaffenden befinden. Mit der autonomen Entwicklung akustischer Instrumente war es Partch möglich implizite Vorbedingungen der Komposition via expliziter Entscheidungen neu zu verhandeln. Die Begründung der Anstrengung für die kompositorische Praxis Programme zu entwickeln, deckt sich für mich mit seiner Begründung für die Konstruktion eigener Instrumente. Vernachlässigte Eigenschaften in bestehender Software verhindern eine Kongruenz von innerer Vorstellung und praktischen Möglichkeiten. Besonders die Voreingenommenheit populärer Musiksoftware ist bekannt. Andrew Gerzso benennt die von konventioneller Musik geprägten Annahmen eines Sequenzers [Gerzso 1992, S. 78]. Kyam Allami bemängelt die oberflächliche, exotisierende Repräsentationen außereuropäischer Tonsystemen in kommerzieller Musiksoftware [Allami 2019, S. 59f]. Aber auch esoterischere Technologien wie Softwarebibliotheken für algorithmische Komposition unterliegen inhärenten Limitierungen.

Limitierende Paradigmen mir bekannter Bibliotheken waren und sind ausschlaggebend für die Entwicklung *mutwos*. Meine initiale Begegnung mit Einschränkungen betraf Tonhöhenstrukturen. In meiner kompositorischen Arbeit begreife ich Intervalle als ganzzahlige Frequenzverhältnisse (reine Stimmung). Meine bevorzugte Tonhöhenrepräsentation von Verhältnissen befindet sich jenseits europäischer Tonhöhennamen oder der MIDI Spezifikation.

In der Bibliothek *SCAMP* werden Tonhöhen als Gleitkommazahlen repräsentiert, die Tonhöhennummern der MIDI Spezifikation bezeichnen [Evanstein 2021].

Die Software *Euterpea* deklariert Tonhöhen als zweielementige Tupel. Das erste Element ist ein Tonklassenname europäischer Tradition (z. B. `cs` oder `fb`) und das zweite eine natürliche Zahl zur Indikation der Oktave [Quick 2019].

Die Notenklasse der Bibliothek *jMusic* definiert drei unterschiedliche Konstruktoren, um Tonhöhen festzulegen: wird dem Tonhöhenargument natürliche Zahlen zugewiesen, werden diese als Tonhöhennummern der MIDI Spezifikation interpretiert. Gleitkommazahlen werden als Frequenz gelesen, Zeichenketten als europäische Tonhöhenamen. Die innere Repräsentation der Klasse erlaubt nur die Darstellung in Frequenz oder MIDI Nummer, sodass Tonhöhenamen vom Konstruktor umgewandelt werden [Brown u. a. 2017].

In *slippery-chicken* sind Tonhöhen über eine eigene Klasse implementiert. Die Funktion `make-pitch` erzeugt eine neue `pitch`-Instanz. Ähnlich wie bei *jMusic* gibt es unterschiedliche mögliche Datentypen des notwendigen Arguments `pitch` der `make-pitch` Funktion. Falls `pitch` ein Symbol oder eine Zeichenkette ist, wird das Argument als Tonname europäischer Tradition gelesen. Ist `pitch` eine Zahl wird diese als Frequenz der Tonhöhe interpretiert. Der Klasse `pitch` sind Attribute zugewiesen wie z. B. `midi-note`, `white-note`, `accidental` oder `frequency` [Edwards u. a. 2021].

Die exemplarische Beschreibungen verdeutlichen die Schwierigkeit der Repräsentation von Tonhöhen als Schwingungsverhältnisse in bestehenden Lösungen. Es ist denkbar, beschriebene Software zu erweitern.

In *SCAMP* oder *Euterpea* könnte man eine neue Klasse bzw. Deklaration und Umwandlungsfunktion definieren, die Tonhöhen als Verhältnisse beschreiben.

In *jMusic* oder *slippery-chicken* könnte man von den Klassen `Note` bzw. `pitch` erben. In der abgeleiteten Klasse könnten dann ein weiteres Attribut (z. B. `ratio`) hinzugefügt werden bzw. der entsprechende Konstruktor hinzugefügt werden.

Die Erweiterungen *SCAMPs* oder *Euterpeas* befände sich aber außerhalb der Bibliothek. Die Operationen mit dem erweiterten Potenzial würden jenseits einer Interaktion mit Funktionen und Instanzen der Bibliothek statt finden.

Die Erweiterung von *jMusic* oder *slippery-chicken* befände sich innerhalb der Bibliothek. Die abgeleiteten Klassen oder neue Konstruktoren enthielten vererbte Attribute, die für sie

potenziell unwesentlich wären (z. B. Informationen zu westlicher Notation).

Das Projekt *mutwo* ist der Versuch der Realisierung der Utopie allen Benutzer:innen unvoreingenommen ihre jeweilige Repräsentation zu ermöglichen. Das Projekt ist Versuch und Utopie, weil die Komplexität der Spezifikation und Implementierung Einschränkungen bedingt. Eine Präzision dieser Limitierungen findet sich im späteren Teil vorliegender Arbeit. Die Einschränkungen betreffen daneben Grenzen, denen alle Programmierumgebungen algorithmischer Komposition unterworfen sind. Sie tendieren z. B. zum Ausschluss musikalischer Praxen, welche Improvisation der Komposition vorziehen.

Ausgangspunkt der Vorstellung *mutwos* ist eine umfassende Darstellung der Motivation und Absicht dessen Entwicklung. Daran knüpft eine abstrakte Spezifikation ihrer Architektur an. Anschließend werden konkrete Programmierstrategien zur Umsetzung der Absicht vorgestellt. Die Strategien dokumentieren innere Zusammenhänge der Software. Umfassendes Verständnis für sie sind für avancierte Anwendungen und Weiterentwicklung oder Instandhaltung *mutwos* notwendig. Vor abschließender Zusammenfassung werden zwei Fallbeispiele präsentiert, die zeigen wie *mutwo* in komplexen Kompositionsprojekten eingesetzt werden kann.

Manchmal sind Konzepte mit Programmcode in der Programmiersprache Python verdeutlicht. Syntax der Sprache wird nicht erklärt, von einem grundsätzlichen Verständnis wird ausgegangen ². Verdeutlichende Graphiken sind informell und folgen nicht (oder nur lose) einer Formalisierung (wie z. B. UML).

2 *mutwo*

2.1 Motivation und Absicht

Motivation der initialen Entwicklung *mutwos* war die Erkenntnis einer Diskrepanz zwischen bestehenden Softwareprojekten und meiner eigenen Arbeitsweise. Diese Diskrepanz bedingte die Vorstellung eines Softwaredesigns, was möglichst generisch und flexibel ist. Je unspezifischer die Software wäre, desto einfacher sollte es für Dritte sein, diese für ihre Zwecke anzupassen ³.

²Bei Unklarheiten wird auf Pythons offizielle Sprachreferenz verwiesen [*The Python Language Reference* o. D.].

³Dritte umfassen auch mich selbst in einer unbestimmten Zukunft, in der ich Notwendigkeiten entdecke, die mir in der Gegenwart noch unbewusst sind.

Zugleich wurde deutlich, dass Gegebenheiten definiert und konkretisiert werden müssen, um eine sinnvolle (d. h. praktikable) Bibliothek bereitzustellen. Die wichtigste Aufgabe einer Bibliothek ist es letztlich wiederverwendbare, praktische Bestandteile zwischen unterschiedlichen Projekten zu teilen. Ist eine Bibliothek zu unspezifisch, mag diese Voraussetzung unerfüllt bleiben.

*Mutwo*s Designabsicht kann unter dem Begriff des Agnostizismus oder der Neutralität zusammengefasst werden.

1. ***Mutwo* ist software- und protokollagnostisch.** *Mutwo* trennt innere Repräsentationen von Spezifikationen dritter Software oder Protokolle. Spezifikationen dieser entstehen am Rande der Bibliothek, wenn innere Repräsentationen in die entsprechenden Strukturen umgewandelt werden. Das ermöglicht eine flexible Adaption an unterschiedliche dritte Software oder Protokolle. Es verhindert auch eine Abhängigkeit von spezifischen dritten Technologien.
2. ***Mutwo* ist medienagnostisch.** Die grundsätzliche Struktur ist so unspezifisch, dass verschiedene zeitbasierte Künste darstellbar sind.
3. ***Mutwo* ist interfaceagnostisch.** *Mutwo* ist nur eine lose zusammenhängende, erweiterbare Sammlung von Objekten. Die Bibliothek macht keine Aussagen über eine bestimmte Arbeitsweise mit diesen Objekten. Die Bibliothek kann als Grundlage für eine bestimmte Benutzeroberfläche oder Benutzerschnittstelle verwendet werden, ist aber unabhängig von diesem.
4. ***Mutwo* ist ästhetikneutral.** Die Software vermeidet ästhetische Entscheidungen.
5. ***Mutwo* ist plattformübergreifend.** *Mutwo* ist mit Technologien entwickelt, die von unterschiedlichen Betriebssystemen (Linux/GNU, Mac OS, Windows) unterstützt werden.
6. ***Mutwo* ist traditionsagnostisch.** *Mutwo*s Kern inkludiert und exkludiert keine Repräsentationen bestimmter (Musik-)Traditionen.

7. **Mutwo ist autonom.** Die Bibliothek wird unabhängig von Institutionen entwickelt. Sie kann von allen Menschen gelesen, geteilt und verändert werden. Sie ist befreit von monetären Absichten.

Der Komplexität unspezifischer Abstraktionen wird die Absicht entgegengesetzt, möglichst einfach, produktiv und (intuitiv) verständlich zu sein. Beide Zielen unterliegen der gemeinsamen Intention offen, zurückhaltend und freundlich gegenüber Benutzer:innen zu sein. In den folgenden Präzisierungen der Softwarearchitektur und der Entwicklungsstrategien wird verdeutlicht, wie diese Gegensätze ineinander integriert sind.

2.2 Softwarearchitektur

2.2.1 Komponenten und Beziehungen

Objekte in *mutwo* sind in strikte Kategorien geteilt.

Ereignisse beschreiben eine Bewegung.

Parameter sind Ereignissen zugeordnet und definieren die Qualität der Bewegung.

Übersetzer transformieren Inhalt oder Form einer Entität (z. B. Ereignis, Parameter).

Generatoren erzeugen Daten für eine künstlerische Arbeit.

Abbildung 1 skizziert die Beziehungen der unterschiedlichen Kategorien zueinander. Sie zeigt, wie sich *mutwo* mithilfe von Übersetzer mit der äußeren Welt verbindet. Übersetzer können anhand ihrer Richtung unterteilt werden; entweder werden externe Daten in innere Repräsentation übersetzt (*backend*) oder innere Repräsentation in externe Daten (*frontend*) oder innere Repräsentationen in andere innere Repräsentationen (*symmetrical*). Die Pfeile der Skizze markieren meist Eingangs- und Ausgangswerte eines Übersetzer. Gleiche Farben gehören zur gleichen Übersetzungsrichtung. Eingangswerte sind mit *in* notiert und Ausgangswerte mit *out*. Hat ein Element der Abbildung mehrere Ein- oder Ausgangswerte, werden zusammengehörige Ein- und Ausgangswerte mit Indizes markiert.

