

# Diplomarbeit

*Verantwortlicher  
Professor*

**Prof. Jochen Ludewig**

*Inhaltlicher Betreuer  
für den Bereich der  
Informatik*

**Prof. Jochen Ludewig**

*Inhaltlicher Betreuer  
für den Bereich der  
Musik*

**Georg Wötzer**

*Technischer  
Betreuer*

**Ralf Melchisedech**

*DA-Nummer*

Diplomarbeit-Nr. 1305

*Titel*

**Graphische Darstellung  
von Noten**

*Diplomand*

**Christoph Weiß**

*Beginn am*

24.5.1995

*Beendet am*

23.11.1995

*CR-Klassifikation*

D.2.0, D.2.1, D.2.2, D.2.m

## Danksagungen

Ich möchte mich an dieser Stelle herzlichst bei allen bedanken, die mir beim Zustandekommen und Realisieren dieser Arbeit geholfen haben. Mein besonderer Dank geht dabei an die folgenden Personen:

Jochen Ludewig, der sich für dieses Thema begeistern ließ und diese Arbeit durch deren Ausschreibung erst möglich machte, obwohl er, organisatorisch gesehen, Berge versetzen mußte, um dafür eine gute Betreuung gewährleisten zu können.

Frank Wankmüller, der mir den letzten „Push“ dazu gegeben hat, zu versuchen, diese Arbeit ins Leben zu rufen.

Ralf Melchisedech, der wesentlich zu der hoffentlich guten Verständlichkeit dieses Berichts beitrug. Wenn irgend etwas unnötig schwer verständlich war oder eventuell falsch interpretiert werden konnte, so war spätestens Ralf derjenige, der es gefunden hat.

Georg Wötzer, der immer wieder für den nötigen Auftrieb sorgte und für musikalische Fragen jederzeit zur Verfügung stand.

Stephan Hauser, der mir bei technischen Problemen mit Rat und Tat zur Seite stand.

Wolfram Möllen, stellvertretend für alle, die dazu bereit waren, mir ihren Laser-Drucker zur Verfügung zu stellen, als mir mein Laser-Drucker am Schluß seinen Dienst versagte.

# Inhalt

<b>Kurzübersicht .....</b>	<b>4</b>
<b>Inhaltsverzeichnis .....</b>	<b>5</b>
<b>Verzeichnis der Abbildungen.....</b>	<b>9</b>

## Kurzübersicht

<i>Kapitel 1</i>	Überblick.....	10
<i>Kapitel 2</i>	Einführung in die Thematik .....	12
<i>Kapitel 3</i>	Einsatz von MUNOS.....	15
<i>Kapitel 4</i>	Funktionalität von MUNOS.....	21
<i>Kapitel 5</i>	Ein abstraktes Modell für objektorientierte Methoden.....	31
<i>Kapitel 6</i>	Die Entwicklungsmethode.....	40
<i>Kapitel 7</i>	Das Use-Case-Konzept an einem Beispiel aus der Entwicklung von MUNOS .....	56
<i>Kapitel 8</i>	Entwurfsmuster in MUNOS.....	75
<i>Kapitel 9</i>	Vergleich mit anderen Lösungen.....	82
<i>Kapitel 10</i>	Kritischer Rückblick und Ausblick .....	88
<i>Anhang A</i>	Glossar .....	91
<i>Anhang B</i>	Literatur.....	108
<i>Anhang C</i>	Erklärung zum selbständigen Arbeiten.....	110

## Inhaltsverzeichnis

<b>Kapitel 1</b>	<b>Überblick.....</b>	<b>10</b>
	1.1 Über die Aufgabenstellung.....	10
	1.2 Über diesen Bericht .....	10
<b>Kapitel 2</b>	<b>Einführung in die Thematik .....</b>	<b>12</b>
	2.1 Hintergrund.....	12
	2.2 Warum MUNOS ? .....	13
	2.3 Was ist MUNOS ? .....	13
<b>Kapitel 3</b>	<b>Einsatz von MUNOS.....</b>	<b>15</b>
	3.1 Das Model-View-Controller-Konzept.....	15
	3.2 Einsatzmöglichkeiten von MUNOS.....	17
<b>Kapitel 4</b>	<b>Funktionalität von MUNOS.....</b>	<b>21</b>
	4.1 Aufbau einer Programm-View von MUNOS.....	21
	4.2 Funktionen für ein Notations-Modell.....	24
	4.2.1 Die Repräsentation eines Tones erstellen, ändern oder löschen.....	24
	4.2.2 Die Repräsentation einer Pause erstellen, ändern oder löschen.....	25
	4.2.3 Ein Liniensystems erzeugen, ändern oder löschen .....	26
	4.2.4 Eine Stimme erzeugen oder löschen.....	27
	4.2.5 Eine Bezifferung erzeugen, ändern oder löschen .....	28
	4.3 Funktionen für eine Notations-View.....	28
	4.3.1 Ein Layout erstellen, ändern oder löschen.....	28
	4.3.2 Ein Layout einfrieren.....	29
	4.4 Funktionen für einen Notations-Controller .....	30
	4.4.1 Einen Layout-Manager erzeugen oder löschen.....	30

<b>Kapitel 5</b>	<b>Ein abstraktes Modell für objektorientierte Methoden.....</b>	<b>31</b>
	5.1 Der eigentliche Zweck des Modells.....	31
	5.2 Die Idee hinter der Methode des MUNOS-Projekts.....	32
	5.3 Das abstrakte Modell.....	33
	5.3.1 Phasenübergreifende Konzepte.....	34
	5.3.2 Phasen einer objektorientierten Entwicklung.....	35
	5.3.3 Lebenszyklus-Modelle.....	36
<b>Kapitel 6</b>	<b>Die Entwicklungsmethode.....</b>	<b>40</b>
	6.1 Warum Phasen ?.....	40
	6.2 Phasenübergreifende Konzepte.....	41
	6.2.1 Das Use-Case-Konzept.....	41
	6.2.2 Gewinnung und Einsatz von Mustern.....	44
	6.2.3 Die Notation.....	45
	6.3 Tätigkeiten und Ergebnisse der einzelnen Phasen.....	51
	6.3.1 Strategisches Modellieren.....	51
	6.3.2 Analytisches Modellieren.....	51
	6.3.3 Entwurfsbezogenes Modellieren.....	52
	6.3.4 Implementierungsbezogenes Modellieren.....	52
	6.3.5 Implementierung.....	52
	6.3.6 Vertrieb.....	53
	6.4 Das Lebenszyklus-Modell.....	53
	6.4.1 Die Progressionsstrategien.....	53
	6.4.2 Die Iterationsstrategie.....	54
	6.4.3 Die Verpackungsstrategie.....	55
	6.4.4 Die Meilensteinsstrategien.....	55

<b>Kapitel 7</b>	<b>Das Use-Case-Konzept an einem Beispiel aus der Entwicklung von MUNOS .....</b>	<b>56</b>
	7.1 Der Use-Case zur Erzeugung eines Liniensystems in der Spezifikation.....	58
	7.1.1 Aufbau der Beschreibung eines Use-Case.....	58
	7.1.2 Die Beschreibung des Use-Case zur Erzeugung eines Liniensystems.....	60
	7.2 Der Use-Case zur Erzeugung eines Liniensystems im Entwurf.....	65
	7.2.1 Das statische Diagramm.....	65
	7.2.2 Die Interaktionsdiagramme .....	66
	7.2.3 Aufbau der Beschreibung eines Objekttyps.....	68
	7.2.4 Die Beschreibungen der Objekttypen für Liniensysteme .....	69
	7.2.5 Aufbau der Beschreibung einer Operation.....	72
	7.2.6 Die Beschreibung einer Operation des abstrakten Objekttyps für Liniensysteme.....	74
<b>Kapitel 8</b>	<b>Entwurfsmuster in MUNOS.....</b>	<b>75</b>
	8.1 Ein Beispiel für ein wiederverwendetes Entwurfsmuster.....	75
	8.2 Ein Beispiel für ein gewonnenes Entwurfsmuster.....	78
<b>Kapitel 9</b>	<b>Vergleich mit anderen Lösungen .....</b>	<b>82</b>
	9.1 Die vermeintliche Lösung.....	82
	9.2 Die MIDI-Treiber Lösung.....	83
	9.3 Die MIDI-Standardfile Lösung .....	84
	9.4 Die fragwürdige Lösung.....	86
	9.5 Der letzte Ausweg .....	86
	9.6 Zusammenfassung.....	87

<b>Kapitel 10</b>	<b>Kritischer Rückblick und Ausblick .....</b>	<b>88</b>
	10.1 Kritischer Rückblick auf den Verlauf der Arbeit .....	88
	10.2 Ausblick.....	89
<b>Anhang A</b>	<b>Glossar .....</b>	<b>91</b>
<b>Anhang B</b>	<b>Literatur.....</b>	<b>108</b>
<b>Anhang C</b>	<b>Erklärung zum selbständigen Arbeiten .....</b>	<b>110</b>

## Verzeichnis der Abbildungen

<i>Abbildung 3.1</i>	Ein Programm, das ein Modell und verschiedenen Arten von Views darauf verwendet.....	17
<i>Abbildung 3.2</i>	Ein Programm, das verschiedene Modelle und jeweils dieselbe Art von View darauf verwendet.....	19
<i>Abbildung 4.1</i>	Aufbau einer Programm-View von MUNOS.....	21
<i>Abbildung 5.1</i>	Abstraktes Modell für objektorientierte Methoden .....	34
<i>Abbildung 6.1</i>	Objektyp.....	46
<i>Abbildung 6.2</i>	abstrakter Objektyp.....	46
<i>Abbildung 6.3</i>	Eine Monarchie ist eine Staatsform.....	46
<i>Abbildung 6.4</i>	Eine Monarchie enthält genau einen Monarchen .....	47
<i>Abbildung 6.5</i>	Ein Monarch verwendet beliebig viele Berater .....	47
<i>Abbildung 6.6</i>	Ein Staatssystem, das aus einem Monarchen und dem gemeinen Volk besteht .....	48
<i>Abbildung 6.7</i>	Interaktionsdiagramm eines Abhängigkeitsmechanismus .....	49
<i>Abbildung 8.1</i>	Eine MUNOS realisierte Variante des Entwurfsmusters <i>Observer</i> .....	76
<i>Abbildung 8.2</i>	Die beiden Realisierungen des Entwurfsmusters <i>Server</i> in MUNOS.....	79
<i>Abbildung 9.1</i>	Ein direkter Datenaustausch zwischen dem eigenen Programm und einem Sequenzerprogramm.....	82
<i>Abbildung 9.2</i>	Ein Austausch von Daten im MIDI-Format über einen MIDI-Treiber zwischen dem eigenen Programm und einem Sequenzerprogramm .....	83
<i>Abbildung 9.3</i>	Ein Austausch von Daten im MIDI-Format über ein MIDI-Standardfile zwischen dem eigenen Programm und einem Sequenzerprogramm.....	85

# 1 Überblick

## 1.1 Über die Aufgabenstellung

Bei dieser Diplomarbeit geht es um die Entwicklung einer Bibliothek, die Programmen die graphische Darstellung von Noten am Bildschirm ermöglichen soll. Die Betonung der Arbeit liegt bei der Spezifikation und dem Entwurf der Bibliothek. Zur Abrundung des Ganzen soll schließlich noch eine Teil-Implementierung erfolgen.

Die Bibliothek wird unter dem Namen **MUNOS** entwickelt, der für „**M**usic **N**otation on **S**creen“ steht.

## 1.2 Über diesen Bericht

Nach diesem Überblick wird in Kapitel 2, „Einführung in die Thematik“, der Hintergrund geschildert, unter dem diese Arbeit zustande kam. Dann wird ersichtlich gemacht, warum ein Bedarf an dieser Arbeit bestand, um was es eigentlich genau geht und für wen diese Arbeit von Interesse ist.

In Kapitel 3, „Einsatz von MUNOS“, wird die Frage beantwortet, wie MUNOS eingesetzt werden kann, d.h. welche Rolle MUNOS im Kontext eines gesamten Systems zukommt. In diesem Zusammenhang wird auch auf das **Model-View-Controller-Konzept** eingegangen.

Nachdem die Rolle von MUNOS innerhalb eines Systems geklärt ist, befaßt sich Kapitel 4, „Funktionalität von MUNOS“, damit, welche Funktionalität MUNOS zur Verfügung stellt, um dieser Rolle gerecht zu werden. Die Funktionalität von MUNOS spielt sich auf einer bestimmten Ebene von MUNOS ab. Daher wird in diesem Zusammenhang auch auf den Aufbau dieser Ebene eingegangen.

In Kapitel 5, „Ein abstraktes Modell für objektorientierte Methoden“, wird das abstrakte Modell für objektorientierte Methoden der **Object Management Group** vorgestellt. Dabei wird auch darauf eingegangen, wofür das abstrakte Modell eigentlich gedacht ist und wie daraus die Methode zustande kam, die in für die Entwicklung von MUNOS verwendet wird.

In Kapitel 6, „Die Entwicklungsmethode“, wird die Methode, die für die Entwicklung von MUNOS verwendet wird, anhand des abstrakten Modells aus Kapitel 5 charakterisiert. Vor der eigentlichen Charakterisierung wird jedoch die Frage diskutiert, ob die Auftrennung einer Entwicklung in einzelne Phasen überhaupt sinnvoll ist. In der Charakterisierung wird dann unter anderem das **Use-Case-Konzept** vorgestellt, das bei dieser Methode eine zentrale Rolle spielt, denn durch dessen Anwendung kann einer Entwicklung ein „roter Faden“ gegeben werden. Außerdem wird in der Charakterisierung auf aktuelles Forschungsgebiet eingegangen, nämlich die Gewinnung und den Einsatz von **Mustern**, wie beispielsweise Entwurfsmustern.

In Kapitel 7, „Das Use-Case-Konzept an einem Beispiel aus der Entwicklung von MUNOS“, wird anhand eines Beispiels dargelegt, wie mit Hilfe von Use-Cases spezifiziert werden kann und wie diese Art der Spezifikation dann in einen Entwurf umgesetzt wird.