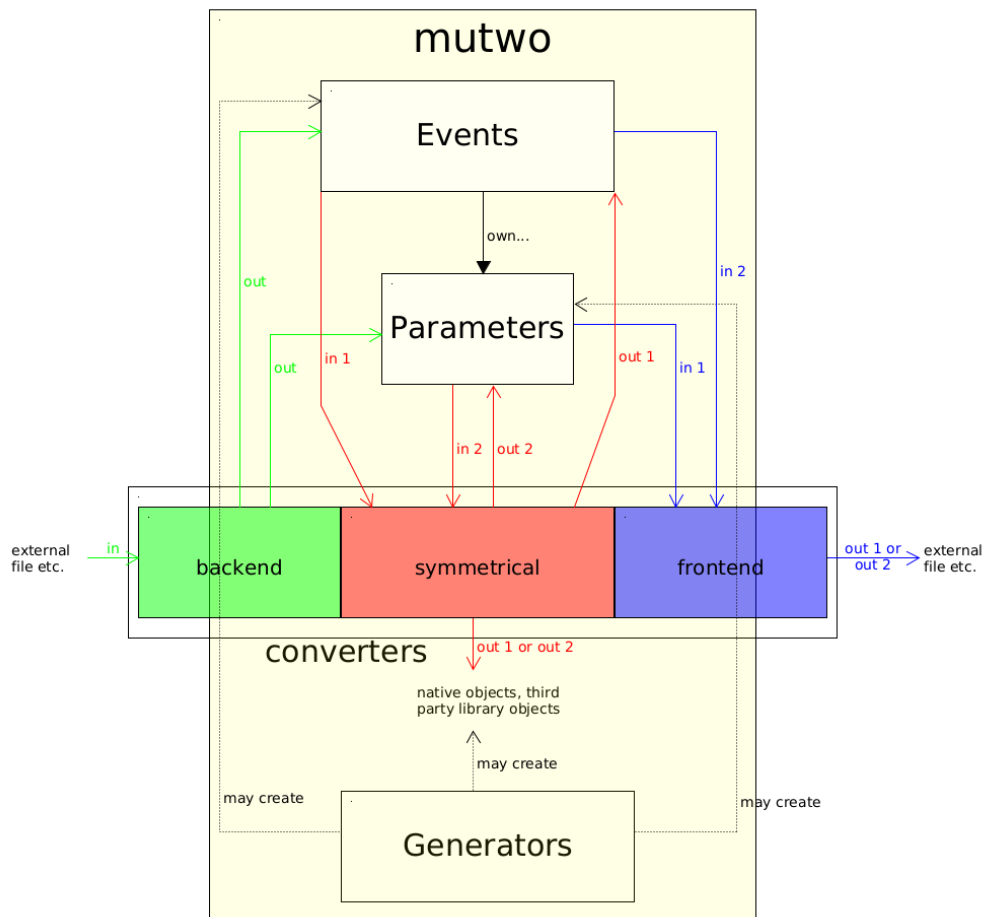


Abbildung 1: Beziehungen zwischen verschiedenen Komponenten *mutwo*s.

2.2.2 Ereignisse

Ein Ereignis beschreibt eine Bewegung in einem Koordinatensystem mit $n \in \mathbb{N} : n > 0$ Dimensionen. Die Bewegung wird durch n Vektoren dargestellt. Dimensionen können der Zeit, räumlichen Achsen (X, Y und Z) oder theoretischen Konstrukte (z. B. den Primzahlexponenten eines Tonnetzes) entsprechen.

Ereignissen sind zusätzliche Objekte zugeordnet. Die Objekte definieren Eigenschaften der Bewegung. Ein zugewiesenes Objekt könnte für einen Klang z. B. eine Tonhöhe sein. Für eine zweidimensionale räumliche Bewegung wäre eine Farbe denkbar, für eine dreidimensionale Bewegung eine bestimmte Gangart.

In *mutwo*s Terminologie werden Objekte, die als Attribute einem bestimmten Ereignis zugewiesen sind, als Parameter bezeichnet. Weil auch Vektoren als Attribute einem Ereignis zugeordnet sind, sind sie nur eine besondere Unterkategorie von Parametern. In *mutwo*s objektorientiertem Paradigma sind Ereignisse und Parameter Klassen und deren Instanzen.

Klassenname	ist Behälter	innere Verhältnisse	Beispiel
SimpleEvent	nein	-	ein Ton, ein Strich
SequentialEvent	ja	akkumulierend	eine Melodie, ein Quadrat
SimultaneousEvent	ja	parallel	Polyphonie, zwei Rechtecke

Tabelle 1: Kernereignisse in *mutwo*

Ereignisse können andere Ereignisse enthalten oder keine anderen Ereignisse enthalten. Falls Ereignisse andere Ereignisse enthalten, können die enthaltenen Ereignisse wiederum iterativ weitere Ereignisse enthalten (Verschachtlung). Falls ein Ereignis Ereignisse enthält, können diese entweder simultan (parallel) oder sequenziell (akkumulierend) angeordnet sein. Mit den drei Ereignisklassen SimpleEvent, SequentialEvent und SimultaneousEvent sind alle Möglichkeiten enthalten.

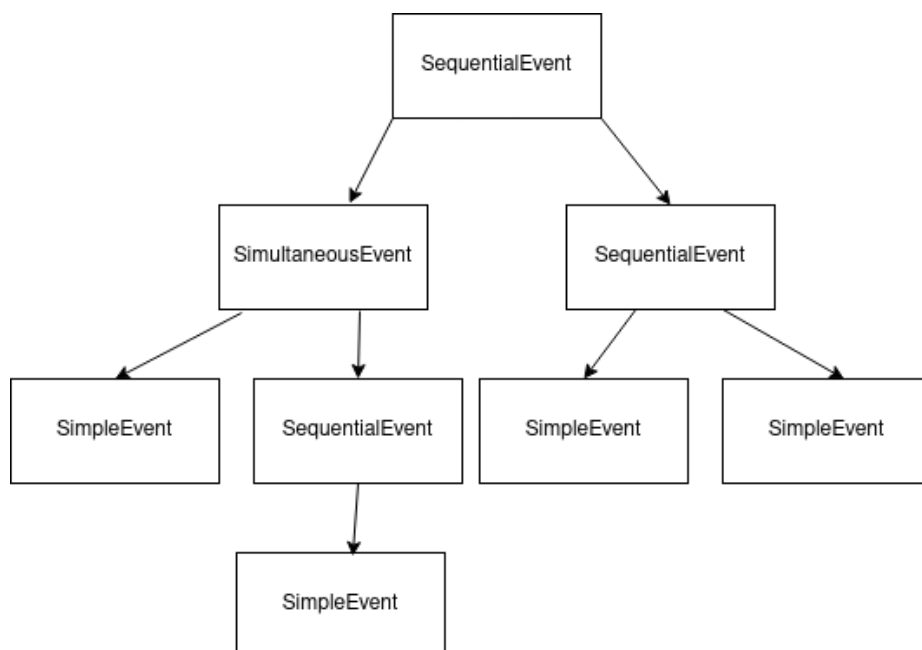


Abbildung 2: Exemplarische Verschachtlung von Ereignissen

2.2.3 Parameter

Parameter repräsentieren generische Kategorien, die Ereignissen zugeordnet werden. Generische Kategorien können beispielweise eine Farbe, eine Tonhöhe oder ein Luftdruck sein.

Mutwo versucht Parameter über eine möglichst kompakte, generische Identität zu beschreiben. Wenn möglich besteht diese kompakte Identität aus nur einem Wert (z. B. Zeichenkette oder Zahl). Wenn möglich, ist der Wert implizit einer physikalischen Einheit zugeordnet.

Parameter	Einheit
Tonhöhe	Hertz
Tonhöhenintervall	Cents
Dauer	beats
Text	X-SAMPA ⁴

Tabelle 2: Liste exemplarischer Ein-Wert-Parameter

2.2.4 Übersetzer

Ein Übersetzer transformiert eine Entität in eine andere Entität. Entitäten sind entweder Objekte innerhalb der Programmiersprache oder externe Dateien. Objekte können Instanzen von *mutwo*s internen Klassen, dritten Bibliotheken oder nativen Klassen sein. Externe Dateien umfassen z. B. MIDI- oder Textdateien. Transformieren bedeutet entweder eine Veränderung des Inhalts oder eine Veränderung des Formats.

Klassenname	Eingangsentität	Ausgangsentität	Typ
EventToMidiFile	Ereignisinstanz	Standard Midi File (SMF)	Format
MidiFileToEvent	Standard Midi File (SMF)	Ereignisinstanz	Format
PitchPairToCommonHarmonicTuple	Zwei Tonhöheninstanzen	Tonhöheninstanzen	Inhalt
PulseToComplementaryPulsePair	Ereignisinstanz	Zwei Ereignisinstanzen	Inhalt

Tabelle 3: Exemplarische Übersetzer

Übersetzer in *mutwo* folgen einem funktionalem Paradigma, d. h. ein Übersetzer verändert nicht die Eingangsentität (keine Seiteneffekte), sondern erzeugt eine neue, unabhängige Entität. Das vereinfacht das Übersetzen derselben Entität mit unterschiedlichen Übersetzer.

⁴Das "Extended Speech Assessment Methods Phonetic Alphabet" ermöglicht die Darstellung der phonetischen IPA Symbole in ASCII. [X-SAMPA o. D.]

2.2.5 Generatoren

Generatoren liefern (zumeist generische) Daten, die für generative Werke nützlich sein mögen. Generatoren umfassen Funktionen, Klassen, Konstanten oder andere Objekte. Häufig sind Rückgabewerte der Funktionen und Klassen native Objekte der Programmiersprache. Sie können von Benutzer:innen kreativ angewendet werden.

Objekt	Beschreibung
<code>reflected_binary_code</code>	Erzeugt variable Gray-Codes
<code>TUNEABLE_INTERVAL_TUPLE</code>	intonierbare Intervalle nach Marc Sabat
<code>ActivityLevel</code>	Zyklen der Werte 0 und 1 nach Michael Edwards

Tabelle 4: Exemplarische Generatoren

2.3 Entwicklungsstrategien

2.3.1 Programmiersprache

Mutwo ist in der Programmiersprache Python implementiert. Python ist eine interpretierte, höhere, multi-paradigmatische, plattformübergreifende Sprache. Sie wurde 1991 erstveröffentlicht [Brandl u. a. o. D.].

Die Entscheidung für die Implementierung *mutwo* in Python kann in folgenden Argumenten zusammengefasst werden.

1. **Python ist einfach zu lernen und zu benutzen.** Pythons imperativer vereinfachter Syntax ist intuitiv rasch zu begreifen. Als interpretierte Sprache entfällt die Komplexität der Kompilierung. *Mutwo*s Zielgruppe sind nicht primär professionelle Softwareentwickler:innen, sondern Künstler:innen. Deshalb ist eine flache, schnelle Lernkurve sehr bedeutend.
2. **Python ist populär.** Es gibt eine hohe Wahrscheinlichkeit, dass bereits Dritte Probleme gelöst haben, die in einem Moment für Projekte in *mutwo* relevant werden. Die Lösungen können je nach Lizenz verwendet und angepasst werden. Das reduziert die notwendige Entwicklungszeit und erlaubt einen stärkeren Fokus auf die eigentliche

Aufgabe, auf die künstlerische Arbeit. Zweitens ermöglicht eine populäre Sprache eine schnelle Recherche gewöhnlicher Probleme im Netz. Besonders für Anfänger:innen kann die Einstiegshürde damit signifikant gesenkt werden.

3. **Python ist plattformübergreifend.** Weil entsprechend der *Motivation und Absicht* die Bibliothek *mutwo* plattformübergreifend verwendbar sein soll, ist eine unterliegende Technologie notwendig, die das unterstützt.

2.3.2 Strukturierung des Quellcodes

*Mutwo*s Quellcode ist nach rigiden Regeln strukturiert. Die Struktur basiert auf Pythons System von verschachtelten Modulen, Importe und Paketen. Die Strenge der Struktur folgt zwei Absichten. Erstens soll sie für Benutzer:innen einfach – da konsistent und repetitiv – verwendbar sein. Zweitens vereinfacht sie die Entwicklung und Instandhaltung eines komplexen Softwareprojekts.

Der Paketname der Bibliothek ist *mutwo*. Das Paket *mutwo* ist in unterschiedliche Module geteilt. Die unterschiedlichen Module korrelieren mit den zuvor beschriebenen elementaren Bestandteilen *mutwo*s. Sie werden flankiert von zusätzlichen Hilfsmodulen.

Modulname	Modulbeschreibung
<code>configurations</code>	Globale modulübergreifende Konfigurationsvariablen
<code>constants</code>	Globale modulübergreifende Konstanten
<code>converters</code>	Import und Export von Daten, Übersetzen interner Strukturen
<code>events</code>	Definition verschiedener Ereignisklassen
<code>generators</code>	Generierung von Daten für künstlerische Arbeiten
<code>parameters</code>	Klassen, deren Instanzen Ereignisattributen zugeordnet werden
<code>version</code>	Versionsdefinition des Moduls
<code>utilities</code>	Hilfsmethoden, Errordefinition

Tabelle 5: Moduldefinitionen

Module oder Pakete können in Python auf unterschiedliche Weisen importiert werden. Die folgende Zeile dokumentiert die in *mutwo* bevorzugte Weise:

```
>>> from mutwo import parameters
```

Werden Module auf diese Weise importiert, genügt ein einziger Aufruf Pythons Punktoperator, um Zugriff auf öffentliche Objekte des Moduls zu erhalten.

```
>>> # Direkter Zugriff auf die Klasse "WesternVolume"
>>> my_volume = parameters.WesternVolume("p")
```

Mutwos Module dürfen eine limitierte Anzahl expliziter Teilmodule enthalten. Der Zugriff auf Objekte dieser Teilmodule gestaltet sich (als Ausnahme) komplexer; eine Wiederholung des Punktoperators ist notwendig.

```
>>> # Der Punktoperator ist zweimal notwendig, einmal für Zugriff
>>> # auf das "configurations" Teilmodul und dann für Zugriff auf
>>> # die globale Variable "DEFAULT_CONCERT_PITCH".
>>> print(parameters.configurations.DEFAULT_CONCERT_PITCH)
440
```

Abgesehen von den wenigen Teilmodulen genügt ein Punktoperator, um auf Objekte zuzugreifen. Weil die Regel für alle *mutwo* Module gilt, ist ihre Struktur für neue Benutzer:innen einfach zu verstehen. Sie korreliert mit der fünften Zeile des *Zen of Python*:

“Flat is better than nested.” [Peters 2004]

Folgende Tabelle beschreibt alle erlaubten Teilmodule eines Moduls:

Submodulname	Modulbeschreibung
abc	<i>Abstrakte Basisklassen</i>
configurations	Globale modifizierbare Variablen zur Modulkonfiguration
constants	Globale Konstanten des Moduls

Tabelle 6: Submoduldefinitionen

Die Strukturierung des Quellcodes in thematisch getrennte Module (mit wenigen Teilmodulen) ist aber unzureichend. Weil die grundsätzliche Designprämisse von sehr generischen Strukturen ausgeht, die aber präzise spezifiziert werden können, ist der potenzielle Umfang der Bibliothek schwer fasslich. In *mutwo* ist das Problem durch eine modulare Struktur von

thematisch getrennten Paketen gelöst. Jedes Paket hat einen einzigartigen Namen, hat eine unabhängige Version (und Versionskontrolle), kann eigene Abhängigkeiten definieren und ist je nach Abhängigkeitsstruktur unabhängig von anderen Paketen installierbar.

Die modulare Strukturierung in separate Pakete hilft nicht nur der Entwicklung und Instandhaltung, sondern ermöglicht auch Nutzer:innen nur diejenigen Programmbestandteile zu installieren, die für ein bestimmtes Projekt benötigt werden. Das macht die Bibliothek leichter. Mit der modularen Struktur können Dritte unkompliziert die Bibliothek durch weitere Funktionen erweitern. Sie können einfach ein neues Paket dem *mutwo* Ökosystem hinzufügen.

Technisch ist die Modularität durch Pythons Unterstützung von *namespace packages* gelöst. Das ermöglicht voneinander unabhängigen Pakete die Installation von Quellcode unter einem gemeinsamen Paketnamen.

Einzelne Pakete im *mutwo* Ökosystem sind auf standardisierte Weise benannt. Ihr Name setzt sich durch das Wort *mutwo* und einem Begriff für die enthaltenen Funktionen zusammen.

<i>mutwo.core</i>	Kunstform-, medien- und kulturagnostische Objekte
<i>mutwo.music</i>	Musikspezifische Objekte
<i>mutwo.reaper</i>	Funktionen, die mit der DAW Reaper zusammenhängen

Das *mutwo* Ökosystem setzt die anfänglich beschriebene Struktur von Modulen und Teilmodulen auch in separaten Paketen um. Die Namen der Module sind Kompositionen aus einem Präfix und einem Suffix. Der Präfix ist der Suffix des Paketnamen (z. B. *core* oder *music*). Der Suffix beschreibt die Funktion des Moduls (z. B. *events* oder *parameters*). Präfix und Suffix sind durch einen Unterstrich getrennt.

Paketidentität (Präfix)	Modulfunktion (Suffix)	Modulname
core	parameters	core_parameters
music	events	music_events
midi	converters	midi_converters

Tabelle 7: Exemplarische Module

Das Importieren im Quellcode funktioniert auf gleiche Weise wie oben beschrieben:

```
>>> from mutwo import core_parameters
>>> from mutwo import music_events
>>> from mutwo import midi_converters
```

2.3.3 Abstrakte Basisklassen

Wie in *Strukturierung des Quellcodes* vorgestellt, enthält jedes *mutwo* Modul potenziell das *abc*⁵ Teilmodul. In diesen Modul werden abstrakte Klassen definiert. Abstrakte Klassen sind Klassen, deren Methoden oder Attribute nicht oder nur stellenweise implementiert sind. Abstrakte Klassen können deshalb nicht initialisiert werden. Von abstrakten Klassen können unterschiedliche dritte Klassen erben, die fehlende Teile implementieren. Sind alle fehlende Teile implementiert, kann eine abgeleitete Klasse initialisiert werden [*Abstract type* o. D.].

In *mutwo* ermöglichen abstrakte Klassen die Definition der öffentlichen Schnittstelle (API) einer Programmkomponente. Zusammenhängende Programmkomponente erwarten voneinander ihre jeweils öffentlich definierte API. Sie funktionieren damit unabhängig von spezifischen Implementierungen der erwarteten anderen Programmkomponente. Diese Technologie unterliegt der im im Kapitel *Parameter* beschriebenen Definition der Parameterklassen. Die im *abc* Teilmodul des *parameters* Modul deklarierte Klassen definieren ihre kompakte Identität über ein nicht-implementiertes (d. h. abstraktes) Attribut.

Parameter	Klassenname	Abstraktes Attribut	Attributdatentyp
Tonhöhe	Pitch	frequency	Gleitkommazahl
Tonhöhenintervall	PitchInterval	interval	Gleitkommazahl
Dauer	Duration	duration	Bruch
Text	Lyric	phonetic_script	Zeichenkette
Lautstärke	Volume	amplitude	Gleitkommazahl

Tabelle 8: Präzision der Liste exemplarischer Ein-Wert-Parameter

⁵i. e. *abstract base classes*

Dritte Bestandteile *mutwos* erwarten von einem bestimmten Parameter nur diese minimaldefinierte Schnittstelle. Das ermöglicht Benutzer:innen die Implementierung einer Repräsentation einer Kategorie, die der jeweiligen Interpretation entspricht. Das Design *mutwos* versichert, dass die von Benutzer:innen hinzugefügten Repräsentationen mit anderen Bibliothekskomponenten kompatibel sind.

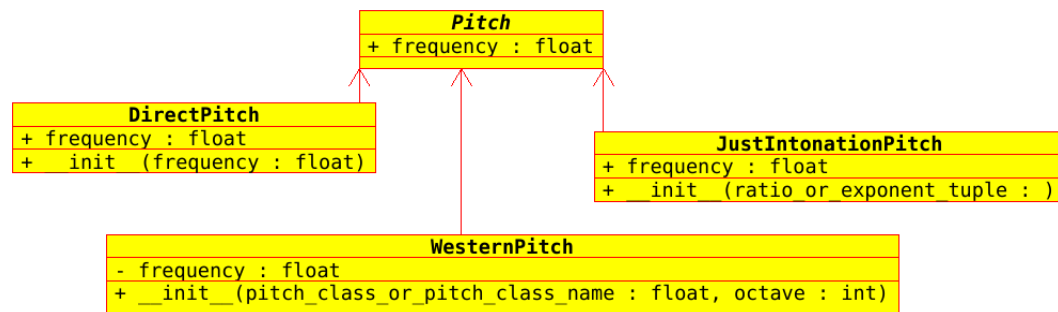


Abbildung 3: Schematische Darstellung eines Parameter und seine unterschiedliche Teilklassen. Die abgeleiteten Klassen ermöglichen unterschiedliche Perspektiven derselben Entität. Sie werden mit unterschiedlichen Argumenten initialisiert.

2.3.4 Konvention vor Konfiguration

Das Entwicklungsparadigma *Konvention vor Konfiguration* entspannt den in *Motivation und Absicht* beschriebenen Konflikt zwischen Anpassbarkeit und Einfachheit. Viele Funktionen und Methoden *mutwos* haben eine große Anzahl potenzieller Argumente. Häufig ist Intention der reichen Argumente das generische, adaptive Designziel zu erfüllen. Benutzer:innen werden für einige Argumente selten eine Notwendigkeit entwickeln, diese explizit zu deklarieren.

In dieser Situation kann genanntes Paradigma helfen. *Konvention vor Konfiguration* empfiehlt, dass Benutzer:innen einer Bibliothek nur unkonventionelle Konfigurationen spezifizieren müssen [*Convention over configuration* o. D.].

Mutwo realisiert diese Empfehlung mithilfe sensibler Voreinstellungen. Viele Argumente haben voreingestellte Standardwerte. Wird kein expliziter Wert deklariert, fallen Funktionen und Methoden auf diese zurück. Dadurch müssen Benutzer:innen nur elementare oder für sie relevante erweiterte Argumente angeben.