In Kapitel 8, „Entwurfsmuster in MUNOS“, wird jeweils anhand eines Beispiels gezeigt, wie bei der Entwicklung von MUNOS ein bereits bewährtes Entwurfsmuster eingesetzt wurde und wie ein neues Entwurfsmuster gewonnen wurde.

In Kapitel 9, „Vergleich mit anderen Lösungen“, werden andere Lösungen für die graphische Darstellung von Noten vorgestellt und es wird beschrieben, wo diese Lösungen im Vergleich zu MUNOS anzusiedeln sind. Dabei wird der Bedarf an dieser Arbeit noch einmal deutlich gemacht.

In Kapitel 10, „Kritischer Rückblick und Ausblick“, wird ein kritischer Rückblick auf den Verlauf der Arbeit vorgenommen und ein Ausblick darauf gegeben, was für die Zukunft noch geplant ist. In dem kritischen Rückblick wird auch auf ein generelles Problem beim Umgang mit Use-Cases hingewiesen.

Der Bericht schließt mit den Anhängen A bis C. Im Anhang A werden in einem Glossar Begriffe aus unterschiedlichen Fachgebieten erklärt, im Anhang B folgt das Literaturverzeichnis und im Anhang C findet sich die Erklärung zum selbständigen Arbeiten.

## 2 Einführung in die Thematik

### 2.1 Hintergrund

An der Staatlichen Hochschule für Musik und Darstellende Kunst in Stuttgart läuft seit einiger Zeit ein Projekt mit dem Ziel, ein Programm zu entwickeln, das in der Lage ist, einen Generalbaß auszusetzen. Der **Generalbaß** war im Barockzeitalter (ca. 1600-1740) ein Wesensmerkmal der Musik. Er diente hauptsächlich als harmonisches Grundgerüst für Improvisationen, anhand dessen auch musiktheoretische Kenntnisse vermittelt wurden. Deshalb wird in der Musik synonym zum Barockzeitalter auch vom „Generalbaßzeitalter“ gesprochen, in dem Johann Sebastian Bach (1685-1750) eine der prägensten Persönlichkeiten war.

Ein Generalbaß besteht aus einer Folge von Noten, die von einem oder mehreren Baßinstrumenten gespielt werden. Wenn zu einem Generalbaß eine Begleitung geschrieben wird, so nennt man das „den Generalbaß aussetzen“. Dabei müssen zu jedem Ton eines Generalbasses üblicherweise drei Töne als Begleitung gefunden werden, um insgesamt einen vierstimmigen Satz zu erhalten. Eine derartige Begleitung wurde zu Zeiten Bachs meist nicht niedergeschrieben, sondern „live“ improvisiert. Die Musiker verwendeten dann einen Generalbaß als Orientierung und erfanden während des Spielens die Begleitung und Melodien dazu, die zur Verzierung der Begleitung dienten. Es gab damals eine feste Regel dafür, wie einzelne Baßnoten innerhalb einer Tonart zu verstehen waren und welche Töne für eine Begleitung in Frage kamen. Bei Ausnahmen zu dieser Regel wurde die entsprechende Baßnote mit einer sogenannten „Bezifferung“ versehen, die darüber Auskunft gab, wie diese Baßnote in diesem Fall zu verstehen war, d.h. in welchem harmonischen Kontext diese an der Stelle zu sehen war.

Einen Generalbaß auszusetzen, ist zwar einerseits nicht trivial, aber andererseits doch bis zu einem gewissen Grade reines Handwerk. Deshalb sollte das auch für einen Rechner machbar sein. Wenn nun der Rechner einen Generalbaß aussetzt, dann wäre es sicherlich wünschenswert, das Ergebnis in Form der üblichen graphischen Notendarstellung zu bekommen, anstatt nur die Namen der einzelnen Töne am Bildschirm ablesen zu müssen. An diesem Punkt soll MUNOS weiterhelfen.

## 2.2 Warum MUNOS ?

Häufig wird in Programmen auf die übliche Notendarstellung verzichtet, weil der Aufwand, diese selbst zu schreiben, gescheut wird. Auch scheinen keinerlei Hilfsmittel zu existieren, die einem Softwareentwickler diese Arbeit abnehmen oder wenigstens erleichtern. Zumindest blieben alle Nachforschungen in diese Richtung bislang erfolglos. Auch die Fachliteratur bezieht sich lediglich auf das Endprodukt, d.h. den Notentext selbst und wie dieser auszusehen hat (vgl. Roemer, 1973). Die Zielgruppe dafür sind dann auch eher Kopisten, die Notentexte von Hand verfassen, und weniger Softwareentwickler.

Die Gründe dieser Misere sind vielschichtig. Einerseits gibt es nur einen relativ kleinen Anteil an Entwicklern, die hauptberuflich für den Musikbereich entwickeln. Andererseits erfordert Software, in der durch ein bestimmtes Regelwerk Töne erzeugt werden, unter Umständen viel musiktheoretisches Wissen. Daher sind es eher hauptberufliche Musiker, vor allem Komponisten, die nebenher in diesem Bereich Software entwickeln. Diese sind dann primär an den musiktheoretischen Problemen ihrer Programme interessiert und verfügen in der Regel auch nicht über das Fachwissen eines Informatikers. Eine Zusammenarbeit aus beiden Fachgebieten, wie bei dem Generalbaßprojekt (vgl. 2.1, S. 12) oder bei dieser Diplomarbeit, stellt leider noch immer die Ausnahme dar.

Mit MUNOS soll nun ein Hilfsmittel zur Verfügung gestellt werden, das einem Softwareentwickler weitgehendst die Arbeit abnimmt, in seinen Programmen Noten graphisch darzustellen. Auf diese Weise soll eine Art Infrastruktur im Bereich der Musiksoftware geschaffen werden.

## 2.3 Was ist MUNOS ?

MUNOS ist eine Bibliothek, die Programmen eine graphische Darstellung von Noten innerhalb deren eigener Programmoberflächen ermöglicht. MUNOS ist ganz bewußt kein Notensatzprogramm, um Notentexte für Publikationen zu erstellen, sondern eine Klassenbibliothek, die auf mögliche Bedürfnisse von Softwareentwicklern und deren Programmen zugeschnitten ist. MUNOS ist daher eigentlich für jeden Entwickler interessant, der in seinem Programm eine Notendarstellung benötigt.

MUNOS kann einerseits von Musikprogrammen wie dem Generalbaßprogramm aus 2.1 (vgl. S. 11) verwendet werden, denkbar wäre aber auch der Einsatz von MUNOS in einem Textverarbeitungsprogramm, in dem die Darstellung von Noten-beispielen möglich sein soll. Ähnlich wie auch Formeln mit Hilfe eines Formeleditors erstellt werden, würden Notenbeispiele mit Hilfe eines kleinen Noteneditors erstellt, der dann auf MUNOS basieren würde. Dieses Beispiel geht zwar wieder in Richtung Notensatzprogramm, aber das ist, wie bereits erwähnt wurde, nicht das eigentliche Ziel. Welche Rolle MUNOS generell innerhalb eines Programms zukommt, wird im folgenden Kapitel beschrieben.

## 3 Einsatz von MUNOS

Am sinnvollsten wird MUNOS in einem Programm eingesetzt, das nach dem Model-View-Controller-Konzept aufgebaut ist. Daher wird in diesem Kapitel zunächst auf das Model-View-Controller-Konzept selbst eingegangen. Zwar kann MUNOS ohne weiteres auch innerhalb eines anderen Programmkontexts verwendet werden, aber die Grundintention liegt bei einem Einsatz innerhalb eines Model-View-Controller-Kontexts. Anschließend werden zwei Beispiele für den Einsatz von MUNOS gegeben, wobei verschiedene Aspekte des Einsatzes beleuchtet werden.

### 3.1 Das Model-View-Controller-Konzept

Das Ziel des Model-View-Controller-Konzept ist es, die Unabhängigkeit der Daten eines Programms von dessen Benutzeroberfläche zu erreichen. Dazu wird innerhalb eines Programms eine Aufgabenteilung in ein Modell, eine oder mehrere Views und einen oder mehrere Controller vorgenommen. Durch diese Aufgabenteilung werden die einzelnen Programmteile mehr entkoppelt. Das bringt unter anderem den Vorteil mit sich, flexibler auf Änderungen reagieren zu können. Und wenn sich Änderungen der Benutzeroberfläche nie auf das Modell auswirken, bietet das den zusätzlichen Vorteil, ein Programm leichter auf andere Plattformen portieren zu können. Im folgenden wird darauf eingegangen, was in diesem Kontext unter einem Modell zu verstehen ist, was Views bzw. Controller sind und wie dieses „Dreigestirn“ zusammenarbeitet.

Das **Modell** enthält die Daten eines Programms, die von diesem bearbeitet werden, d.h. ein Modell kann ein beliebiges Datenmodell sein. Im Falle eines Musikprogramms könnte ein Modell beispielsweise alle Informationen zu einem Musikstück enthalten. Und das wiederum könnten dann Informationen über Töne, Pausen, Instrumente, Taktarten, Tonarten, etc. sein.

Eine **View** erzeugt eine Darstellung eines Modells, bei der das Modell auf eine ganz bestimmte Art und Weise interpretiert wird. Die Interpretation äußert sich in der Auswahl dessen, was von dem Modell wie dargestellt wird.

Denkbare Views für Töne wären beispielsweise deren tabellarische Darstellung in Form von MIDI-Daten, deren graphische Darstellung in Form von Wellen oder deren graphische Darstellung in Form von Noten. Bei jeder dieser Repräsentationen spielen andere Aspekte der Töne eine Rolle. Verschiedene Views auf ein Modell ermöglichen so, ein Modell aus verschiedenen Blickwinkeln heraus zu betrachten. Oft wird durch eine View nur ein bestimmter Teil des Modells dargestellt, der einem ganz bestimmten Gesichtspunkt entspricht. Zum Beispiel würde für einen Pianisten in erster Linie nur der Teil interessant sein, den er auf dem Klavier spielen soll. Ihm würde deshalb ein Klavierauszug von dem Modell eines Musikstücks genügen.

Ein **Controller** hat die Aufgabe, Benutzereingaben, die das Modell betreffen, in Aufrufe an das Modell umzusetzen. Wenn die Benutzereingaben nicht das Modell, sondern eine oder mehrere Views davon betreffen, so setzt der Controller die Benutzereingaben direkt in Aufrufe an die betroffenen Views um. Wenn ein Benutzer beispielsweise nur den Abstand zweier Noten ändert, ohne dabei gleichzeitig deren Taktpositionen zu ändern, so läßt diese Änderung die Töne im Modell unberührt und wird direkt an die betroffene Noten-View weitergeleitet.

Eine Änderung am Modell hat immer die Änderung der Views zur Folge, die den Aspekt des Modells zeigen, der sich geändert hat. Wenn beispielsweise die Tonart geändert wird, die absolute Tonhöhe der einzelnen Töne aber gleich bleibt, so betrifft diese Änderung das Modell und wird an dieses weitergeleitet. In der MIDI-View und in der Noten-View wird jeweils auch die Tonart dargestellt. Deshalb sind diese beiden Views von der Änderung am Modell betroffen und werden vom Modell über die Art der Änderung informiert. Die betroffenen Views aktualisieren sich anschließend selbst, indem sie vom Modell die neue Tonart erfragen und ihre Darstellungen dementsprechend abändern. Da die absolute Tonhöhe der einzelnen Töne in diesem Fall unverändert blieb und in der Wellen-View keine Tonarten dargestellt werden, ist die Wellen-View von der Änderung am Modell nicht betroffen und bleibt unverändert.

## 3.2 Einsatzmöglichkeiten von MUNOS

Aus dem Kontext des Model-View-Controller-Konzepts heraus gesehen, bietet MUNOS einem Programm die Möglichkeit, eine View auf ein Modell des Programms zu erstellen und zu kontrollieren. Da es sich bei MUNOS um eine Bibliothek handelt, ist in diesem Kontext mit einem Programm immer ein Programm gemeint, von dem MUNOS verwendet werden könnte. Es folgen nun zwei Beispiele für generelle Einsatzmöglichkeiten von MUNOS, in denen zur Vereinfachung des Sachverhalts jeweils das Modell eines Musikstücks auf ein Ton-Modell reduziert wurde.

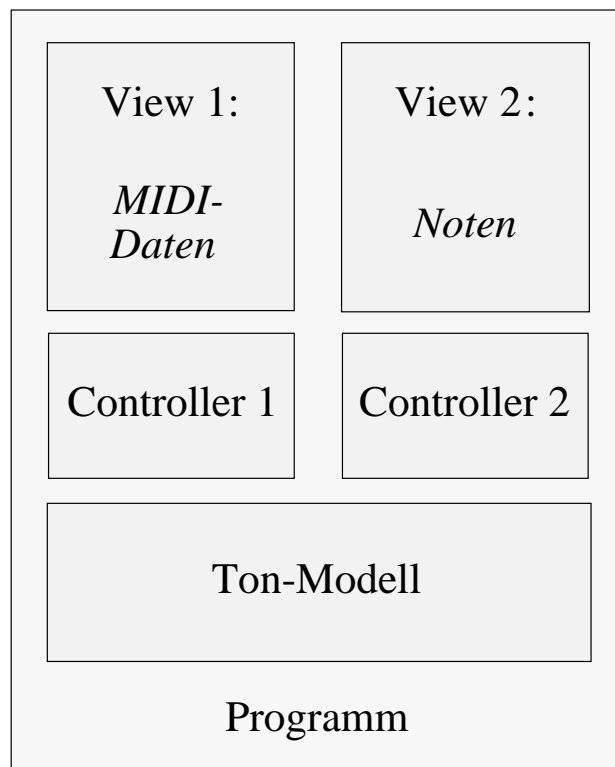


Abb. 3.1: Ein Programm, das ein Modell und verschiedene Arten von Views darauf verwendet

Die Abbildung 3.1 zeigt ein Programm, das ein Ton-Modell und zwei unterschiedliche Arten von Views darauf besitzt. In View 1 wird das Ton-Modell in Form von MIDI-Daten repräsentiert. In View 2 wird das Ton-Modell in Form von Noten repräsentiert. Die View 2 könnte daher von MUNOS stammen.