Dieses Paradigma birgt die Gefahr bestimmte (Musik-)Traditionen oder Ästhetiken zu priorisieren. Die Gefahr ist darin begründet, dass manche (die den Konventionen entsprechenden) Lösungen einfacher umzusetzen sind als andere. In meinem Verständnis wiegt der Mehrwert einer einfachen, intuitiven und von Standardformulierungen befreiten Anwendung diese Gefahr auf.

2.3.5 Globale Voreinstellungen

Um beschriebene Gefahr einzudämmen, vereinfacht *mutwo* das Überschreiben der Voreinstellungen mithilfe globaler Variablen. Wird kein expliziter Wert deklariert, weisen Funktionen und Methoden während der Laufzeit dynamisch die Werte ihrer Argumente diesen Variablen zu. Wird eine Abweichung einer Konvention notwendig, kann die Konvention mit wenigen imperativen Zeilen überschrieben werden.

Globale Voreinstellungen

Dieses Jupyter Notebook demonstriert die Idee globaler Standardwerte.

```
[2]: from mutwo import music_parameters
```

```
[3]: a4 = music_parameters.WesternPitch('a', 4)
```

```
[4]: print(a4.frequency)
```

440.0

Der Standardwert für den Kammerton kann global konfiguriert werden.

```
[5]: music_parameters.configurations.DEFAULT_CONCERT_PITCH = 443
```

Wird danach eine neue Tonhöhe initialisiert, wird *mutwo* den Kammerton dynamisch anpassen.

```
[6]: a4_443 = music_parameters.WesternPitch('a', 4)
```

```
[7]: print(a4_443.frequency)
```

443.0

Der Kammerton kann aber auch explizit als Eingangswert wieder überschrieben werden.

```
[8]: a4_440 = music_parameters.WesternPitch('a', 4, concert_pitch=440)
```

```
[9]: print(a4_440.frequency)
```

440.0

Daneben unterstützt diese Strategie eine flexible Arbeitsweise, die unmittelbar auf Veränderungen reagiert. Ein Beispiel ist ein Szenario einer Komposition mit Instrumenten alter Musik und einer Klangdatei. Der Kammerton Instrumente alter Musik variiert. Wird die Klangdatei in *mutwo* erzeugt, kann der Kammerton an unterschiedliche Instrument angepasst werden.

2.3.6 Dokumentation öffentlicher Schnittstellen

Im Quellcode *mutwos* sind die für Dritte intendierten Objekte dokumentiert. Mithilfe der Software *Sphinx* wird automatisiert aus dem Quellcode eine Dokumentation erzeugt. Das unterstützt die Transparenz vielfältiger Argumente und Konfigurationsvariablen. Im Appendix oder im Web kann die automatisierte Dokumentation nachvollzogen werden ⁶.

2.3.7 Unspezifische Ereignisse; dynamische Akzessoren

Kapitel *Abstrakte Basisklassen* beschreibt, wie *mutwos* Parametermodell individuelle Repräsentationen ermöglichen. Adäquat dazu werden Ereignisse möglichst generisch behandelt. Entwickler:innen werden nicht zur Verwendung bestimmten Formen spezifischer Ereignisse (z. B. Pause, Akkord, Note, Takt) gezwungen. *Mutwos* Vertrag erwartet nur, dass Benutzer:innen Instanzen der Ereignisbasisklassen oder Instanzen davon abgeleiteter Klassen verwenden. Die Ereignisbasisklassen sind die in Kapitel *Ereignisse* beschriebenen Klassen `SimpleEvent`, `SequentialEvent` und `SimultaneousEvent` ⁷.

Die Abwesenheit spezifischer Ereignisklassen ist mithilfe Pythons Unterstützung von Introspektion, Funktionen höherer Ordnung und dynamischen Attributen möglich. Spezifische Klassen mögen darin begründet sein, dass dritte Programmkomponente bestimmte Attribute oder Methoden von ihren Eingangswerten erwarten (müssen). Indem dritte Programmkomponente als Funktionen höherer Ordnung implementiert sind, kann *mutwo* diese Anforderung mit geringeren Einschränkungen von unterstützten Datentypen erfüllen.

⁶Aufgrund der hohen Seitenanzahl der PDF Version der API-Dokumentation wurde bei der gedruckten Version vorliegender Arbeit auf das Anhängen der Dokumentation verzichtet.

⁷Übersetzer und andere *mutwo* Objekte benötigen Basisklassen, um Blattknoten eines Ereignisbaums erkennen zu können und um zwischen akkumulierenden und parallelen Ereignisbehälter unterscheiden zu können. Zukünftig könnte das Problem mit dynamischen Akzessoren und standardisierten Rückfallwerte gelöst werden. Dann müsste *mutwos* Vertrag nur das Erben der generischen Klasse `Event` fordern.

Unspezifische Ereignisse

Funktionen höherer Ordnung

Dieses Jupyter Notebook demonstriert den Umgang mit Funktionen höherer Ordnung in dritten Programmkomponenten.

Die exemplarische dritte Programmkomponente ist hier ein Übersetzer von Ereignisse in MIDI Dateien.

Zuerst müssen die spezifischen Module geladen werden.

```
[2]: from mutwo import core_events
     from mutwo import midi_converters
     from mutwo import music_parameters
```

Als nächstes wird für dieses Beispiel die vereinfachte Klasse *Note* definiert. Sie soll eine Note im Kontext westlicher Musik repräsentieren.

```
[3]: class Note(core_events.SimpleEvent):
     # Dynamik ist konstant
     volume = music_parameters.WesternVolume('p')

     def __init__(self, pitch, duration):
         self.pitch = pitch

         # Das ist Python spezifischer Syntax, um dem Konstruktor
         # der Basisklasse das "duration" Argument zu übermitteln.
         super().__init__(duration)
```

Jetzt wird eine einfache Melodie aus zwei Noten definiert.

```
[4]: melody = core_events.SequentialEvent(
     [
         Note(music_parameters.WesternPitch('c'), 1),
         Note(music_parameters.WesternPitch('d'), 1),
     ]
 )
```

Jetzt soll die Melodie in eine MIDI Datei übersetzt werden.

In der API Dokumentation der Klasse *EventToMidiFile* kann nachgelesen werden, dass diese unter anderem mit dem Argument *simple_event_to_pitch_list* initialisiert wird. Hier ist auch dokumentiert, dass dessen Standardwert davon ausgeht, dass einem *SimpleEvent* ein Attribut namens *pitch_list* zugewiesen ist. Weil *Note* das Attribut nicht kennt, muss das Argument überschrieben werden, sodass es das *pitch* Attribut der Klasse *Note* finden kann.

```
[5]: # Definiere zuerst den Übersetzer
     event_to_midi_file = midi_converters.EventToMidiFile(
         simple_event_to_pitch_list=lambda simple_event: [
             getattr(simple_event, 'pitch')
         ]
     )
     # Übersetze jetzt die Melodie
     event_to_midi_file.convert(melody, 'my_melody.mid')
```

Neben der Ereignisinstanz sind die Eingangswerte dritter Programmkomponente Funktionen. Diese Funktionen erhalten wiederum später die Ereignisinstanz als Eingangswert. Als Ausgangswert müssen sie ein bestimmtes erwartetes Attribut zurückgeben. Über den Umweg der als Eingangswert mitgegebenen Funktion, kann die dritte Programmkomponente somit versichern, ein spezifisches Attribut zu erhalten.

Python ermöglicht dynamische Zuweisungen von Attributen. Ein zusätzliches Entwurfsmuster in *mutwos* Ereignismodell ist damit die ad-hoc Zuordnung benötigter Attribute. In manchen Fällen bedingt das elegantere Lösungen als das Schreiben einer neuen oder das Modifizieren einer bestehenden Klasse.

Unspezifische Ereignisse

Dynamische Attribute

Dieses Jupyter Notebook demonstriert den Umgang mit dynamischen Attributen.

Die MIDI Datei wurde im vorgehenden Beispiel erfolgreich erzeugt. Die erzeugte MIDI Datei enthält *Note On* und *Note Off* Nachrichten, denen eine bestimmte *velocity* zugeordnet ist. Allerdings können MIDI Dateien auch noch weitere Nachrichten enthalten, wie z. B. Kontrollnachrichten. In der API Dokumentation der Klasse *EventToMidiFile* kann das Argument *simple_event_to_control_message_tuple* gefunden werden.

Ist jetzt intendiert, dass die MIDI Datei auch Kontrollnachrichten (z. B. zur Steuerung der Klangsynthese) enthält, könnte eine neue *Note* Klasse definiert werden. Aber vielleicht brauchen die meisten Noteninstanzen keine Kontrollwerte. In dem Fall mag es eleganter sein, bestimmten Noten in der Melodie dynamisch Kontrollnachrichten hinzuzufügen.

```
[6]: # Importiere das externe mido Paket, um Kontrollnachrichten
# initialisieren zu können. Mutwo verwendet auch mido um MIDI
# Dateien zu lesen und zu schreiben.
import mido
# Ordne jetzt der ersten Note Kontrollwerte zu.
# Siehe die mido Dokumentation bezüglich der Initialisierung
# von Nachrichteninstanzen.
melody[0].control_message_tuple = (
    mido.Message("control_change", channel=0, control=10, value=127),
    mido.Message("control_change", channel=0, control=11, value=64),
)
# Übersetze jetzt die Melodie mit den Kontrollnachrichten.
event_to_midi_file.convert(melody, 'my_controlled_melody.mid')
```

Die Komplexität unspezifischer Ereignisse wird mit der Strategie *Konvention vor Konfiguration* abgefangen. Das Paket `mutwo.music` implementiert die Klasse `NoteLike`. Die Klasse `NoteLike` ist von der gleichnamigen Klasse der Bibliothek *SCAMP* beeinflusst. Sie repräsentiert ein diskretes musikalisches Ereignis mit keiner, einer oder mehreren Tonhöhen (ein Ton, eine Pause, ein Akkord). Sie mag ausreichen für viele Anwendungsfälle musikbezogener Funktionen *mutwos*. Deshalb sind alle Standardwerte so gesetzt, dass sie den Feldern der Klasse `NoteLike` entsprechen. Das gewährleistet, dass das avancierte, komplexere Potenzial *mutwos* Ereignismodell nur verstanden und angewandt werden muss, wenn es die projektspezifischen Anforderungen bedingen.

2.3.8 Annotation von Typen im Quellcode

In Python muss der Datentyp von Variablen, Argumente oder Rückgabewerte nicht spezifiziert werden. Seit Python 3.5 werden aber optionale Annotationen der Datentypen unterstützt [*typing - Support for type hints* o. D.]. Die Annotationen haben keinen Einfluss in der Programmlaufzeit. Sie ermöglichen aber in der Entwicklungsphase dritten Programmen inkonsistente Argumenttypen im Quellcode zu markieren. Das mag einer frühzeitigen Vermeidung bestimmter Fehler helfen [Rossum und Levkivskyi 2014].