Um mit MUNOS eine Noten-View auf ein Ton-Modell erzeugen zu können, muß ein Programm in dem Ton-Modell für jeden Ton Informationen über dessen Tonhöhe, dessen Einsatzpunkt und dessen Dauer festhalten. Darüber hinaus könnte ein Programm in dem Ton-Modell für jeden Ton auch noch weitere Informationen festhalten, wenn diese für die Darstellung des Modells durch andere Arten von Views relevant sind. In dem Beispiel aus der Abbildung 3.1 wären das weitere Daten, die für eine MIDI-View benötigt werden. Das soll jetzt aber nicht über die eigentliche Denkweise hinwegtäuschen, denn normalerweise steht am Anfang das Modell, in dem man bestimmte Informationen festhalten möchte. Dann überlegt man sich, wie diese Informationen am sinnvollsten dargestellt werden sollen und kommt so zu seinen Views.

Um dem Controller eines Programms die Kontrolle über Benutzereingaben zu ermöglichen, hält MUNOS in jedem Graphikobjekt des Notenbilds einer Noten-View die semantischen Informationen fest, was durch das Graphikobjekt dargestellt wird. In dem Beispiel aus der Abbildung 3.1 würde MUNOS in jedem Graphikobjekt, das eine Note darstellt, festhalten, welchen Ton und welche Note dieses Graphikobjekt darstellt. Beides muß nicht identisch sein, denn ein einzelner Ton kann auch in Form mehrerer gebundener Noten geschrieben werden (vgl. 4.1, S. 22). MUNOS hält in jedem Graphikobjekt außerdem das Vorgabeobjekt für das Graphikobjekt fest, denn jedes Graphikobjekt in MUNOS besitzt ein Vorgabeobjekt, das jeweils die Daten enthält, die für die Erzeugung und Darstellung des Graphikobjekts maßgebend sind.

Wenn dann von einem Benutzer des Programms in Abbildung 3.1 mit Hilfe der Maus eine Note bzw. eigentlich das Graphikobjekt, das diese Note darstellt, angeklickt würde, könnte der Controller des Programms bei dem Graphikobjekt nachfragen, was durch das Graphikobjekt dargestellt wird und erhielte von dem Graphikobjekt als Antwort, den Ton und die Note, um die es geht. Der Controller des Programms könnte von dem Graphikobjekt außerdem das Vorgabeobjekt für das Graphikobjekt erfragen. So wäre der Controller des Programms in der Lage, sich jeweils die Informationen zu beschaffen, die er benötigt, um auf eine Benutzereingabe entsprechend reagieren zu können.

Aber mit Hilfe von MUNOS kann ein Programm nicht nur eine View, sondern mehrere Views gleichzeitig erstellen und diese auch parallel verwalten. Das ist für Programme interessant, in denen mit mehreren verschiedenen Ton-Modellen gleichzeitig gearbeitet wird, wie es in dem nun folgenden im zweiten Beispiel für den Einsatz von MUNOS der Fall ist.

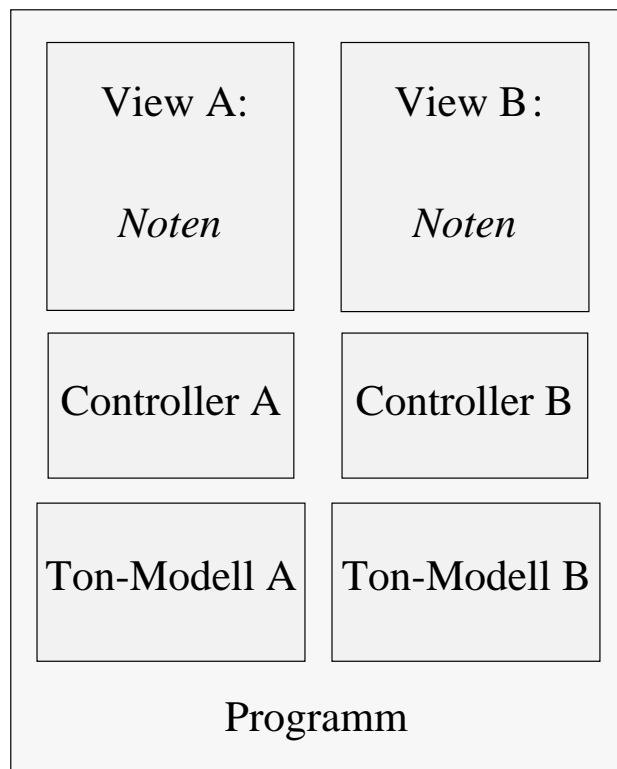


Abb. 3.2: Ein Programm, das verschiedene Modelle und jeweils dieselbe Art von View darauf verwendet

Die Abbildung 3.2 zeigt ein Programm, in dem zwei verschiedene Ton-Modelle A und B verwendet werden. Jedes dieser Tonmodelle könnte z.B. ein anderes Musikstück verkörpern. Für jedes der beiden Ton-Modelle existiert eine eigene View mit jeweils einem Controller für diese View. Das Ton-Modell A wird durch View A dargestellt und das Ton-Modell B durch View B. Beide Views zeigen im Prinzip dasselbe, nämlich Noten. Es handelt sich in diesem Beispiel also um dieselbe Art von Views, die sich nur durch ihr jeweiliges Bezugsmodell unterscheiden.

Es wäre natürlich auch denkbar, mehrere Views der gleichen Art vom gleichen Modell zu erzeugen. Das wird aber nicht notwendig, denn MUNOS bietet die Möglichkeit, von der Noten-Repräsentation eines Ton-Modells innerhalb von MUNOS verschiedene Views zu erzeugen. So wird dieser Fall auf die MUNOS-Ebene verlagert. Wie eine Programm-View von MUNOS aufgebaut ist und wie diese erzeugt werden kann, ist Thema des nächsten Kapitels.

## 4 Funktionalität von MUNOS

Die gesamte Funktionalität von MUNOS spielt sich auf der Programm-View-Ebene ab. Daher wird zunächst der Aufbau einer Programm-View von MUNOS beschrieben. Anschließend wird auf die Funktionalität eingegangen, die in MUNOS einem Programm zur Verfügung gestellt wird, um eine solche View zu erstellen.

### 4.1 Aufbau einer Programm-View von MUNOS

*Um die Ebenen klar zu trennen, wird entweder von einer „Programm-View von MUNOS“ oder von einer „Notations-View“ gesprochen*

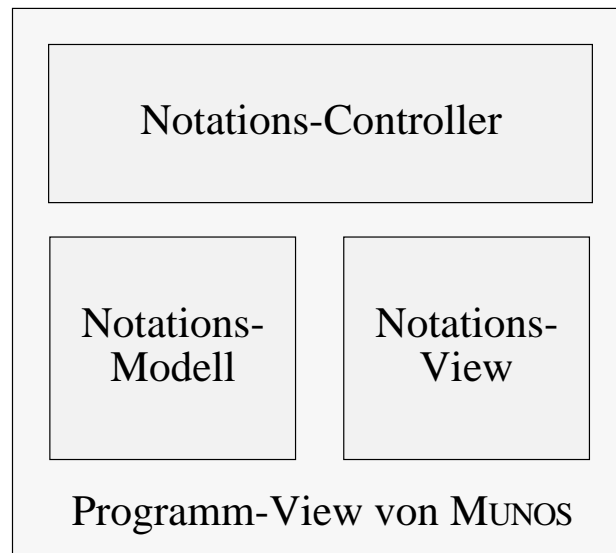


Abb. 4.1: Aufbau einer Programm-View von MUNOS

Wie Abbildung 4.1 zeigt, ist jede Programm-View von MUNOS auch wieder nach dem Model-View-Controller-Konzept aufgebaut (vgl. 3.1, S. 15ff). Die Frage nach dem Sinn dieses Aufbaus einer Programm-View lässt sich auf folgende Kernfrage reduzieren: Wozu braucht man in einer Programm-View ein Modell des Modells des Programms? In erster Linie braucht man das Modell der Programm-View, um dort Informationen explizit festzuhalten, die in dem Modell des Programms nur implizit enthalten sind. Beispielsweise entsteht aus dem in 3.2 geschilderten Ton-Modell eines Programms in der Programm-View von MUNOS ein Noten-Modell.

Ein Ton, der über mehrere Takte hinweg andauert, kann aber nicht mehr durch eine einzige Note dargestellt werden. In diesem Fall wird es nötig, diesen Ton in Form mehrerer Noten zu schreiben, die jeweils über eine Taktgrenze hinweg übergebunden werden. Dies ist nur einer von vielen Fällen, in dem ein Ton durch mehrere Noten repräsentiert werden muß. Deshalb ist das Ton-Modell des Programms nicht unbedingt 1:1 in Noten darstellbar.

Die Information, wieviel Noten jeweils zur Darstellung eines Tons benötigt werden, steckt implizit im Modell des Programms und wird aus den Informationen über die Dauer des Tons und über die dort gültige Taktart gewonnen. Das macht ein dazugehöriges Noten-Modell in MUNOS sinnvoll, in dem ein Ton explizit durch eine oder mehrere Noten repräsentiert wird. Um eine höhere Flexibilität zu erreichen macht es außerdem Sinn, die Graphikobjekte, die zur Darstellung der Noten verwendet werden, völlig unabhängig vom Noten-Modell zu betrachten. Auf diese Weise wird es möglich, diese beliebig austauschbar und frei definierbar zu machen, womit es einem Entwickler offensteht, die graphische Darstellung seinen Bedürfnissen entsprechend anzupassen und z.B. eine nichttraditionelle Schreibweise zu verwirklichen. Mit der Idee der generellen Abkopplung der graphischen Darstellung vom Modell, hat man im Prinzip auch schon eine View kreiert und ist mitten in der Verwirklichung des Model-View-Controller-Konzepts. Dabei herausgekommen sind ein Notations-Modell, eine oder mehrere Notations-Views und ein Notations-Controller für jede Programm-View von MUNOS. Um die Views der beiden Ebenen besser auseinanderhalten zu können, wird im folgenden entweder von der „Programm-View von MUNOS“ oder von der „Notations-View“ gesprochen.

Zu einem Notations-Modell gehören in jedem Fall ein oder mehrere Liniensysteme, denn jede Notations-View basiert letztendlich immer auf Liniensystemen. Ein Liniensystem enthält Informationen über Notenschlüssel, Taktarten und Tonarten. Es kennt außerdem die Noten und Pausenzeichen, die in ihm dargestellt werden. Neben den Liniensystemen können zu einem Notations-Modell noch Repräsentationen von Tönen und Pausen, sowie Stimmen und Bezifferungen gehören. Dabei werden Töne jeweils in Form von einer oder mehreren Noten und Pausen jeweils in Form von einem oder mehreren Pausenzeichen repräsentiert.

Eine Notations-View wird als „Layout“ bezeichnet. Ein Layout enthält die für die gewünschte Darstellung notwendigen Graphikobjekte, sowie Vorgabeobjekte, in denen Layout-spezifische Vorgaben für die Graphikobjekte festgehalten werden. Wenn ein Programm ein Layout erzeugt, so erzeugt dieses wiederum die gewünschte Darstellung in Form eines Bild-Objekts. Ein Bild-Objekt ist ein Graphikobjekt, das andere Graphikobjekte enthält. Das Bildobjekt eines Layouts kann dann bei Bedarf vom Programm auf beliebigen Ausgabegeräten dargestellt werden. Von einem Notations-Modell können beliebig viele Layouts erstellt werden. Durch ein Layout kann das gesamte Notations-Modell oder ein beliebiger Ausschnitt daraus dargestellt werden.

Ein Layout basiert letztendlich immer auf Liniensystemen, selbst wenn von einem Liniensystem nur eine bestimmte Stimme, also nur eine beliebige Teilmenge der darin darzustellenden Noten und Pausenzeichen, gewünscht wird. Das bedeutet auch keine wesentliche Einschränkung, wenn bei einer Darstellung die Liniensysteme selbst unsichtbar bleiben sollen, denn in diesem Fall kann ihnen die Farbe des Hintergrunds zugewiesen werden.

Ein Notations-Controller wird als „Layout-Manager“ bezeichnet. Änderungen am Notations-Modell werden über den Layout-Manager jeweils an davon betroffene Layouts weitergegeben. Der Layout-Manager reagiert in diesem Fall also auch auf „Benutzereingaben“, nur ist der Benutzer in diesem Fall ein Programm und die Reaktionen des Layout-Managers beschränken sich auf die Layouts, denn das Notations-Modell wird jeweils vom Programm direkt geändert.

Ein Layout-Manager hält jeweils eine Programm-View von MUNOS zusammen. Wenn beispielsweise Liniensysteme oder Stimmen erzeugt werden, muß jeweils mit angegeben werden, welcher Layout-Manager für diese zuständig ist und damit, zu welchem Notations-Modell einer Programm-View von MUNOS diese gehören. Deshalb muß ein Programm, das mit MUNOS eine Programm-View auf ein Modell des Programms erzeugt, als erstes einen Layout-Manager für diese Programm-View von MUNOS erzeugen. Allerdings steht es jedem Entwickler frei dies zu ändern. Mehr darüber findet sich in Kapitel 8.

Als nächstes geht es um die Funktionalität, die MUNOS einem Programm bietet, um eine Programm-View auf ein Modell des Programms zu erzeugen, zu ändern oder zu löschen. Dazu wurden die Funktionen in drei Blöcke eingeteilt: Funktionen für ein Notations-Modell, Funktionen für eine Notations-View und Funktionen für einen Notations-Controller.