Die Argumente und Rückgabewerte der Funktionen und Methoden *mutwos* sind mit ihrem Datentyp annotiert. Neben dem Mehrwert eines fehlerreduzierten Programmes, ist primäre Absicht der Annotation Kommunikation. Die Kommunikation ist an Entwickler:innen und Benutzer:innen *mutwos* gerichtet. Aufgrund der unspezifischen Struktur provoziert *mutwo* projektspezifische Adaptionen. Um Anpassungen vornehmen zu können müssen Dritte ein klares, schnelles Verständnis eines zu erweiternden Objekts gewinnen können. Annotationen helfen hierbei unmittelbar zu begreifen, welche Art von Daten operiert werden. Sie befreien von mühevoller Analyse des Quellcodes, um z. B. nachzuvollziehen, welchem Datentyp der Rückgabewert einer Funktion entspricht.

2.3.9 Konsistente Namenskonventionen

Mutwos Quellcode verfolgt eine konsistente Benennung von Objekten. Die Benennung ist einerseits in Konventionen der Sprache Python begründet (z. B. Klassennamen sind Binnenversalien). Andererseits werden diese Konventionen mit zusätzlichen Regeln erweitert.

1. **Abkürzungen müssen vermieden werden.** Abkürzungen verdecken die Bedeutung von Variablen. Wollen Dritte einen Quellcode oder eine Schnittstelle verstehen, erhöhen Abkürzungen die Einstiegshürde.
2. **Variablen müssen niemals im Plural sein.** Stattdessen muss für einen Behälter der Datentyp des Behälters angegeben werden. Exemplarisch soll eine Liste von Tonhöhenobjekte deshalb nicht `pitches`, sondern `pitch_list` benannt werden. Das vermittelt unmittelbar Eigenschaften über die Objektsammlung (z. B. ob sie modifizierbar ist).
3. **Modulnamen sollten im Plural sein.** Einige Standardmodule Pythons stehen im Plural (z. B. `fractions`, `collections`, `types`). Das ist als Konvention von diesen Modulen abgeleitet. Mithilfe der vorhergehenden Regel sind Module am Namen erkennlich.

Die Begründung für die ausführliche, wortreiche Benennung entspricht der Begründung für die *Annotation von Typen im Quellcode*. Möglichst eindeutig und klar für Lesende zu sein wird als entscheidender gewichtet als ein reduzierter Text.

2.4 Limitierungen und Grenzen

Die in *Komposition und Werkzeuge (II)* verwendeten Begriffe Utopie und Versuch implizieren das Scheitern der Wirklichkeit. Sie implizieren zugleich ein kontinuierliches Weiter, in dem Scheitern Teil eines Zyklus ist. Zentral wird ein Konflikt zwischen Wirklichkeit (Pragmatismus) und Vorstellung (Idealismus) verhandelt. Die anspruchsvolle Spezifikation *mutwos*, besonders der generischen Strukturen, steht projektbezogenen Arbeiten (wegen den damit verbundenen Fristen) unverträglich entgegen.

In Version 0.17.1 des Pakets `mutwo.music` wird der Parameter `PlayingIndicator` definiert. Die Klasse `PlayingIndicator` fungiert als Sammelbegriff verschiedener Spieltechni-

ken wie Artikulationen, Ornamente oder Flageolets. Die jeweilige Spieltechnik ist als eigene Klasse implementiert, die von der Klasse `PlayingIndicator` erbt. Beschriebene Spieltechniken stehen in enger Beziehung zu europäischer Notation. Europäische Notation kann im *mutwo* Ökosystem gegenwärtig nur mit dem Paket `mutwo.abjad` erzeugt werden. Mit `mutwo.abjad` können *mutwo* Datenstrukturen in Datenstrukturen der Bibliothek *abjad* übersetzt werden. *Abjad* ist ein Adapter der Notationssoftware *Lilypond* [About *Abjad* o. D.]. Die Repräsentationen *abjads* stehen deshalb *Lilyponds* Syntax und Befehle sehr nahe. Diese Verkettung von Eigenschaften bedingt, dass verschiedene Spielindikatoren *mutwos* Syntax und Eigenheiten von *Lilypond* widerspiegeln. Um ein Objekt der Klasse `WoodwindFingering` zu initialisieren sind z. B. die Argumente `cc`, `left_hand` und `right_hand` erforderlich. Die Argumente erwarten jeweils einen Tupel, der mit Zeichenketten gefüllt ist. Diese Form der Repräsentation von Holzbläsergriffen entspricht dem Befehl in *Lilypond*. Eine so direkte Beziehung verletzt die Intention *mutwos*. Die korrekte Umsetzung des oberen Beispiels könnte eine Implementierung einer einfachen, verständlichen, dokumentierten Repräsentation von Holzbläsergriffen umfassen. Die innere Repräsentation *mutwos* wäre dann so gestaltet, dass sie offen und neutral gegenüber unterschiedliche dritte Formate wie *MusicXML*, *MSCX* oder *Lilypond* wäre. Ursache der ungenügenden Situation ist eine Mischung aus einer projektspezifischen Notwendigkeit für eine bestimmte, noch abwesende Funktion und der projektspezifischen Einschränkung der Ressource Zeit.

Im abstrakten Ereignismodell *mutwos* sind Ereignisse Kompositionen einer unbestimmten, benutzerdefinierten Menge von Vektoren. In Version 0.61.6 des Pakets `mutwo.core` enthalten Ereignisse immer nur den Vektor `Dauer`. In der initialen Entwicklung *mutwos* enthielt das Ereignismodell nur Ereignisse mit Dauern. Erst in kürzlichen Projekten (einer räumlichen Laufpartitur und einer algorithmischen Zeichnung) verstand ich die Notwendigkeit, Ereignisse generischer zu beschreiben.

Mutwo enthält und enthielt eine Vielzahl solcher Beispiele. Die Entwicklungsstrategie ist die ad-hoc Realisierung eines Prototypen in einem projektspezifischen Kontext und anschließender Verbesserung.

Das zweite Beispiel macht auf eine weitere Schwierigkeit aufmerksam. Sie hängt mit der gerin-

gen Anzahl Entwickler:innen und dem jungen Lebensalter des Projekts *mutwo* zusammen. Das Design *mutwo*s fordert die Beschreibung generischer Strukturen (z. B. eine allgemeine Tonhöhenklasse). Zugleich sollen die generischen Strukturen spezifiziert werden, um konkrete Vorstellungen repräsentieren zu können (z. B. eine Tonhöhenklasse im Sinne europäischer Musiktheorie oder eine Tonhöhenklasse für die reine Stimmung). Die Konkretisierungen, die in *mutwo* bereits implementiert sind, vereinfachen die Realisierung bestimmter Vorstellungen. Sie erschweren damit zugleich die abwesenden Konkretisierungen, welche von Benutzer:innen autark entwickelt werden müssen. Damit bevorzugt *mutwo* indirekt bestimmte Ästhetiken, Vorstellungen oder Traditionen, nämlich solche, die von mir benötigt werden. Das Ungleichgewicht kann niemals aufgelöst werden. Es kann aber ausgeglichen werden, wenn die Anzahl der Entwickler:innen der Bibliothek *mutwo* zunehmen. Eine höhere Anzahl Mitwirkender verdichtet die Menge unterschiedlicher Weltvorstellungen; die Bibliothek wird damit diverser und unparteiischer. Dieses Argument ist nicht nur für Ereignisse und Parameter, sondern auch für Übersetzer valide. Bestehende Übersetzerpakete wie *mutwo.mbrola* (Sprachsynthese), *mutwo.csound* (Klangsynthese) oder *mutwo.ekmelily* (mikrotonale Notation mit *Lilypond*) spiegeln Anforderungen meiner eigenen künstlerischen Arbeit wieder, die nicht notwendigerweise Anforderungen Dritter entsprechen.

2.5 Fallbeispiele

2.5.1 *thanatos trees for Tim Pauli*

Das erste Werk, welches mit *mutwo* entstand, ist die Klanginstallation *thanatos trees for Tim Pauli*. Sie besteht aus einer Klangdatei mit 15 bis 17 Audiokanäle, die über 25 Lautsprecher abgespielt wird. Die Lautsprecher sind an Ästen dünner Stämme befestigt. Die Stämme sind mithilfe von Baumständer aufgestellt.

Ausgangspunkt der Arbeit war eine Reflexion über die Skalen gemeinsamer Produktmengen (*common-product set scales* oder kurz CPS Skalen) des US-amerikanischen Stimmungstheoretiker Erv Wilson.

Eine CPS Skala nach Wilson entsteht durch die Multiplikation von n Faktoren aus einer Menge S . Für z. B. $n = 2$ und $S = \{1, 3, 5, 7\}$ besteht der Modus aus den sechs Partialtönen $1 * 3 = 3$,



Abbildung 4: *thanatos trees* for Tim Pauli im LTK₄ in Köln.

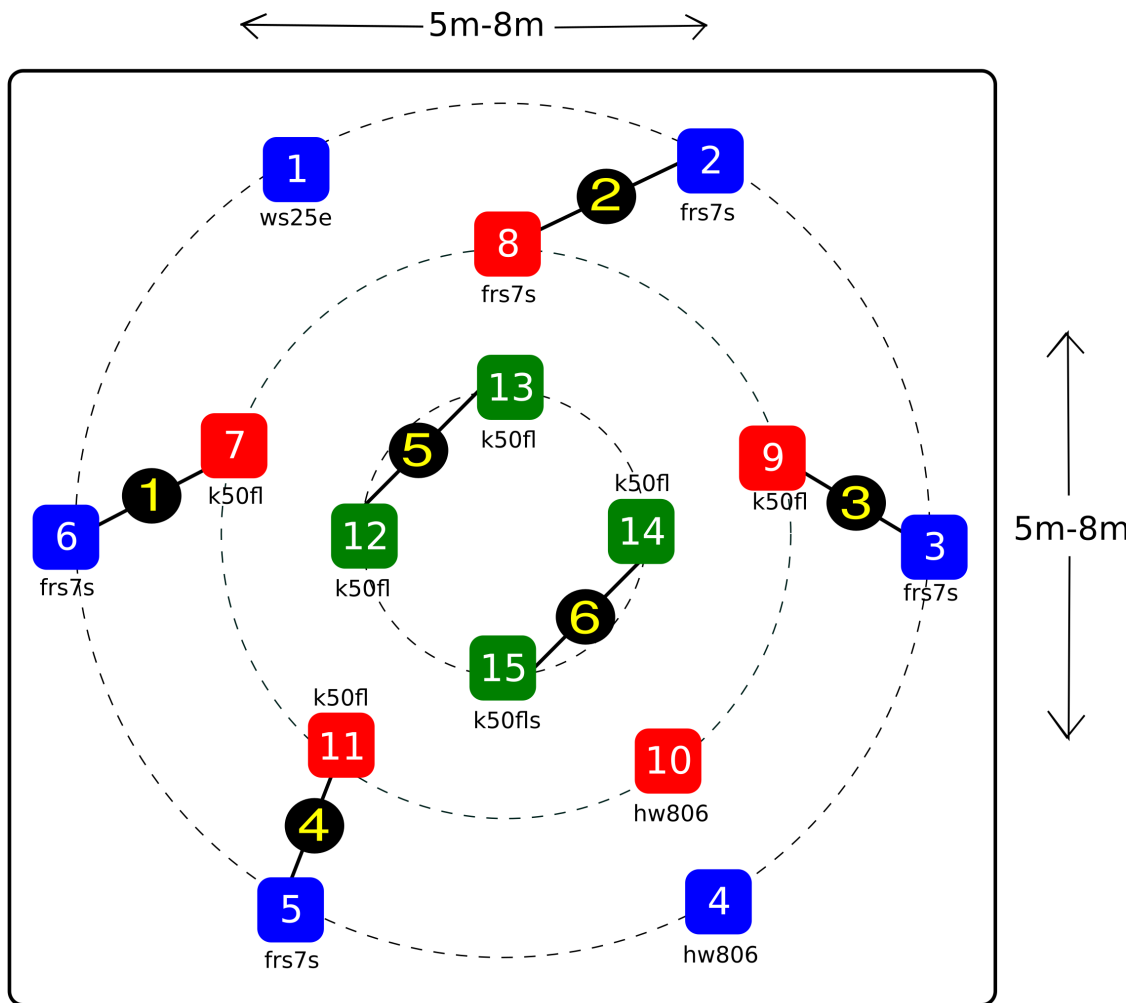
$1 * 5 = 5$, $1 * 7 = 7$, $3 * 5 = 15$, $3 * 7 = 21$ und $5 * 7 = 35$ [Narushima 2018, S. 150f].