## 4.2 Funktionen für ein Notations-Modell

Es werden nun die Funktionen besprochen, die MUNOS einem Programm zur Verfügung stellt, um für eine Programm-View ein Notations-Modell zu erstellen, zu ändern und wieder zu löschen.

### 4.2.1 Die Repräsentation eines Tones erstellen, ändern oder löschen

Wie in 4.1 bereits beschrieben wurde, wird von einem Programm an MUNOS ein Ton übergeben, der nicht unbedingt dessen richtiger Schreibweise in Noten entspricht. Deshalb wird der Ton von MUNOS so in Form von Noten repräsentiert, wie dieser normalerweise notiert würde (vgl. Roemer, 1973). Das Programm kann dabei durch zahlreiche Vorgabemöglichkeiten in die Interpretation von MUNOS eingreifen. Die Repräsentation eines Tones kann bei Bedarf auch geändert oder gelöscht werden.

Wenn die Repräsentation eines Tones erstellt wird, muß dazu angegeben werden, in welchem Takt der Ton an welcher Taktposition einsetzt, wie lange der Ton andauert, in welchen Liniensystemen (vgl. 4.2.3, S. 26) der Ton dargestellt werden soll und an welcher vertikalen Position innerhalb der Liniensysteme die Noten des Tons jeweils stehen sollen. In MUNOS spielt die Tonhöhe eines Tons keine Rolle, sondern nur deren graphische Repräsentation in Form einer vertikalen Positionierung der Noten des Tons innerhalb eines Liniensystems. Die Positionsangaben beziehen sich hierbei auf die Linien bzw. Zwischenräume des jeweiligen Liniensystems, also z.B. „dritter Zwischenraum“ oder „zweite obere Hilfslinie“. Das hat den Vorteil, von der Semantik der Position unabhängig zu sein und so mit verschiedenen Arten von Liniensystemen und Notenschlüsseln arbeiten zu können, ohne deren jeweilige Semantik kennen zu müssen.

Dadurch wird es für einen Entwickler möglich, beliebige Liniensysteme oder Notenschlüssel einzuführen und damit zu arbeiten, ohne an dieser Stelle etwas ändern zu müssen.

Optional können zu einem Ton beliebig viele Stimmen (vgl. 4.2.4, S.27) angegeben werden, denen er zugeordnet ist und es kann außerdem ein beliebiges Graphikobjekt spezifiziert werden, das als Notenkopf verwendet werden soll. Allerdings besteht im letzteren Fall die Repräsentation des Tones generell aus nur einer Note, die dann das gewünschte Graphikobjekt als Notenkopf besitzt. Die Option, ein Graphikobjekt anzugeben, ist eigentlich nur für Sonderfälle gedacht, denn es besteht in MUNOS die Möglichkeit auf die Graphikobjekte, die normalerweise als Notenköpfe verwendet, Einfluß zu nehmen, ohne dabei die Interpretation von MUNOS zu beeinträchtigen. Mehr darüber findet sich in Kapitel 8.

Auch bei den Angaben zu Akzidentien wird in MUNOS die Semantik von der Graphik getrennt behandelt. Einerseits wird in MUNOS mit einem Akzidens keine Tonhöhenveränderung verbunden, sondern die Semantik beschränkt sich darauf, ob es sich bei einem Akzidens um ein Auflösungszeichen oder ein Vorzeichen handelt. Mehr Informationen sind für die korrekte graphische Darstellung eines Akzidens nicht notwendig. Andererseits kann zu jedem Akzidens ein beliebiges Graphikobjekt für dessen Darstellung angegeben werden, ohne dessen Semantik in MUNOS zu verändern. Die Interpretation in MUNOS wird in diesem Fall also nicht beeinträchtigt. Dadurch steht es beispielsweise einem Entwickler frei, ob und wo er Warnungsakzidentien verwendet und wie diese jeweils aussehen.

## 4.2.2 Die Repräsentation einer Pause erstellen, ändern oder löschen

Die Repräsentation einer Pause in Pausenzeichen wird analog zu der Repräsentation eines Tones in Noten erstellt (vgl. 4.2.1). Auch hier wird die Pause von MUNOS so in Form von Pausenzeichen repräsentiert, wie diese normalerweise notiert würde (vgl. Roemer, 1973). Ein Eingriff in die Interpretation von MUNOS ist genauso möglich wie die optionale Angabe eines beliebigen Graphikobjektes, das als Pausenzeichen dienen soll. Letzteres wieder mit der Konsequenz, die Interpretation zu umgehen und die Pause

als einzelnes Pausenzeichen in Form des gewünschten Graphikobjekts darzustellen. Aber auch bei Pausenzeichen läßt sich auf die Wahl der Graphikobjekte, die MUNOS ständig verwendet, Einfluß nehmen, ohne die Interpretation zu beeinträchtigen. Ein Unterschied zu Repräsentationen von Tönen, ist die vertikale Positionsangabe innerhalb des jeweiligen Liniensystems, die für Pausenzeichen nicht an Linien oder Zwischenräumen orientiert ist, sondern in Form einer lokalen Verschiebung des Pausenzeichens gegenüber dem jeweiligen Liniensystem angegeben wird.

Eine Variante der Repräsentation einer Pause ist die Repräsentation einer fiktiven Pause, mit der MUNOS neben realen Pausen intern arbeitet. Eine fiktive Pause ist eine Pause an einer Stelle in einem Liniensystem, an der bisher weder eine Note noch ein Pausenzeichen einer realen Pause steht. Damit werden Lücken innerhalb eines Liniensystems aufgefüllt. Die fiktiven Pausen werden dann genauso graphisch dargestellt wie reale Pausen. Die graphische Darstellung fiktiver Pausen läßt sich aber auf Wunsch auch unterdrücken. Eine weitere Anwendung fiktiver Pausen wird in 4.2.4 besprochen.

Wie zu einem Ton, können auch zu einer Pause beliebig viele Stimmen (vgl. 4.2.4, S. 27) angegeben werden, denen sie zugeordnet ist. Die Repräsentation einer Pause kann auch geändert und wieder gelöscht werden.

### **4.2.3 Ein Liniensystem erzeugen, ändern oder löschen**

Liniensysteme können erzeugt, geändert und wieder gelöscht werden. MUNOS verfügt vorläufig nur über das herkömmliche 5-Linien-System. Im Laufe der Zeit sollen noch mehr Arten von Liniensystemen hinzukommen. Ein Liniensystem verwaltet neben den Noten und Pausenzeichen auch alle Informationen über die Vorzeichnungen, die in ihm dargestellt werden. Dazu gehören der Notenschlüssel, Angaben zur Taktart und zur Tonart, sowie eventuelle Wechsel derselben. Alle Vorzeichnungen sind optional, müssen also nicht unbedingt zu einem Liniensystem angegeben werden.

#### 4.2.4 Eine Stimme erzeugen oder löschen

Eine Stimme ist in MUNOS als eine beliebige Menge von Tönen und Pausen definiert. Jedem Ton und jeder Pause können mehrere unabhängige Stimmen zugeordnet werden. Durch diese freie Art mit Stimmen umzugehen, wird einem Entwickler die Möglichkeit gegeben, Darstellungsfiler zu realisieren, indem sein Programm mit jeder Stimme eine ganz bestimmte Eigenschaft assoziiert, die MUNOS nicht zu kennen braucht. Wenn sein Programm dann mit MUNOS das Layout einer bestimmten Stimme erstellt (vgl. 4.3.1, S. 28), so kann er auf diese Weise alle Noten darstellen lassen, die über diese Eigenschaft verfügen.

Zum Beispiel kann ein Entwickler in seinem Programm eine Stimme mit dem gesamten Schlagzeug und weitere Stimmen jeweils mit den konkreten Klangerzeugern des Schlagzeugs assoziieren. Eine Klangerzeuger-Stimme ist in diesem Fall die Stimme der Fellinstrumente des Schlagzeugs. Um sich dann von der Schlagzeug-Stimme nur die Noten der Fellinstrumente zeigen zu lassen, läßt der Entwickler sein Programm von MUNOS ein Layout der Stimme erstellen, die er im Programm mit den Fellinstrumenten assoziiert hat. Zwar gehört jede der Noten der Fellinstrument-Stimme gleichzeitig auch zur Schlagzeug-Stimme, aber diese ist im Fall der Fellinstrumente nicht in ihrer Gesamtheit gefragt. Deshalb kommen in diesem Beispiel auch fiktive Pausen ins Spiel (vgl. 4.2.2, S. 26). Die Stellen im Liniensystem des Schlagzeugs, an denen sonst Noten der Schlagzeug-Stimme stehen, die nicht von Fellinstrumenten gespielt werden, erfahren eine Behandlung wie Lücken und werden automatisch mit Pausenzeichen fiktiver Pausen ausgefüllt.

Eine Stimme kann auch wieder gelöscht werden. Nur die Änderung einer Stimme wird nicht notwendig, denn einerseits erfolgt die Zuordnung einer Stimme jeweils in den Repräsentationen der Noten bzw. der Pausen (vgl. 4.2.1, S.25 bzw. 4.2.2, S. 26) und andererseits wird in MUNOS selbst mit einer Stimme keine Eigenschaft assoziiert.

### 4.2.5 **Eine Bezifferung erzeugen, ändern oder löschen**

Eine Bezifferung gibt über den harmonischen Kontext Aufschluß, in dem eine Note zu sehen ist (vgl. 2.1, S. 12). MUNOS bietet die Möglichkeit, eine Note mit einer oder mehreren Bezifferungen zu versehen und bei Bedarf diese Bezifferungen wieder zu ändern oder zu löschen. Eine Bezifferung besteht aus bestimmten Ziffern und Zeichen, die unterhalb eines Liniensystems unter der Note, auf die sie sich beziehen, geschrieben werden. Diese Ziffern und Zeichen werden an MUNOS einfach in Form von Text übergeben.

Wenn zu einer Note mehrere Bezifferungen gehören, so wechselt der harmonische Kontext der Note noch während diese andauert, d.h. die Dauer der Gültigkeit einer Bezifferung muß nicht mit der Dauer der Note, auf die sie sich bezieht, identisch sein. Deshalb kann mit jeder Bezifferung deren Gültigkeitsdauer mit angegeben werden. Die Bezifferungen werden in diesem Fall jeweils an der Stelle eines Taktes unter der Note geschrieben, ab der sie gelten.

## 4.3 **Funktionen für eine Notations-View**

Es werden nun die Funktionen besprochen, die MUNOS einem Programm bietet, um für eine Programm-View eine Notations-View zu erstellen, zu ändern und wieder zu löschen oder eine Kopie des Bild-Objekts einer Notations-View zu erzeugen, die aus dem Notations-Modell-View-Controller-Kontext der Programm-View herausgelöst ist.

### 4.3.1 **Ein Layout erstellen, ändern oder löschen**

Es können Layouts erstellt, geändert und wieder gelöscht werden. Jedes Layout basiert letztendlich auf einer Anzahl von Liniensystemen aus dem Notations-Modell. Um ein Layout zu erstellen, werden entweder die Liniensysteme direkt angegeben oder es wird eine beliebige Anzahl von Stimmen angegeben, für die das Layout gemacht werden soll. Wenn Stimmen angegeben wurden, so wird das Layout auf den Liniensystemen aufgebaut, die zur Darstellung der Stimmen notwendig sind. Es basiert im Falle von Stimmen also ebenfalls auf Liniensystemen, wenn auch indirekt, und es werden dann nur die Noten und Pausen in den Liniensystemen dargestellt, die zu den angegebenen Stimmen gehören.

Ein Layout enthält die gewünschte Darstellung immer in Form eines Bild-Objekts, das vom Programm auf einem beliebigen Ausgabegerät dargestellt werden kann. Jedes Graphikobjekt, das zu dem Bild-Objekt gehört, hat im Layout ein Vorgabeobjekt, in dem alle Daten festgehalten sind, die für dessen Erzeugung und Darstellung relevant sind. Deshalb kann eine Note aus dem Notations-Modell in verschiedenen Views auch verschieden dargestellt werden. In jedem Graphikobjekt werden sowohl dessen Vorgabeobjekt als auch die semantischen Informationen, was durch das Graphikobjekt dargestellt wird, festgehalten. So ermöglicht MUNOS einem Programm, Benutzereingaben entsprechend auszuwerten (vgl. 3.2, S. 18). Wenn sich dabei Änderungen am Notations-Modell ergeben, dann werden davon betroffene Layouts automatisch aktualisiert, d.h. an die Änderungen im Notations-Modell angepaßt.

### 4.3.2 Ein Layout einfrieren

Es besteht die Möglichkeit, ein Layout einfrieren zu lassen. In diesem Fall wird eine Kopie des Bild-Objekts des Layouts erstellt, allerdings ohne dessen semantische Informationen zu übernehmen. Es handelt sich bei dieser Kopie um ein Bild-Objekt, das vom Notations-Modell und vom Layout abgekoppelt ist. Dieses kann jederzeit als Bild wieder dargestellt, aber nicht mehr aktualisiert werden und zeigt daher einen „eingefrorenen“ Zustand des Notations-Modells.

Das Programm erhält von MUNOS dann die gewünschte Kopie des Bild-Objekts zurück und ist allein für ein späteres Löschen dieser Kopie verantwortlich, d.h. die Kopie wird auch beim Löschen des Layouts nicht mitgelöscht. Ein Programm wird dadurch zum Beispiel in die Lage versetzt, die Entstehung eines Stückes im Notenbild festzuhalten und darzustellen oder Bilder über die Zwischenablage an andere Programme zu übergeben.

## 4.4 Funktionen für einen Notations-Controller

Es werden nun die Funktionen besprochen, die MUNOS einem Programm zur Verfügung stellt, um für eine Programm-View einen Notations-Controller zu erstellen und wieder zu löschen.

### 4.4.1 Einen Layout-Manager erzeugen oder löschen

Es können Layout-Manager erzeugt und wieder gelöscht werden. Ein Layout-Manager ist jeweils für genau eine Programm-View von MUNOS verantwortlich und gibt darin Änderungen am Notations-Modell an die betroffenen Layouts und deren Objekte weiter (vgl. 4.1, S.23). Ein Notations-Modell, ein Layout-Manager und beliebig viele Layouts bilden so gesehen ein in sich abgeschlossenes System, also eine autarge Programm-View von MUNOS. Es steht einem Entwickler allerdings die Möglichkeit offen, diese Verhältnisse zu ändern und beispielsweise mit mehreren Layout-Managern pro View zu arbeiten, die jeweils für ganz bestimmte Änderungen zuständig sind. Wie das MUNOS ermöglicht, wird in Kapitel 8 besprochen.

MUNOS stellt einen Standard-Typ für einen Layout-Manager zur Verfügung. Wenn ein Programm für seine Programm-View in MUNOS einen Standard-Layout-Manager erzeugt, ist in diesem Fall gleichzeitig auch die Funktionsweise des Layout-Managers festgelegt, denn darauf kann kein Einfluß genommen werden. Es können daher an einem erzeugten Standard-Layout-Manager auch keine Änderungen mehr vorgenommen werden. Allerdings steht es jedem Entwickler frei, MUNOS um seine eigenen Layout-Manager-Typen zu ergänzen. Wie das MUNOS ermöglicht, wird, wie bereits oben erwähnt wurde, in Kapitel 8 besprochen.

## 5 Ein abstraktes Modell für objektorientierte Methoden

Bei der Entwicklung von MUNOS wird eine objektorientierte Methode angewendet, die speziell für dieses Projekt aus verschiedenen anderen Methoden zusammengestellt wurde. Bei der Zusammenstellung dieser Methode diente ein abstraktes Modell für objektorientierte Methoden als eine Art „Schablone“. Doch eigentlich ist dieses Modell gar nicht dafür gedacht. Deshalb wird in diesem Kapitel zunächst auf den eigentlichen Zweck des Modells eingegangen. Dann wird die Idee hinter der Methode des MUNOS-Projekts geschildert, d.h. warum anhand des Modells eine Methode zusammengestellt werden kann und wie so etwas funktioniert. Im Rest des Kapitels wird schließlich das Modell selbst besprochen.

### 5.1 Der eigentliche Zweck des Modells

Ein Problem beim Vergleich und Umgang mit verschiedenen objektorientierten Methoden liegt in deren unterschiedlicher Terminologie. Teilweise besitzt ein Sachverhalt in verschiedenen Methoden verschiedene Bezeichnungen oder eine Bezeichnung wird in verschiedenen Methoden jeweils für einen anderen Sachverhalt verwendet. So werden die Attribute von Objekttypen bei Booch beispielsweise als „Felder“ („fields“) bezeichnet (vgl. Booch, 1994). Manchmal gibt es auch in einzelnen Methoden kein entsprechendes Pendant für eine bestimmte Bezeichnung. Zum Beispiel ist ein „Objekttyp“ eine semantische Eigenschaft und dessen syntaktische Repräsentation ist dann eine „Klasse“. Doch wird in einigen Methoden, wie bei Jacobson, nicht zwischen Semantik und Syntax unterschieden und in beiden Fällen die Bezeichnung „Klasse“ verwendet.

Etwas Ordnung in diesen Dschungel der Terminologie verschiedener Methoden zu bringen, ist eines der Ziele der „**Object Management Group**“ (OMG). Das generelle Ziel der OMG besteht darin, möglichst rasch Standards für den objektorientierten Bereich zu entwickeln, so auch für den Teilbereich der objektorientierten Entwicklung. Um eine Ausgangsbasis für die Entwicklung von Standards im Teilbereich der objektorientierten Entwicklung zu schaffen, wurden zunächst 16 verschiedene objektorientierte Methoden miteinander verglichen.

Der Vergleich erfolgte anhand eines „technical framework“, in dem eine einheitliche Terminologie, in einem abstrakten Modell für objektorientierte Methoden verpackt wurde. Dieses abstrakte Modell für objektorientierte Methoden wurde aus dem Vergleich verschiedener objektorientierter Methoden heraus entwickelt, um damit einen Vergleich objektorientierter Methoden zu ermöglichen. Das abstrakte Modell wurde dann zusammen mit dem Vergleich der Methoden veröffentlicht (vgl. Hutt/OMG, 1993).

## 5.2 Die Idee hinter der Methode des MUNOS-Projekts

Das in der Veröffentlichung der OMG beschriebene abstrakte Modell für objektorientierte Methoden ist in verschiedene Teilbereiche untergliedert (vgl. Abb. 5.1, S. 34), die jeweils separat verglichen werden können. Bei dem Vergleich der objektorientierten Methoden innerhalb dieser Teilbereiche fallen zwei Dinge auf. Einerseits werden in manchen Methoden verschiedene Tätigkeiten oder Konzepte für denselben Teilbereich angeboten, mit denen jeweils dasselbe Ziel verfolgt wird. Diese können dann entweder ergänzend oder alternativ eingesetzt werden, je nachdem, was im Einzelfall erforderlich bzw. machbar ist. Ein Beispiel dafür wären die zahlreichen Wege, um Objekttypen zu finden. Andererseits sind in einzelnen Teilbereichen viele Tätigkeiten oder Konzepte der einzelnen Methoden deckungsgleich, d.h. es gibt viele Überschneidungen zwischen objektorientierten Methoden. Zum Beispiel sind die Wege, um Objekttypen zu finden, auch überall sehr ähnlich.

Das legt den Gedanken nahe, nicht eine bestimmte Methode anzuwenden, sondern das abstrakte Modell als eine Art „Schablone“ zu verwenden und sich selbst eine Methode zusammenzustellen, die genau auf das jeweilige Projekt zugeschnitten ist. Zumal es mittlerweile Werkzeuge gibt, die verschiedene Methoden gleichzeitig unterstützen.

Eine Methode wird dann zusammengestellt, indem man sich für jeden Teilbereich des abstrakten Modells aus dem Pool einzelner Tätigkeiten und Konzepte bekannter Methoden bedient und sich daraus jeweils diejenigen auswählt, die einem für das geplante Projekt am geeignetsten erscheinen.

Auf diese Art und Weise ist dann auch die Methode zustande gekommen, die im MUNOS-Projekt verwendet wird. Das Ganze läßt sich ein wenig mit dem Zusammenstellen eines mehrgängigen Menüs vergleichen. Das abstrakte Modell würde dann vorgeben, welche Gänge zu einem Menü gehören. Die Tätigkeiten oder Konzepte einzelner Methoden, die jeweils einem Teilbereich des abstrakten Modells zugeordnet sind, wären die verschiedenen Alternativen für einen einzelnen Gang, also beispielsweise verschiedene denkbare Vorspeisen. Die schließlich kreierte Methode wäre das Menü. Die Kunst besteht dann darin, ein Menü zusammenzustellen, bei dem die einzelnen Gänge nicht nur zubereitet werden können, sondern auch gut zusammenpassen.

Ein Haken an diesem Vergleich sind die phasenübergreifenden Konzepte einer Methode, die sich damit weniger gut illustrieren lassen. Wie wäre es zum Beispiel mit dem Arrangement des Tischschmucks und dem Motto eines Menüs ?

### 5.3 Das abstrakte Modell

Im Rest dieses Kapitels wird nun näher auf das abstrakte Modell eingegangen, wie es in Hutt/OMG (1993) veröffentlicht wurde. Dazu zeigt die Abbildung 5.1 auf der nächsten Seite ein Gesamtüberblick. Es handelt sich dabei um die eingedeutschte Version eines Diagramms der OMG, welches das abstrakte Modell im Gesamtkontext einer objektorientierten Entwicklung zeigt. Die einzelnen Teilbereiche des abstrakten Modells sind jeweils schattiert dargestellt.

Die Ausgangsbasis einer Entwicklung bilden jeweils die phasenübergreifenden Konzepte einer Methode. Aufbauend auf den phasenübergreifenden Konzepten, führt die Anwendung der Modellierungskonzepte für die einzelnen Entwicklungsphasen im Rahmen entsprechender Tätigkeiten zu der vollständigen Beschreibung eines Systems („full definition of system“).

Parallel dazu müssen eventuell sowohl die Entwicklergruppen innerhalb eines Projekts als auch die gleichzeitig laufenden Projekte jeweils miteinander koordiniert werden. Zusätzlich können auch noch Maßnahmen getroffen werden, um Ergebnisse oder Teilergebnisse einzelner Projekte anderen Projekten zugänglich zu machen, damit diese dort bei Bedarf wiederverwendet werden können.

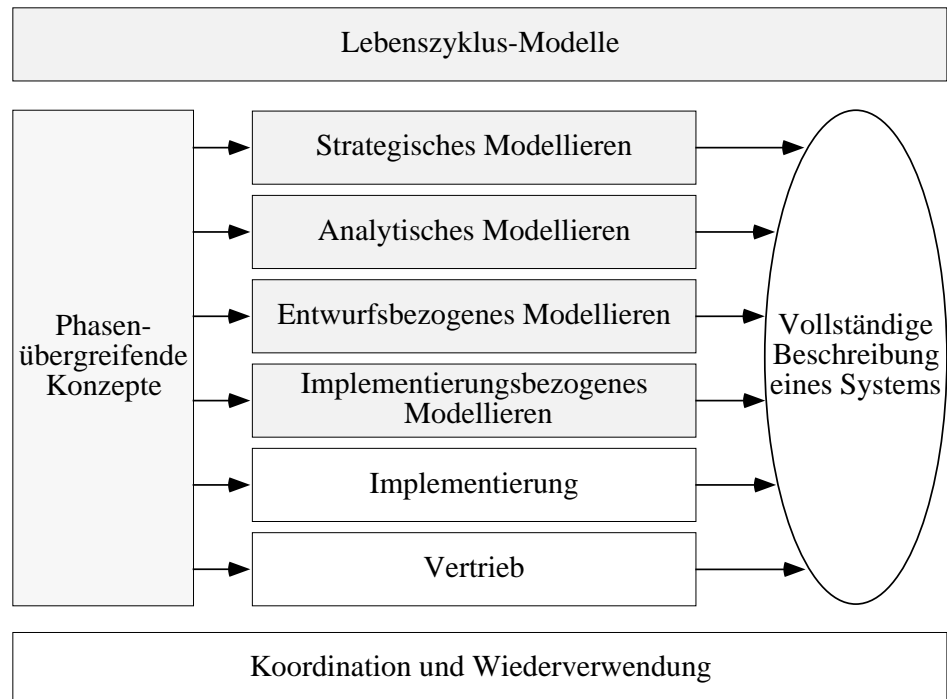


Abb. 5.1: Abstraktes Modell für objektorientierte Methoden

Nun wird zunächst erläutert, was unter den phasenübergreifenden Konzepten einer Methode zu verstehen ist. Anschließend wird auf die einzelnen Phasen einer objektorientierten Entwicklung und die Lebenszyklus-Modelle, die dahinter stehen können, eingegangen.

### 5.3.1 Phasenübergreifende Konzepte

Die phasenübergreifenden Konzepte einer Methode werden im Original unter „object modeling“ zusammengefaßt. Die Originalbezeichnung erscheint mir etwas zu spezifisch, denn unter dieser Bezeichnung werden nicht nur die Konzepte zusammengefaßt, die eine Methode bietet, um Objekte zu modellieren, sondern dazu gehören allgemein die Konzepte einer Methode, die phasenübergreifend sind, wie zum Beispiel die Konzepte, um die Ergebnisse der einzelnen Phasen einer Entwicklung miteinander zu verbinden oder noch weiter reichende Konzepte, mit deren Hilfe ein roter Faden durch eine gesamte Entwicklung gebildet werden kann, wie zum Beispiel das Use-Case-Konzept, auf das in Kapitel 6 noch ausführlich eingegangen wird.

Unter den Konzepten, um Objekte zu modellieren, versteht man zum Beispiel Beziehungstypen, wie Vererbung, und Kommunikationsarten, wie Nachrichten, sowie graphische Ausdrucksmittel dafür, d.h. eine Notation.

### 5.3.2 Phasen einer objektorientierten Entwicklung

Um möglichst vielen Methoden gerecht zu werden, hat die OMG in die einzelnen Phasen jeweils sehr viel hineingepackt, was dort möglicherweise alles getan wird. Es wurde daher in der folgenden Beschreibung versucht, jeweils das Wesentliche herauszuholen, das für das MUNOS-Projekt relevant ist.

Eine objektorientierte Entwicklung wird von der OMG in folgende Phasen unterteilt:

- Strategisches Modellieren („strategic modeling“). Das strategische Modellieren besteht im wesentlichen darin, ein Projekt zu planen.
- Analytisches Modellieren („analysis modeling“). Das analytische Modellieren besteht im wesentlichen darin, eine Spezifikation zu erstellen.
- Entwurfsbezogenes Modellieren („design modeling“). Das entwurfsbezogene Modellieren besteht im wesentlichen darin, einen implementierungsunabhängigen Entwurf zu erstellen. In diesem Zuge werden auch strukturelle Maßnahmen zur Verbesserung der Produktqualität und der Gebrauchqualität getroffen.
- Implementierungsbezogenes Modellieren („implementation modeling“). Das implementierungsbezogene Modellieren besteht darin, den implementierungsunabhängigen Entwurf an implementierungsabhängige Gegebenheiten anzupassen, d.h. an eine bestimmte Plattform, die verwendete Programmiersprache, die zum Einsatz kommende Klassenbibliothek, Datenbank, etc.
- Implementierung („construction“). In dieser Phase wird der implementierungsabhängige Entwurf auf der Zielplattform implementiert.
- Vertrieb („delivery“). Erste Testversionen werden verteilt bzw. das fertige Produkt wird ausgeliefert.

### 5.3.3 Lebenszyklus-Modelle

In dem abstrakten Modell der OMG wird mit einer objektorientierten Entwicklung kein bestimmtes Lebenszyklus-Modell in Verbindung gebracht, sondern dort werden Progressionsstrategien, Iterationsstrategien, Verpackungsstrategien und Meilensteinstrategien definiert, anhand deren sich Lebenszyklus-Modelle charakterisieren lassen. Diese Strategien werden nun beschrieben.