Ich bemerkte, dass eine Überlagerung verwandter CPS Skalen eine symmetrische, zyklische Struktur beschreiben kann. Die Struktur besteht aus verschiedenen Gruppen von Harmonien, die über gemeinsame Tonhöhen verbunden sind. Jede Harmonie besteht aus einer einzigartigen Menge von Tonhöhen. Alle Harmonien enthalten dieselben Intervalle, aber in anderen Lagen oder Umkehrungen. Die symmetrische, dezentrale, ambulante Struktur deutete für mich darauf hin, dass sie in einer installativen Situation (d. h. ohne Anfang und Ende) konsistenter erfahrbar wäre, als in einem konzertanten Kontext.

Mithilfe von *mutwo* wollte ich eine Klangdatei erzeugen, die in Dauerschleife abgespielt werden würde. Eine Schleife schien adäquat für die zyklischen Eigenschaft der Struktur. Die Klangdatei sollte primär aus Sinusoiden bestehen. Jedem Sinusoiden sollte exakt eine Tonhöhe einer Harmonie und einen Lautsprecher zugewiesen werden.

Der Quellcode des Projekts ist in vier Teilmodule gegliedert.

1. **classes.** Verschiedene projektspezifische Klassen werden hier unsortiert definiert. Das umfasst Ereignisse, Generatoren und Parameter.
2. **constants.** Hier werden einerseits die globalen Vorbedingungen der Komposition de-



Legende



-  Lautsprecher (Kanalnummer, Lautsprechermodell)
ws25e
-  Baumstamm mit Ast (Höhe 2-3m)

Abbildung 5: Skizze zum Aufbau der Installation.

finiert (z. B. die verwendeten Primzahlen oder Dateipfade). Andererseits werden aus den gegebenen Vorbedingungen konkrete Instanzen der zuvor beschriebenen Klassen initialisiert.

3. **converters.** Die Übersetzung unterschiedlicher interne Repräsentationen (von einer groben zu einer feineren Auflösung) sind implementiert. Die feinste Auflösungen wer-

den in Klangdateien übersetzt. Auch diese Übersetzungen sind als Klassen definiert.

4. **synthesis.** In dieser Kategorie finden sich der Quellcode für die Klangsynthesesprache *Csound*. Dieser Quellcode wird von den Übersetzer benutzt, um die Klangdateien zu generieren.

Die wichtigste kompositorische Identität findet sich in den Kategorien *converters* und *constants*. Abbildung 6 zeigt schematisch, wie die Arbeit als eine Entwicklung unkonkreter (grob aufgelöster) Ereignisse in konkretere (fein aufgelöste) Ereignisse konzipiert ist. Das Element *Weather* in der Abbildung beeinflusst die Übersetzung der *Partial* Instanzen in die *Vibration* Instanzen. Der Zyklus der harmonischen Gruppen wird in der Klangdatei mehrmals wiederholt. Jede Repetition variiert die Realisierung einer Gruppe. Variiert werden Eigenschaften wie präsente Frequenzbereiche oder das Ein- und Ausschwingverhalten einzelner Töne. Ursache der Variation ist eine Instanz der *Weather* Klasse. Die Idee ist, dass im Verlauf der Installation dasselbe Objekt unter verschiedenen Umwelteinflüsse (unterschiedlichem Wetter) erfahrbar wird.

Mutwos innere generische Repräsentationen von Ereignissen waren für den Projektverlauf entscheidend. Während anfänglich die Prämisse bestand nur Sinusoide zu verwenden, wurde im Arbeitsprozess deutlicher, dass auch weitere Klangquellen bereichernd sein könnten. Aufgrund der inneren abstrakten Repräsentation war es unmittelbar möglich dieselben Daten in MIDI Dateien, Textpartituren für IRCAMs Gesangssynthesesoftware *ISiS* und Zeichenketten im Markierungsformat des Programmes *Reaper* zu übersetzen. Letztgenannte Übersetzung war wertvoll, um globale Entscheidungen mithilfe einer visuellen Rückkoppelung besser begreifen zu können.

Ein Teilproblem in dem Projekt war die Verteilung von Dauern auf Gruppen und ihre einzelne Partialtöne. Sich anschließende Gruppen überlappen sich. Jede Gruppe besteht deshalb aus einem überlappenden Teil mit der vorhergehenden Gruppe, einem solistischen Teil und einem überlappenden Teil mit der anschließenden Gruppe. Jede Gruppe hat nur eine bestimmte Menge von erlaubten Dauern, die aus der harmonischen Struktur der Gruppe abgeleitet ist.

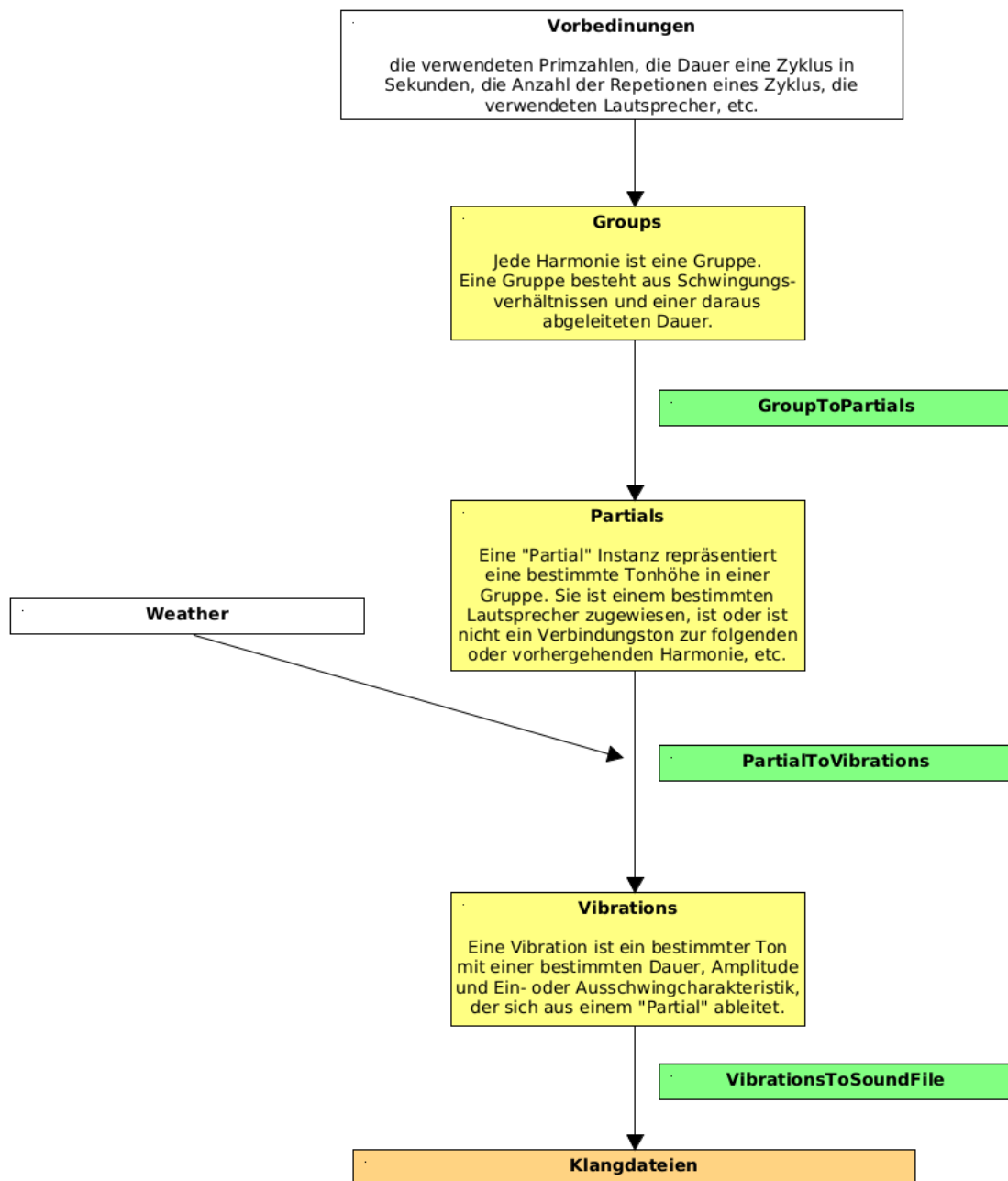


Abbildung 6: Schematischer Aufbau der Programmstruktur von *thanatos trees for Tim Pauli*. Gelb markierte Elemente repräsentieren Ereignisse, grün markierte Elemente sind Übersetzer.

Beschriebens Teilproblem kann mit Constraintprogrammierung gelöst werden. Constraintprogrammierung ist ein Paradigma mit dem kombinatorische Probleme gelöst werden können. Benutzer:innen deklarieren Einschränkungen für erlaubte Lösungen einzelner Variablen. Ein Algorithmus versucht aus diesen Einschränkungen eine oder mehrere Kombinationen erlaub-

ter Werte für alle definierte Variablen zu finden [*Constraint programming* o. D.].

Die von Google in C++ entwickelte, quelloffene Software *OR-Tools* zur Lösung kombinatorischer Optimierungsprobleme unterstützt Constraintprogrammierung. Die Software hat Adapter für die Sprachen Java, C# und Python [*About OR-Tools* o. D.]. Die Popularität der Programmiersprache der Realisierung des Projekts *thanatos trees for Tim Pauli* erlaubte so unmittelbaren Zugriff auf avancierte Algorithmen des umfassend dokumentierten Programmes *OR-Tools*. Gegebenes Beispiel ist exemplarisch für einen bestimmten Mehrwert in der Wahl Python als *mutuos* unterliegende *Programmiersprache*.

2.5.2 *ohne Titel (2)* und *ohne Titel (3)*

Die konzertanten Kompositionen *ohne Titel (2)* und *ohne Titel (3)* sind zeitnah und überschneidend innerhalb weniger Monate entstanden. Beide haben eine Dauer von etwa 45 Minuten, entfalten sich in einer langsamen Form, spezifizieren für geräuscharme Klänge präzise Intonationen, sind kammermusikalisch besetzt und kombinieren Instrumente mit einer mehrkanaligen Klangdatei (die in beiden Werken aus synthetisierten Klängen und Feldaufnahmen besteht). Die Instrumentalstimmen beider Werke sind überwiegend in Form von *time-brackets* nach John Cage notiert.