#### 5.3.3.1 Progressionsstrategien

Durch eine Progressionsstrategie („progression strategy“) wird vorgegeben, wie die Ergebnisse der einzelnen Entwicklungsphasen generell zusammenhängen.

Es wird zwischen zwei Progressionsstrategien unterschieden:

- **additiv („additive“):** die Ergebnisse der einzelnen Phasen sind jeweils Teilergebnisse oder Verfeinerungen bisheriger Ergebnisse und können direkt in die nächste Phase übernommen werden.
- **transformierend („transformationale“):** das Ergebnis einer Phase muß auf irgendeine Art und Weise transformiert werden, um in der darauffolgenden Phase verwendet werden zu können.

### 5.3.3.2 *Iterationsstrategien*

Mit einer **Iteration** ist in diesem Kontext ein einzelner Durchlauf der Entwicklungsphasen gemeint. Durch eine Iterationsstrategie („iteration strategy“) wird vorgegeben, wie oft die einzelnen Phasen durchlaufen werden und welche Qualität die Ergebnisse der einzelnen Iterationen jeweils besitzen sollen.

Es wird zwischen vier Iterationsstrategien unterschieden:

- nur-einmal („once-only“): die gesamte Entwicklung erfolgt innerhalb einer Iteration, d.h. jede Phase wird nur einmal durchlaufen, wie es beispielsweise für das Wasserfallmodell (vgl. Sommerville, 1992, S. 6ff) typisch ist.
- evolutionär („evolutionary“): die Entwicklung erfolgt in mehreren Iterationen, wobei in der ersten Iteration bereits das gesamte System entwickelt wird. Allerdings muß das System nach der ersten Iteration nur zu ca. 70% das richtige System sein („...to produce a 70%-right system...“). Dann werden Testversionen herausgegeben und das System wird in weiteren Iterationen zur Marktreife gebracht. Eine verwertbare Interpretation des zu „70% richtigen Systems“ wird von der OMG leider nicht mitgeliefert. Vermutlich handelt es sich in diesem Fall um Prototypen-Entwicklung, wobei die Prototypen zur Marktreife weiterentwickelt werden (vgl. *rapid prototyping*).
- rapid prototyping: es wird versucht, in möglichst kurzer Zeit zu einem lauffähigen System zu kommen. In diesem Fall geht es allerdings nur darum, mit Hilfe des Systems die Benutzeranforderungen zu klären oder zu testen, ob etwas machbar ist. Um das System dann auf den Markt zu bringen, wird es anschließend vollständig neu implementiert.
- inkrementell („incremental“): das geplante System wird in mehrere Subsysteme unterteilt. Bei jeder Iteration wird jeweils ein Subsystem gleich marktreif entwickelt.

### 5.3.3.3 **Verpackungsstrategien**

Durch eine Verpackungsstrategie („packaging strategy“) wird vorgegeben, wie einzelne Tätigkeiten einer Entwicklung (z.B. Subsysteme bilden) in die einzelnen Entwicklungsphasen „verpackt“ werden, d.h. wie die Zuordnung der einzelnen Tätigkeiten auf die Entwicklungsphasen gehandhabt wird. Eine Verpackungsstrategie ist eine organisatorische Maßnahme des Managements und deshalb meist von den äußeren Randbedingungen eines Projekts abhängig. Beispielsweise kann die Verpackungsstrategie davon abhängen, was wann wo verfügbar ist.

Es wird zwischen drei Verpackungsstrategien unterschieden:

- feste Phasenzuordnung („fixed stage“): eine bestimmte Tätigkeit wird immer in derselben Entwicklungsphase ausgeführt, wenn auch eventuell nur optional. Dabei kann es für eine Tätigkeit innerhalb einer Entwicklungsphase durchaus auch alternative Tätigkeiten geben.
- flexible Phasenzuordnung („flexible stage“): die Tätigkeiten werden jeweils so auf die einzelnen Phasen verteilt, wie es für das jeweilige Projekt am günstigsten erscheint.
- keine Phasenzuordnung („nonstaged“): es werden keine Tätigkeiten auf Entwicklungsphasen verteilt. Von dieser Verpackungsstrategie gibt es zwei Extreme. Entweder werden die einzelnen Phasen ohne jegliche Zwischenkontrolle starr durchlaufen oder es existieren keine Phasen mehr und alle Tätigkeiten können weitgehendst in beliebiger Reihenfolge ausgeführt werden. Im letzteren Fall wird die ausführbare Reihenfolge der Tätigkeiten höchstens durch Regeln eingeschränkt, die in der verwendeten Methode vorgegeben werden. Beispielsweise könnte es verboten sein, Vererbungsbeziehungen festzulegen, bevor die Rollen der beteiligten Objekte feststehen. Besonders bei Methoden, in denen eine additive Progressionsstrategie angewandt wird, verschwimmen häufig einzelne Phasenübergänge und eine Zuordnung einzelner Tätigkeiten zu bestimmten Phasen ist gar nicht mehr möglich.

#### 5.3.3.4 **Meilensteinstrategien**

Durch eine Meilensteinstrategie („checkpointing strategy“) wird vorgegeben, wie der Umgang mit Meilensteinen gehandhabt wird.

Es wird zwischen drei Meilensteinstrategien unterschieden:

- genehmigungspflichtig („rubber-stamp“): wenn das Projekt an einen genehmigungspflichtigen Meilenstein gelangt ist, dann gibt das Management entweder grünes Licht und das Projekt kann ohne Änderungen fortgesetzt werden oder das Projekt wird abgebrochen. In diesem Fall werden weder Zwischenergebnisse überarbeitet, noch wird das Projekt an die gegebenen Umstände angepaßt. Die Entscheidung, ob ein Projekt fortgesetzt wird, muß nicht einmal von den Zwischenergebnissen abhängen, sondern kann auf rein firmenpolitischen Erwägungen beruhen.
- Management-Review: wenn das Projekt an einen Management-Review Meilenstein gelangt ist, dann wird das Projekt auch entweder fortgesetzt oder abgebrochen. Bevor es allerdings zu einem Abbruch kommt, wird eher der Projektumfang geändert, werden Zwischenergebnisse überarbeitet oder Gewichtungen innerhalb des Projekts verlagert.
- Risiko-gesteuert („risk-driven“): wenn das Projekt an einen Risiko-gesteuerten Meilenstein gelangt ist, werden vom Management formale Verfahren zur Verminderung von Risiken angewandt, wie es zum Beispiel auch in Boehms Spiralenmodell (vgl. Sommerville, 1992, S. 15ff) beschrieben wird. Als Ergebnis kann dann beispielsweise ein anderes Lebenszyklus-Modell adaptiert werden oder es können parallele Projekte ins Leben gerufen werden oder das bestehende Projekt kann neu organisiert und umbesetzt werden. Im schlimmsten Fall wird aber auch hier das Projekt abgebrochen.

## 6 Die Entwicklungsmethode

In diesem Kapitel wird zunächst in einer grundsätzlichen Überlegung dargelegt, warum die Auftrennung einer Entwicklung in einzelne Phasen durchaus sinnvoll ist. Anschließend wird die im MUNOS-Projekt verwendete Entwicklungsmethode anhand des abstrakten Modells für objektorientierte Methoden charakterisiert (vgl. 5.3, S. 33ff). Dabei wird mit den phasenübergreifenden Konzepten begonnen, gefolgt von den Tätigkeiten und Ergebnissen der einzelnen Phasen. Die Beschreibung der Methode endet schließlich mit der Charakterisierung des Lebenszyklus-Modells.

### 6.1 Warum Phasen ?

Die Frage, ob die Auftrennung einer Entwicklung in einzelne Phasen überhaupt sinnvoll und wünschenswert ist, gibt immer wieder Anlaß zur Diskussion. Zum Beispiel empfiehlt die OMG für eine objektorientierte Entwicklung Lebenszyklus-Modelle, die keine Auftrennung in einzelne Phasen vorsehen (vgl. Hutt/OMG, 1994, S. 155). Vergleicht man in diesem Zusammenhang die Lebenserwartung einer Software mit der einer Hardware, ist es mittlerweile fast normal, wenn eine Software die Hardware, auf der sie läuft, überlebt. Längerfristig gesehen, ist es daher sinnvoll, von Anfang an plattformunabhängig zu entwickeln. Dazu ist es wiederum nötig, eine klare Trennung zwischen dem plattformabhängigen und dem plattformunabhängigen Modell einer Entwicklung vorzunehmen. Nun können Entwicklungsphasen bis zu einem gewissen Grade auch parallel ablaufen. Um einen Überblick zu behalten, was zu einer bestimmten Phase gehört, ist es deshalb außerdem sinnvoll, die einzelnen Phasen jeweils an den Teilergebnissen festzumachen, die diese Phasen liefern. Wenn dann ein plattformabhängiges Modell und ein plattformunabhängiges Modell zwei verschiedene Teilergebnisse einer Entwicklung darstellen, so hat man zwangsläufig auch schon zwei verschiedene Entwicklungsphasen. Deshalb ist die Auftrennung einer Entwicklung in einzelne Phasen durchaus sinnvoll und ist auch die Entwicklung von MUNOS in einzelne Phasen unterteilt.

## 6.2 Phasenübergreifende Konzepte

Es wird zunächst das Use-Case-Konzept vorgestellt, mit dessen Hilfe ein roter Faden durch die gesamte Entwicklung gebildet werden kann. Dann wird auf die Gewinnung und den Einsatz von Mustern eingegangen, der umfassendsten Art von Wiederverwendung, die man betreiben kann. Schließlich werden noch die Notation und deren Semantik besprochen.

### 6.2.1 Das Use-Case-Konzept

Das Use-Case-Konzept ist eine Idee von Ivar Jacobson, über die ein ständig wachsender Konsens zu herrschen scheint (vgl. Jacobson et al., 1993). So haben unter anderem bereits Booch, Rumbaugh und Wirfs-Brock dieses Konzept in ihre Methoden übernommen (vgl. Jacobson/Christerson, 1995). Das Use-Case-Konzept sieht vor, alle möglichen späteren Anwendungen eines Systems als Use-Cases zu modellieren, d.h. in Form von Use-Case-Beschreibungen festzuhalten.

Ein **Use-Case** ist ein Objekttyp für mögliche Anwendungsfälle. Wenn ein Use-Case durchgeführt wird, dann liegt ein konkreter Anwendungsfall vor, der einen der Abläufe nehmen kann, die in der Beschreibung des jeweiligen Use-Case beschrieben werden. Zum Beispiel ist „die Repräsentation eines Tones erstellen“ ein Use-Case aus der Entwicklung von MUNOS. Wenn MUNOS dann von einem Programm den Auftrag bekommt, die Repräsentation eines bestimmten Tones zu erstellen, liegt ein konkreter Anwendungsfall dieses Use-Case vor.

Ein Use-Case wird entweder von einem **Akteur** („actor“) oder von einem anderen Use-Case durchgeführt. Ein Akteur kann eine Person, eine Hardware oder eine Software sein. Dabei wird ein Akteur nicht fest mit einer bestimmten Person oder Sache verbunden, sondern mit einer bestimmten Rolle, die eine Person oder Sache in dem Moment übernimmt. Wenn beispielsweise ein Bankangestellter gleichzeitig Kunde bei seiner Bank ist, so kann er einmal in der Rolle des Kunden und ein andermal in der Rolle des Angestellten auftreten. Der Bankangestellte kann in diesem Fall eine von zwei verschiedenen Rollen annehmen, d.h. als einer von zwei verschiedenen Akteuren auftreten, und in Abhängigkeit seiner aktuellen Rolle jeweils andere Use-Cases durchführen.

Im Falle von MUNOS verhält sich die Sache wesentlich einfacher, denn es gibt nur einen Akteur in Form des Programms, von dem MUNOS benutzt wird.

Use-Cases, die nicht von Akteuren durchgeführt werden können, sind **abstrakte** Use-Cases. Abstrakte Use-Cases werden immer von anderen Use-Cases verwendet und damit nur indirekt von Akteuren durchgeführt. Meist entsteht ein abstrakter Use-Case, wenn man aus verschiedenen Use-Cases einen gemeinsamen Ablauf herauszieht und diesen Ablauf mittels eines neuen Use-Case modelliert, der selbst keine Akteure besitzt und von den Use-Cases, die jeweils diesen Ablauf enthalten haben, verwendet wird. Zum Beispiel kann aus den Use-Cases „Weihnachten feiern“ und „Geburtstag feiern“ der gemeinsame Ablauf „viele Geschenke bekommen“ herausgezogen und in einem abstrakten Use-Case untergebracht werden, der dann von den anderen beiden Use-Cases verwendet wird.

Abstrakte Use-Cases sind aber nur eine Teilmenge der Use-Cases, die von anderen Use-Cases **verwendet** werden, denn verwendete Use-Cases können auch Akteure haben und sind in dem Moment keine abstrakten Use-Cases mehr. Zum Beispiel wäre „Eis essen“ ein Use-Case, der für sich alleine durchgeführt werden kann oder von dem Use-Case „ein Menü zu sich nehmen“ beim Nachtschiff verwendet werden kann.

Neben Use-Cases, die von anderen Use-Cases verwendet werden, gibt es auch Use-Cases, die andere Use-Cases **ergänzen**. Zum Beispiel kann der Use-Case „einen Radiosender empfangen“ den Use-Case „eine Radiosendung senden“ zu dem Use-Case „Radio hören“ ergänzen.

Ergänzende und verwendete Use-Cases erlauben es einem, aus zweierlei Sichtweisen heraus modellieren zu können. Ein Use-Case, der einen anderen Use-Case verwendet, weiß von diesem Use-Case, aber ein Use-Case weiß nie, ob und von welchem Use-Case er verwendet wird. Zum Beispiel weiß das Menü von dem Eis das zu ihm gehört, aber das Eis weiß nicht, in welchem Zusammenhang es gegessen wird. Analog verhält es sich bei ergänzenden Use-Cases. Ein ergänzender Use-Case weiß, welchen Use-Case er ergänzt, aber ein Use-Case weiß nie, ob und von welchem Use-Case er ergänzt wird. Zum Beispiel weiß ein Empfänger von einem Radiosender, wenn er diesen Sender empfängt, aber ein Radiosender weiß nicht, ob er auch empfangen wird.

Deshalb kann auch ein abstrakter Use-Case nie ein ergänzender Use-Cases sein, denn sonst könnte dieser nie durchgeführt werden. Ein ergänzender Use-Case ergänzt immer nur die Funktionalität eines anderen, nicht abstrakten, Use-Case.

Einerseits lassen sich alle Probleme generell mit sich verwendeten Use-Cases lösen, denn ein Use-Case, der einen anderen Use-Case ergänzt, kann stattdessen auch von dem anderen Use-Case verwendet werden. Zwar wird in diesem Fall die Semantik umgedreht, aber das ändert nichts am Ergebnis. Diese Vorgehensweise wird zum Beispiel von Rumbaugh propagiert (vgl. Rumbaugh, 1994). Aber andererseits erlaubt die Semantik hinter ergänzenden Use-Cases, in entsprechenden Situationen, ein realistischeres Modellieren dieser Situationen.

Ein Beispiel, mit dem sich der semantische Unterschied zwischen ergänzenden und verwendeten Use-Cases gut verdeutlichen lässt, wäre das Abhören eines Telefons im Rahmen eines „kleinen Lauschangriffs“. Die Funktionalität der Telefonanlage würde im einen Fall durch den Abhörvorgang ergänzt. Die Abhöranlage „würde“ dann, um welches Telefon es geht und die normale Funktionalität der Telefonanlage bliebe erhalten. Das Umdrehen der Semantik im anderen Fall hätte dann einen Umbau der Telefonanlage zur Folge und für jedes Telefon würde künftig von der Telefonanlage automatisch überprüft, ob dieses abgehört werden soll. Obwohl in beiden Fällen als Ergebnis das gewünschte Telefon abgehört würde, wäre ich persönlich eher dafür, den Abhörvorgang in Form eines ergänzenden Use-Case zu modellieren. Im Falle von MUNOS gab es dagegen bislang keine Situation, in der die Anwendung eines ergänzenden Use-Case sinnvoll erschien.

Wie Use-Cases während der Entwicklung als „roter Faden“ dienen und einzelne Entwicklungsphasen miteinander verbinden, wird in 6.2 im Zusammenhang mit den Tätigkeiten und Ergebnissen der einzelnen Phasen besprochen. In Kapitel 7 wird dies dann ausführlich an einem Use-Case aus der Entwicklung von MUNOS gezeigt. In Kapitel 10 wird schließlich in einem kritischen Rückblick auf den Verlauf der Arbeit auch auf ein generelles Problem beim Umgang mit Use-Cases hingewiesen. Doch nun wird zunächst auf die Gewinnung und den Einsatz von Mustern eingegangen.

## 6.2.2 Gewinnung und Einsatz von Mustern

Ein **Muster** beschreibt ein bestimmtes, in einem gegebenen Kontext immer wiederkehrendes Problem, sowie ein vordefiniertes Schema zu seiner Lösung. Die Gewinnung und der Einsatz von Mustern ist eigentlich die umfassendste Art von Wiederverwendung, die man betreiben kann, nämlich die Wiederverwendung von Lösungen für Probleme auf allen Entwicklungsebenen. Deshalb erscheint es mir als durchaus sinnvoll, die Gewinnung und den Einsatz von Mustern als ein phasenübergreifendes Konzept zu betrachten.

Auf der untersten Ebene, der Code-Ebene, bieten Muster allgemeine Lösungen für gängige Programmierprobleme. Muster liegen dort typischerweise in Form von Bibliotheken mit Standard-Datentypen, wie Listen, etc. vor. Auf der nächsthöheren Ebene, der Implementierungsebene, beschreiben Muster bestimmte generelle Problemlösungen in einer bestimmten Programmiersprache. Zum Beispiel wie sich eine „enthält“-Beziehung (vgl. 6.2.3.1, S.47) mit einer Kardinalität von 1..N in C++ verwirklichen lässt. Auf dieser Ebene können Muster auch in Form von Code-Generatoren vorliegen.

Auf der nächsthöheren Ebene, der implementierungsabhängigen Entwurfsebene, bieten Muster Lösungsschemata für gängige implementierungsabhängige Entwurfsprobleme. Zum Beispiel wie sich auf einem Macintosh ein Aufklapp-Menü verwirklichen lässt. In dieser Ebene können Muster ebenfalls in Form von Code-Generatoren vorliegen. Auf der nächsthöheren Ebene, der implementierungsunabhängigen Entwurfsebene, bieten Muster Lösungsschemata für gängige implementierungsunabhängige Entwurfsprobleme. Zum Beispiel wie die Unabhängigkeit der Daten eines Programms von dessen Benutzeroberfläche erreicht werden kann. Eine Lösungsmöglichkeit dafür ist das Model-View-Controller-Konzept (vgl. 3.1, S. 15ff), das eines der bewährtesten Muster ist, die existieren.

Die implementierungsabhängige und die implementierungsunabhängige Entwurfsebene lassen sich meist jeweils in weitere Ebenen verschiedener Abstraktionsgrade untergliedern, für die auch wieder Muster existieren. Allgemein spricht man bei einem Muster auf einer Entwurfsebene von einem **Entwurfsmuster**.

Auf der höchsten Ebene, der Analyseebene, beschreiben Muster Lösungsschemata für Probleme aus einem Anwendungsbereich. Auf dieser Ebene werden Muster beispielsweise in Form von abstrakten Use-Cases umgesetzt (vgl. 6.2.1, S. 42, sowie Jacobson/Christerson, 1995).

Der Einsatz von Mustern bringt nicht nur den Vorteil von Problemlösungen mit sich, die sich bereits anderswo bewährt haben, sondern diese Problemlösungen sind in der Regel allgemeiner gehalten als es die spezifische Lösung zu einem bestimmten Problem ist. Durch die Verwirklichung von allgemeineren Lösungen wird dann auch die Flexibilität einer bestimmten Problemlösung erhöht.

Die Aufgabe eines Entwicklers besteht nun darin, in jeder Entwicklungsphase zu versuchen, sowohl auftretende Probleme mit Hilfe bereits existierender Muster zu lösen, als auch eigene Problemlösungen zu Mustern zu abstrahieren. Beispiele für die Gewinnung und den Einsatz von Mustern auf der implementierungsunabhängigen Entwurfs-ebene bei der Entwicklung von MUNOS werden in Kapitel 8 gegeben.

### 6.2.3 Die Notation

Mittlerweile haben sich die Notationen von Grady Booch und von James Rumbaugh als Quasi-Standards etabliert. Booch und Rumbaugh sind allerdings zur Zeit im Begriff, ihre beiden Methoden zu einer Methode zu fusionieren. Die daraus entstehende Notation wird wahrscheinlich der De-facto-Standard der Zukunft sein. In diesem Projekt wird die Notation von Booch verwendet (vgl. Booch, 1994). Das hat keine fachlichen Gründe, sondern ist eine reine Geschmacksfrage. Eventuell wird dann später eine Anpassung an die neue Notation erfolgen.

Bei einem Entwurf wird in dieser Methode mit zwei Arten von Diagrammen gearbeitet. Beziehungen und Strukturen, wie Vererbungshierarchien, werden durch **statische Diagramme** veranschaulicht. Dynamische Abläufe, wie die Kommunikation zwischen Objekten, werden mit Hilfe von **Interaktionsdiagrammen** dargestellt. Im folgenden wird zunächst darauf eingegangen, was in einem statischen Diagramm dargestellt werden kann. Daran anschließend werden die Interaktionsdiagramme besprochen.

### 6.2.3.1 Statische Diagramme

Ein Objekttyp wird durch eine „Wolke“ dargestellt, abstrakte Objekttypen werden zusätzlich durch ein „A“ gekennzeichnet. Zum Beispiel ist in Abbildung 6.1 eine Monarchie ein Objekttyp und in Abbildung 6.2 eine Staatsform ein abstrakter Objekttyp.

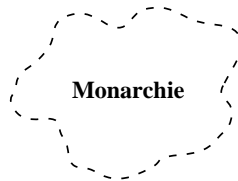


Abb. 6.1: Objekttyp

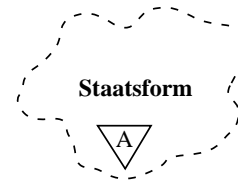


Abb. 6.2: abstrakter Objekttyp

Wenn ein Objekttyp A ein Subtyp eines Objekttyps B ist, so besteht zwischen A und B eine „ist ein“-Beziehung. Diese Beziehung wird durch einen Pfeil symbolisiert. Zum Beispiel ist in Abbildung 6.3 eine Monarchie eine Staatsform.

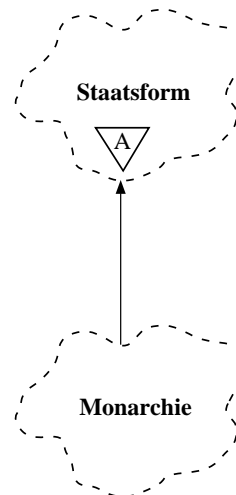


Abb. 6.3: Eine Monarchie ist eine Staatsform

Wenn ein Objekt A ein anderes Objekt B „enthält“, so ist B ein Teil von A und es muß daher möglich sein, von dem Objekt A aus, zu dem Objekt B zu gelangen. Die „enthält“-Beziehung wird durch eine Linie dargestellt, die an einem Ende einen ausgefüllten Kreis hat. Der ausgefüllte Kreis markiert den Objekttyp, der Objekte des anderen Objekttyps enthält. Dazu wird am anderen Ende der Linie die Kardinalität angegeben. Zum Beispiel enthält in Abbildung 6.4 eine Monarchie genau einen Monarchen.

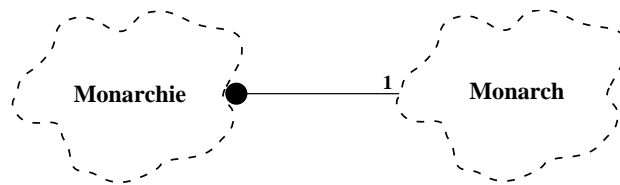


Abb. 6.4: Eine Monarchie enthält genau einen Monarchen

Wenn ein Objekt A ein anderes Objekt B nicht enthält, aber dennoch verwendet, so wird das verwendende Objekt mit einem nicht ausgefüllten Kreis markiert. In Abbildung 6.5 verwendet beispielsweise ein Monarch beliebig viele Berater.

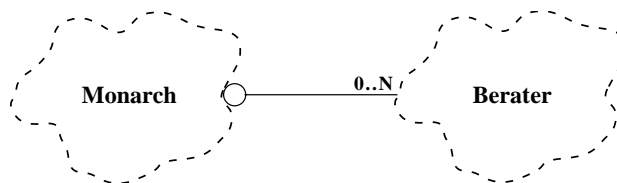


Abb. 6.5: Ein Monarch verwendet beliebig viele Berater

Ein System bzw. ein Subsystem besteht aus einer Menge von Objekttypen und/oder anderen Subsystemen, die in einem funktionalen Zusammenhang stehen. Ein System bzw. ein Subsystem wird von einem schattierten Rechteck mit runden Ecken umschlossen. Zum Beispiel zeigt die Abbildung 6.6 ein System, den Staat, das aus einem Objekttyp, dem Monarchen, und einem Subsystem, dem gemeinen Volk, besteht.

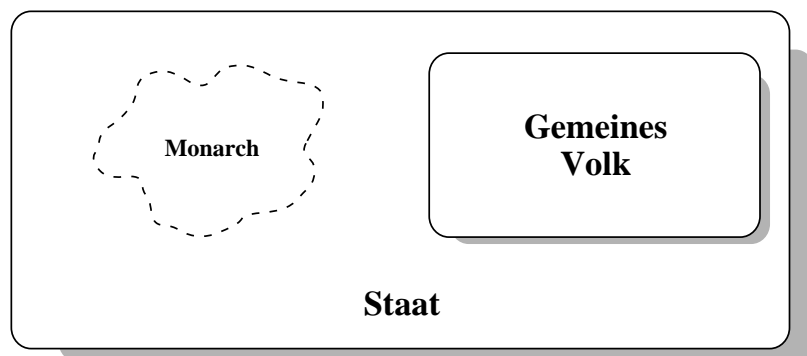


Abb. 6.6: Ein Staatssystem, das aus einem Monarchen und dem gemeinen Volk besteht

Booch verwendet für Subsysteme auf der Entwurfsebene eigentlich Kategorien (vgl. Booch, 1994), die aber streng genommen etwas anderes sind als Subsysteme. Eine Kategorie ist eine Menge von Objekttypen bzw. Operationen und/oder Kategorien, die in einem logischen Zusammenhang stehen. Das können beispielsweise alle Interface-Objekttypen oder alle Zugriffsoperationen auf Attribute sein. Es sollte besser zwischen den beiden Termini unterschieden werden, denn ein Subsystem läßt sich immer als ein solches verwirklichen, eine Kategorie nicht unbedingt. Kategorien werden bislang beim Entwurf von MUNOS nicht verwendet. Um dennoch die Möglichkeit dafür offen zu halten, werden bereits auf der Entwurfsebene Subsystem-Diagramme eingesetzt.

**6.2.3.2 Interaktionsdiagramme**

Ein Beispiel für ein Interaktionsdiagramm zeigt die Abbildung 6.7. Am linken Rand eines Interaktionsdiagramms steht normalerweise eine Beschreibung der Vorgänge, die in dem Interaktionsdiagramm ablaufen. Im Falle des Beispiels aus Abbildung 6.7 werden die Vorgänge allerdings im Fließtext beschrieben.