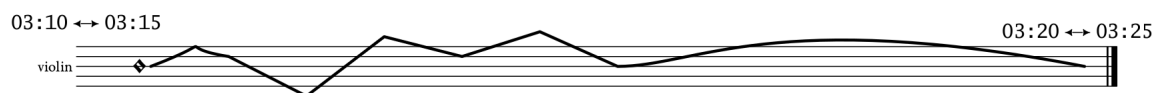


Abbildung 7: Exemplarisches *time-bracket* der Violinstimme von *ohne Titel (3)*.

Time-brackets sind einzeln notierte Abschnitte, die eine variable Anfangs- und Endzeit spezifizieren. Musiker:innen können autonom entscheiden, wann sie innerhalb gegebener Abschnitte notierte Klänge beginnen und beschließen [Weisser 2013, S. 179f].

Neben *time-brackets* gibt es in beiden Arbeiten (quantitativ reduzierte) Passagen synchronisierter Notationen (als Partitur). Die synchronisierten Abschnitte basieren auf bestehende Literatur; in *ohne Titel (2)* auf den cantus firmus Lassus' *Quid prodest stulto habere divitias*; in

ohne Titel (3) auf den Westminsterschlag. Statt eines Startbereiches gibt es für synchronisierte Abschnitte eine explizite Startzeit, statt eines Endbereiches eine Tempoangabe.

Die einzelnen Stimmbücher der Musiker:innen können ausgedruckt oder mithilfe eines PDF-Lesers angezeigt werden, um mit einer Stoppuhr gespielt zu werden. Alle Stimmbücher werden aber in beiden Arbeiten alternativ als Videodateien bereitgestellt. In den Videodateien wechseln die angezeigten Abschnitte im Verlauf der Zeit automatisch, auf eine Stoppuhr kann verzichtet werden.

Abbildung 8 zeigt schematisch die Organisation des Quellcodes zum Erzeugen der Notation, Videos und Klangdateien. Zentrum der Struktur ist eine Instanz der Ereignisklasse `TimeBracketContainer`. Das Objekt sammelt alle `TimeBracket` – Instanzen aller Stimmen (elektronisch und instrumental)⁸. Sie werden über die `register` Routine dem `TimeBracketContainer` zugefügt. Abschließend wird der `TimeBracketContainer` in unterschiedliche Ausgangsformate übersetzt.

Die eigentlichen *time-brackets* sind variabel gewonnen. Wichtigste Quelle ist eine Instanz der Ereignisklasse `FamilyOfPitchCurves` und verbundene Übersetzer. Eine `FamilyOfPitchCurves` beschreibt Wahrscheinlichkeitsverläufe einzelner Intonationen über die Dauer eines Stückes. In *ohne Titel (3)* pendelt die Harmonie zwischen zwei Extrema, die aus potenziellen Flageolets beider Instrumente abgeleitet sind.

In *ohne Titel (2)* interpolieren Wahrscheinlichkeitsverläufe zwischen harmonischen Ankerpunkte, die Transpositionen des *cantus firmus* entsprechen.

Übersetzer erzeugen `TimeBracket` Instanzen aus einem `FamilyOfPitchCurves` – Objekt und Start- und Endzeiträume. Von ihrer Basisklasse erben sie eine Funktion, die Wahrscheinlichkeiten der Intonationen der `FamilyOfPitchCurves` abhängig von gegebenen Start- und Endzeiträume und der absoluten Position eines Ereignis in einer *time-bracket* berechnet. Mithilfe dieser Information verfolgen Übersetzer unterschiedliche Ansätze, um Inhalt einer *time-bracket* zu bestimmen: ein Übersetzer findet Akkorde maximaler Harmonizität⁹

⁸In der Implementierung sind die synchronisierten Abschnitte mit expliziten Startwerte und Tempi nur Sonderformen einer allgemeinen `TimeBracket` – Klasse.

⁹Harmonizität beschreibt die Stabilität eines Intervalls. Harmonizität eines Schwingungsverhältnis kann mit Teilbarkeit und Größe seiner Zahlen quantifiziert werden [Barlow 1999, S. 5f].

innerhalb eines definierten Ambitus, ein anderer Übersetzer sucht melodische Verläufe mit minimalen Sprüngen und einem euklidischen Rhythmus¹⁰. Die Ansätze unterliegen selbst Wahrscheinlichkeitsverläufe, die ihre Häufigkeit bestimmen. Diese Wahrscheinlichkeitsverläufe der Häufigkeit sind händisch definiert.

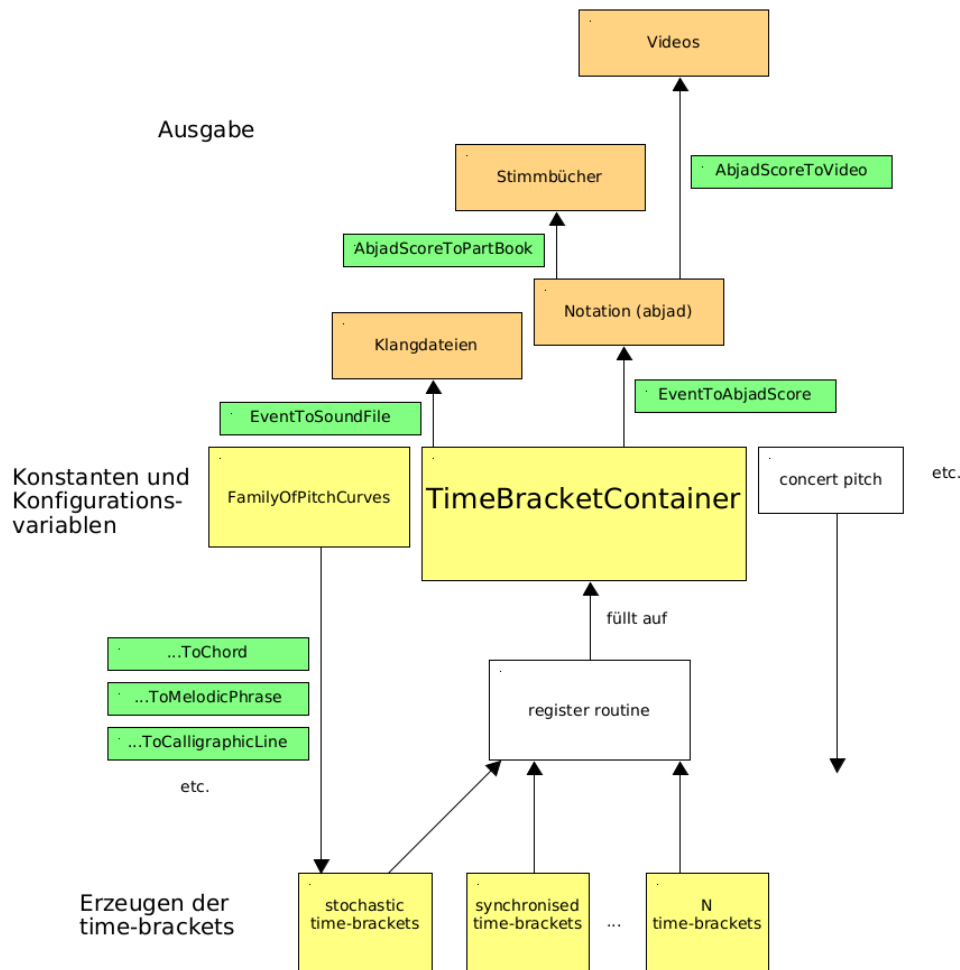


Abbildung 8: Schematischer Aufbau der Programmstruktur von *ohne Titel (2)* und *ohne Titel (3)*. Gelb markierte Elemente repräsentieren Ereignisse, grün markierte Elemente sind Übersetzer.

Trotz geteilter Strukturen, unterscheiden sich die Stücke *ohne Titel (2)* und *ohne Titel (3)* in ihrem klanglichen Erleben. Über gemeinsamen Untergrund sind kompositorisch–kontrastierende Entscheidungen gesetzt. Der gemeinsame Untergrund wird mehrheitlich über Implementierungen im generischen Code *mutwos* geteilt.

¹⁰Ein euklidischer Rhythmus ist die möglichst gleichmäßige Verteilung einer bestimmten Anzahl Anschläge auf eine bestimmte Anzahl Pulse [Toussaint 2005].

In *ohne Titel (2)* verläuft Zeit gerichtet. Die Form ist diskret geteilt. Einzelne Abschnitte alterieren zügig. Das Ende kann als irreversibles Ereignis gelesen werden, das kein Zurück zu einem Vorher (innerhalb der Form) erlaubt. Die Besetzung der fünf Instrumente ist offen. Die Klangdatei wird über vier möglichst neutrale, im Kreis aufgestellte Lautsprecher und drei, vor Spieler:innen aufgestellte Lautsprecher abgespielt. Ideale Aufführungssituation ist ein geschlossener, abgedunkelter Raum mit längerer Nachhallzeit.

In *ohne Titel (3)* verläuft Zeit ungerichtet. Die Form ist kontinuierlich. Alterierende Formabschnitte laufen in langsamen Bewegungen ineinander über. Die global konstante, pendelnde Harmonie¹¹ verhindert im Zeitverlauf Orientierung. Die Abwesenheit der Orientierung bedingt ein Gefühl der Zeitlosigkeit. Die Besetzung der beiden Instrumente ist spezifiziert. Die Klangdatei wird über mobile Kassettenrekorder (Radios) abgespielt, die ungeordnet aufgestellt werden. Ideale Aufführungssituation ist ein offener Raum in der Natur.

Gemeinsam ist der klanglichen Wirklichkeit beider Werke, dass sie von einer Angleichung von Elektronik und Instrumenten lebt. Auf Aufnahmen ist stellenweise schwierig zu erkennen, welches Klangereignis instrumentalen oder elektronischen Ursprungs ist. Ursache der Ähnlichkeit ist das Erzeugen beider Bestandteile mit verwandten, innereren Repräsentationen. Weil Objekte innerhalb *mutwos* bis zur Übersetzung nicht ihr Ausgangsformat kennen, können sie generisch verwendet werden. Klangdateien werden genauso wie instrumentale Stimmen als Sammlung von *time-brackets* repräsentiert. Übersetzer zum Erzeugen der Ereignisinstanzen entsprechen oder ähneln den Übersetzer für die instrumentale Strukturen. Entscheidungen expliziter Start- und Endzeiten trifft für Klangdateien ein (Pseudo-)Zufallsgenerator. Jedem Entscheider wird einen konstanten Startwert (*random seed*) zugewiesen.

Der Umgang mit Flageoletts und Mehrklänge in *ohne Titel (3)* ist exemplarisch für *mutwos* Ideal variabler Repräsentationen. Innerlich wird von resultierenden Tonhöhen ausgegangen. Diese werden den Ereignissen (die später zur Notation der Instrumente übersetzt werden) zugewiesen. Dritte Programmkomponente können mit den Ereignissen (in ihrem klanglichen Resultat) umgehen, z. B. können Simulationen der Instrumentalstimmen mit eine MIDI Übersetzer erzeugt werden oder andere Übersetzer leiten aus ihnen Stimmen für Klangdateien ab.

¹¹Variiert werden Frequenz und Ambitus der Pendelbewegung.

Bevor die Ereignisse dem Notationsübersetzer übergeben werden, wird ein anderer Übersetzer dazwischengeschaltet. Dieser weist den Ereignissen je nach Tonhöhenstruktur Flageolets oder Mehrklänge (mit Saxophongriffen) zu.

Ähnlich verhält es sich mit der Notation der Keyboardstimme in *ohne Titel (2)*. Die Keyboardstimme spielt mithilfe eines Softwaresynthesizers mikrotonale Tonhöhen. Innerlich wird von klingenden Tonhöhen ausgegangen. Aber im Stimmbuch des Keyboards werden die chromatischen Tonhöhen der zu spielenden Tasten notiert. Ein dazwischengeschalteter Übersetzer kommuniziert zwischen den unterschiedlichen Repräsentationen. Die innere, klingende Repräsentation entspricht der Vorstellung der Komposition. Sie ist auch Grundlage für die Notation der Stimmbücher der anderen Instrumente. In ihren synchronisierten Abschnitten werden die Tonhöhen des Keyboards klingend notiert. Weil die Übersetzungen automatisiert erfolgen, ist der übersetzte Inhalt autark von ihnen. In der kompositorischen Arbeit kann sich deswegen auf dynamisch entwickelnden Inhalt konzentriert werden.

Die Arbeiten *ohne Titel (2)* und *ohne Titel (3)* ergänzen das Fallbeispiel *thanatos trees for Tim Pauli* um ein weiteres Konzept, was in *mutwos* Modell (der Ereignisse und Übersetzer) eingebettet realisiert werden kann. Beide Fallbeispiele teilen keinen generischen Algorithmus zum Erzeugen der umfassenden musikalischen Struktur. Sie teilen aber kleine, praktische Funktionseinheiten, ein Modell, das die Umsetzung innerer Vorstellungen erlaubt und Überschneidungen der Arbeitsweise. Die Arbeitsweise überschneidet sich, weil in allen Fallbeispiele Programmcode geschrieben wird, der für die Programmausführung einen Einstiegspunkt definiert. Und der Einstiegspunkt generiert in allen Fallbeispiele die gesamte Komposition für bestimmte oder alle Ausgangsmédien. Es werden nicht einzelne musikalische Identitäten wie Rhythmen oder Harmonien erzeugt, die anschließend händisch-kompositorisch in eine Form gesetzt werden. Stattdessen wird eine vollständige Form in Textdateien encodiert und in andere Médien (z. B. Klangdateien) und Codierungen (z. B. Notation) übersetzt. Die Überschneidungen brechen nicht mit *mutwos Motivation und Absicht* eines interfaceagnostischen Designs, denn das *mutwo* Ökosystem könnte z. B. zum Erzeugen singulärer Bestandteile verwendet werden. Zugleich muss angemerkt werden, dass gegenwärtig *mutwo* keine komplexe Benutzerschnittstelle implementiert. Es gibt z. B. keine graphische Benutzeroberfläche, keine

reiche Unterstützung für Objektpermanenz (Abspeichern in Datenbanken), keine Musikauszeichnungssprache, keine integrierte Entwicklungsumgebung (IDE). Wie in Abschnitt *Limitierungen und Grenzen* beschrieben, sind nicht technische Kompatibilitätsschwierigkeiten oder Unzulänglichkeiten des Designs dafür verantwortlich, sondern die Abwesenheit einer Notwendigkeit gegenwärtiger Nutzer:innen.

3 Schlusswort

3.1 Zusammenfassung

Komposition verwendet unterschiedliche Techniken. Verwendet Komposition die Technik Algorithmen, wird sie als algorithmische Komposition bezeichnet. Algorithmische Komposition kann via einer Software praktiziert werden. *Mutwo* ist eine Softwarebibliothek für algorithmische Komposition. Eine Kongruenz zwischen verwendeten Techniken und inneren Vorstellungen ist Absicht der Entwicklung *mutwos*. *Mutwo* muss generisch sein, aber zugleich allgemeine, praktische Repräsentationen bereitstellen. *Mutwos* generische Design zielt auf langfristige Kongruenz bei sich verändernden Vorstellungen ab. Vordefinierte, allgemeine Repräsentationen machen *mutwo* praktikabel. *Mutwos* Modell zur Formalisierung von Kunst definiert die Elemente *Ereignisse*, *Parameter*, *Übersetzer* und *Generatoren*. *Ereignisse* sind Bewegungen, *Parameter* beschreiben Bewegungen, *Übersetzer* transformieren Entitäten, *Generatoren* erzeugen Daten. Konkrete Entwicklungsstrategien wie die Verwendung abstrakter Basisklassen realisieren *mutwos* unterliegende Absichten. *Mutwos* junges Lebensalter und die kleine Anzahl von Entwickler:innen limitieren die enthaltenen Funktionen und Konsistenz der Implementierung. In drei Fallbeispielen kann eine praxistaugliche Anwendung *mutwos* verifiziert und nachvollzogen werden.

3.2 Aussichten

Vorliegender Text dokumentiert *mutwo* primär auf einer abstrakten, avancierten Vorstellungsebene. Diese Dokumentation unterstützt jede Person, die die Software (weiter-)entwickeln möchte oder fortgeschritten mit ihr arbeitet. Um neuen Nutzer:innen den Einstieg zu erleichtern, sollte der abstrakte Text zukünftig um eine niedrigschwellige, Beispiel-orientierten

Einführung ergänzt werden. Die Einführung sollte sequentiell getrennt wichtige Themen praxisnah vorstellen. Letztlich wäre eine Übersetzung aller dokumentarischen Texte ins Englische erstrebenswert. Denn nur eine Internationalisierung kann die in *Limitierungen und Grenzen* beschriebene Diversität erzielen.

Wie viele Software befindet sich *mutwo* selbst in einem kontinuierlichem Prozess aus Instandhaltung, Aktualisierung, Verbesserung, Fehlerbehebung und Weiterentwicklung. Zukünftige Bemühungen sollten sich darauf fokussieren bestehenden Code und Codedokumentation zu säubern und zu aktualisieren. Während `mutwo.core` und überwiegende Teile des Pakets `mutwo.music` sauber implementiert, dokumentiert und getestet sind, befinden sich nicht alle Pakete in gesundem Zustand. Besonders gewöhnliche Bestandteile wie MIDI- oder Notationsübersetzer müssen hohen Qualitätsstandards genügen, um perspektivisch praktikabel zu sein.

Seit initialer Entwicklung *mutwos* verwende ich die Software in allen Werken meiner künstlerischen Arbeit. Im Vergehen der Zeit verschieben sich meine künstlerischen Interessen, verschiebt sich die konkrete Ausrichtung *mutwos*, verschiebt sich mein Umgang mit *mutwo*. Was konstant bleibt ist aber ein Verständnis für eine musikalische Praxis, die Programmieren inkludiert, die Inneres in formalisierte Textdateien festhält, die Protokolle und Technologien und Traditionen explizit und spät wählt, die in Formalisierung und Abstraktion eine Distanz von Persönlichkeit und Geschmack und Identität und Intention sucht.

Literatur

- [1] *About Abjad*. Zugriffsdatum: 19.09.2022. URL: <https://abjad.github.io/>.
- [2] *About OR-Tools*. Zugriffsdatum: 19.09.2022. URL: <https://developers.google.com/optimization/introduction/overview>.
- [3] *Abstract type*. Zugriffsdatum: 06.09.2022. URL: https://en.wikipedia.org/wiki/Abstract_type.
- [4] Khyam Allami. "Microtonality and the Struggle for Fretlessness in the Digital Age". In: *CTM Magazine* (2019), S. 55–61.
- [5] Clarence Barlow. "The Ratio Book". In: 1999. Kap. On the Quantification of Harmony and Metre.
- [6] Georg Brandl u. a. *General Python FAQ*. Zugriffsdatum: 11.09.2022. URL: <https://docs.python.org/3/faq/general.html>.
- [7] Andrew Brown u. a. *Quellcode von jMusic*. Zugriffsdatum: 21.08.2022. 2017. URL: <https://sourceforge.net/p/jmusic/src/ci/d01f85/tree/>.
- [8] *Constraint programming*. Zugriffsdatum: 12.09.2022. URL: https://en.wikipedia.org/wiki/Constraint_programming.
- [9] *Convention over configuration*. Zugriffsdatum: 06.09.2022. URL: https://en.wikipedia.org/wiki/Convention%5C_over%5C_configuration.
- [10] Thomas H. Cormen u. a. *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 1990.
- [11] Michael Edwards u. a. *Quellcode von slippery-chicken*. Zugriffsdatum: 05.09.2022. 2021. URL: <https://github.com/mdedwards/slippery-chicken/tree/0658eac>.
- [12] Marc Evanstein. *Quellcode von SCAMP*. Zugriffsdatum: 21.08.2022. 2021. URL: <https://git.sr.ht/~marcevanstein/scamp/tree/26438a7/item/scamp/score.py%5C#L2370>.
- [13] Andrew Gerzso. "Paradigms and Computer Music". In: *Leonardo Music Journal* (Juli 1992), S. 73–79.

- [14] Terumi Narushima. *Microtonality and the Tuning Systems of Erv Wilson*. New York: Routledge, 2018.
- [15] Gerhard Nierhaus. *Algorithmic Composition. Paradigms of Automated Music Generation*. Wien: Springer-Verlag, 2009.
- [16] Harry Partch. *Genesis of a Music*. New York: De Capo Press, 1949.
- [17] Tim Peters. *The Zen of Python*. Zugriffsdatum: 06.09.2022. 2004. URL: <https://peps.python.org/pep-0020/>.
- [18] Donya Quick. *Quellcode von Euterpea*. Zugriffsdatum: 21.08.2022. 2019. URL: <https://github.com/Euterpea/Euterpea2/blob/5949185/Euterpea/Music.lhs%5C#L10>.
- [19] Guido van Rossum und Ivan Levkivskyi. *PEP 483 - The Theory of Type Hints*. Zugriffsdatum: 04.09.2022. 2014. URL: <https://peps.python.org/pep-0483/>.
- [20] *The Python Language Reference*. Zugriffsdatum: 19.09.2022. URL: <https://docs.python.org/3/reference>.
- [21] Godfried Toussaint. "The Euclidean Algorithm Generates Traditional Musical Rhythms". In: *Proceedings of BRIDGES: Mathematical Connections in Art, Music, and Science* (2005), S. 47–56.
- [22] *typing - Support for type hints*. Zugriffsdatum: 04.09.2022. URL: <https://docs.python.org/3/library/typing.html>.
- [23] Benedict Weisser. "John Cage: "... The Whole Paper Would Potentially Be Sound": Time-Brackets and the Number Pieces (1981-92)". In: *Perspectives of New Music* (2013), S. 176–225.
- [24] *What is Free Software?* Zugriffsdatum: 04.09.2022. URL: <https://www.fsf.org/about/what-is-free-software>.
- [25] *X-SAMPA*. Zugriffsdatum: 14.08.2022. URL: <https://en.wikipedia.org/wiki/X-SAMPA>.