Die vertikale Achse eines Interaktionsdiagramms ist die Zeitachse, d.h. von der Reihenfolge der Abläufe her, wird ein Interaktionsdiagramm von oben nach unten gelesen. Der zeitliche Ablauf wird darin allerdings nicht maßstabsgetreu wiedergegeben. Prinzipiell ist es auch möglich, damit parallele Abläufe darzustellen. Die Aktivitätsphasen der Objekte werden jeweils durch vertikale Balken angedeutet.

Die horizontale Achse eines Interaktionsdiagramms ist die Nachrichten-Achse. Die Nachrichten der Objekte werden dort durch Pfeile symbolisiert. Über einem Nachrichtenpfeil steht dann jeweils die Nachricht, die gesendet wird. Wenn bei einer Nachricht eine Rückgabe erwartet wird, so wird diese Tatsache durch einen kleinen Pfeil unter dem Nachrichtenpfeil symbolisiert. Unter dem Rückgabepfeil steht dann jeweils, was als Rückgabe erwartet wird. Bei einer Rückgabe kann es sich entweder um ein Objekt oder um einen Wert handeln.

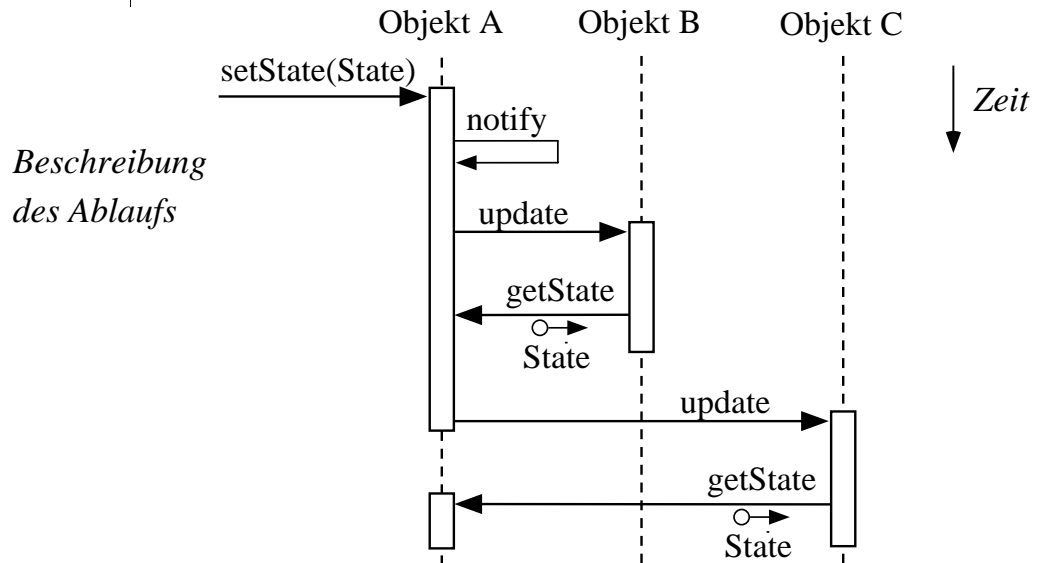


Abb. 6.7: Interaktionsdiagramm eines Abhängigkeitsmechanismus

Abbildung 6.7 zeigt zum Beispiel das Interaktionsdiagramm eines Abhängigkeitsmechanismus, wie er häufig in Model-View-Controller-Kontexten vorkommt (vgl. 3.1, S. 15ff). In diesem Beispiel gehören die Objekte B und C zu einer View und Objekt A ist Teil eines Modells. Daher sind die Objekte B und C von Änderungen am Objekt A abhängig. Wenn nun der Zustand des Objekts A geändert wird, so schickt das Objekt A an sich selbst die Nachricht, alle von ihm abhängigen Objekte eine Aktualisierungsaufforderung zu schicken. Daraufhin fordert das Objekt A, mittels einer Nachricht, das Objekt B auf, sich zu aktualisieren. Anschließend aktualisiert sich das Objekt B, indem es sich von Objekt A den aktuellen Zustand des Objekts A zurückgeben lässt. Analog dazu erhält das Objekt C vom Objekt A eine Aktualisierungsaufforderung und bringt sich in Folge dessen auf den aktuellen Stand. Dazu gibt es zahlreiche Varianten, die von der bloßen Aktualisierungsaufforderung, wie es in dem Beispiel der Fall ist, bis hin zur direkten Übergabe des geänderten Werts reichen.

Der Rückgabepfeil wird in der Booch-Notation eigentlich bei einer anderen Variante dynamischer Diagramme verwendet (vgl. Booch, 1994). Bei Interaktionsdiagrammen ist dieser nicht üblich. Der Grund für zwei alternative Möglichkeiten zur Darstellung dynamischer Abläufe in der Booch-Notation liegt in der erst später erfolgten Hereinnahme von Interaktionsdiagrammen. Über diese Art der Darstellung dynamischer Abläufe scheint man sich immer mehr einig zu sein. In der älteren Variante dynamischer Diagramme gibt es keine Zeitachse, sondern die Reihenfolge der Nachrichten geht lediglich aus deren Numerierungen hervor. Der einzige Vorteil der älteren Variante schien mir der Rückgabepfeil zu sein.

## 6.3 Tätigkeiten und Ergebnisse der einzelnen Phasen

Eine wichtige Rolle bei der Entwicklung von MUNOS spielen Use-Cases. Sie dienen anfangs als Ausgangsbasis für die Spezifikation und sie bilden letztendlich auch den Maßstab dafür, ob alles richtig entwickelt wurde. So verkörpern Use-Cases von der Analysephase bis hin zur Implementierungsphase den „roten Faden“ durch alle dazwischenliegenden Entwicklungsphasen. Nun werden die Tätigkeiten und Ergebnisse der einzelnen Entwicklungsphasen in der Reihenfolge ihres Auftretens besprochen.

### 6.3.1 Strategisches Modellieren

Das strategische Modellieren beschränkt sich in dieser ersten Iteration bislang darauf, einen Projektplan für den Zeitraum der Diplomarbeit zu erstellen. Es sollen noch weitere Projektpläne folgen, doch dazu wird in Kapitel 10 mehr berichtet.

### 6.3.2 Analytisches Modellieren

Das Ergebnis des analytisches Modellierens ist in diesem Projekt die Spezifikation, die im wesentlichen nach den entsprechenden ANSI/IEEE-Richtlinien von 1984 (vgl. dort) aufgebaut ist. Wo es sinnvoll erschien, wurde eine Anpassung an die Gegebenheiten bei MUNOS vorgenommen.

Um beispielsweise unnötigen Ballast zu vermeiden, wurde eine Kategorie von Vorgaben dann weggelassen, wenn für MUNOS keine Vorgaben dieser Kategorie existieren und die Aussage des Nichtexistierens dieser Vorgaben, auch nicht von Bedeutung ist.

Die Rolle eines funktionalen Modells wird in der Spezifikation von einem Use-Case-Modell übernommen. Dabei werden Muster in Form von abstrakten Use-Cases realisiert (vgl. 6.2.2, S. 45). Das Use-Case-Modell enthält neben den Beschreibungen der einzelnen Use-Cases auch die Regeln und deren Ausnahmen, die in den einzelnen Use-Cases zur Anwendung kommen. Außerdem wird das Use-Case-Modell durch mehrere Übersichten ergänzt, die einerseits verschiedene Zusammenhänge verdeutlichen und andererseits als Orientierungshilfe dienen sollen.

### 6.3.3 Entwurfsbezogenes Modellieren

Das Ergebnis des entwurfsbezogenen Modellierens ist in diesem Projekt ein implementierungsunabhängiger Entwurf. Dazu werden die Use-Cases der Spezifikation mittels der Booch-Notation modelliert und Entwurfsmuster eingesetzt, wenn dies sinnvoll erscheint. Außerdem wird versucht Use-Case-spezifische Problemlösungen zu Entwurfsmustern zu abstrahieren. Beim Modellieren werden die Bezeichner jeweils in Englisch abgefaßt, da MUNOS international zur Verfügung gestellt werden soll. Ein Entwurf besteht dann aus statischen Diagrammen und Interaktionsdiagrammen, die durch Beschreibungen der Objekttypen, der Operationen und der Beziehungstypen ergänzt werden.

### 6.3.4 Implementierungsbezogenes Modellieren

Das Ergebnis des implementierungsbezogenen Modellierens ist in diesem Projekt ein implementierungsabhängiger Entwurf. Dazu wird der implementierungsunabhängige Entwurf um implementierungsabhängige Details ergänzt bzw. notfalls an implementierungsabhängige Gegebenheiten angepaßt. Das Ergebnis ist ein implementierungsabhängiger Entwurf. Darin wird die Booch-Notation weiterverwendet und der Aufbau des implementierungsunabhängigen Entwurfs wird übernommen. Auch der Umgang mit Entwurfsmustern verläuft analog zum entwurfsbezogenen Modellieren.

Im Verlauf der Diplomarbeit wurde auf das implementierungsbezogene Modellieren verzichtet, da die Unterschiede der beiden Entwürfe nur minimal ausgefallen wären. Der einzige plattformabhängige Teil in MUNOS ist die Graphik, die wiederum auf Apples QuickDraw GX basiert (vgl. Apple, 1994). Es wird daher nur eine minimale Anpassung an QuickDraw GX notwendig. QuickDraw GX ist zwar von der Konzeption her ebenfalls plattformunabhängig, steht aber bisher nur auf Apple-Rechnern zur Verfügung.

### 6.3.5 Implementierung

Geplant sind bislang zwei Implementierungen in C++. Eine für Macintosh-Rechner ab System 7 und eine für Windows-Umgebungen. Das sind die beiden Plattformen, die sich im Musikbereich am meisten durchgesetzt haben.

Bei der Implementierung wird versucht, die Beziehungen zwischen den Objekttypen und die Kommunikation zwischen den Objekten aus dem Entwurf mit Hilfe dafür gängiger Implementierungsmuster zu realisieren (vgl. 6.2.2, S. 44, sowie Gamma et al., 1995 und Papurt, 1995). Außerdem werden auf Code-Ebene für Standard-Datentypen entsprechende Bibliotheken eingesetzt (vgl. 6.2.2, S.44). Die Bezeichner und die Kommentare werden jeweils in Englisch abgefaßt. Als Testmodell werden die Use-Case-Beschreibungen aus der Spezifikation herangezogen.

### **6.3.6 Vertrieb**

MUNOS soll über Internet frei verfügbar gemacht werden.

## **6.4 Das Lebenszyklus-Modell**

Es folgt nun eine Charakterisierung dieses Lebenszyklus-Modells anhand der vier Strategiearten, die in 5.3.3 (vgl. S. 36ff) besprochen wurden. Dazu wird für jede der vier Strategiearten angegeben, wo welche Strategien zur Anwendung kommen und warum gerade diese Strategien ausgewählt wurden.

### **6.4.1 Die Progressionsstrategien**

Um einen möglichst reibungslosen Übergang zwischen den einzelnen Entwicklungsphasen zu bekommen, ist eigentlich immer eine additive Progressionsstrategie erstrebenswert. Allerdings lassen sich Transformationen in den seltensten Fällen ganz vermeiden, denn spätestens, wenn es zur Implementierungsphase kommt, findet meist eine Transformation des Entwurfs in Programm-Code statt. Auch in dieser Methode kommen beide Progressionsstrategien zum Einsatz.

Vom strategischen Modellieren zum analytischen Modellieren gibt es bei diesem Projekt keine direkte Verbindung. Der Projektplan sagt zwar etwas darüber aus, wie beispielsweise der Entwurf ausgeführt werden soll, hat aber keinen inhaltlichen Bezug zu diesem. Im Falle von MUNOS ist es für die eigentliche Problemlösung auch nicht notwendig, das Anwendungs-Umfeld zu modellieren, was bei Entwicklungen für Firmen durchaus vorkommen kann.

Vom analytischen Modellieren zum entwurfsbezogenen Modellieren ist eine Transformation von Nöten. Diese Transformation kann nicht maschinell vorgenommen werden, denn einerseits sind die Use-Case Beschreibungen in der Spezifikation dazu nicht formal genug und andererseits würde das zu sogenannten „naiven“ Entwürfen führen, die vor allem auf die Problemlösung ausgelegt sind, ohne die spätere Wartbarkeit der Software und die Wiederverwendbarkeit einzelner Teile zu berücksichtigen. Eine interessante Frage wäre in diesem Zusammenhang, ob und in wie weit sich die Gewinnung bzw. der Einsatz von Entwurfsmustern automatisieren lässt. Zum Beispiel könnte versucht werden, die dazu notwendigen Informationen aus dem Entwurf, den Entwurfsmustern und eventuell aus den Use-Cases jeweils mit Hilfe einer formalen Semantik zu modellieren.

Der Übergang vom entwurfsbezogenen Modellieren zum implementierungsabhängigen Modellieren ist der einzig machbare additive Übergang, denn dort kann die Notation beibehalten werden. Ein Kriterium, um diese Phasen noch auseinanderhalten zu können, sind die implementierungsspezifischen Details, die im entwurfsbezogenen Modell noch nicht enthalten sein sollten.

Vom implementierungsbezogenen Modellieren zur Implementierung ist wieder eine Transformation von Nöten. Für diese Transformation wäre eine maschinelle Lösung durchaus denkbar. Beispielsweise könnte ein Code-Generator die Beziehungen zwischen den Objekttypen und die Kommunikation zwischen den Objekten aus dem Entwurf mit Hilfe dafür gängiger Implementierungsmuster realisieren (vgl. 6.2.2, S. 44, sowie Gamma et al., 1995 und Papurt, 1995).

## 6.4.2 Die Iterationsstrategie

Die Entwicklung von MUNOS ist nach der inkrementellen Iterationsstrategie aufgebaut, die sich mit ihren relativ kurzen Iterationen am besten für Projekte eignet, in denen in relativ kurzen Zeiträumen möglichst gute Teilergebnisse verwirklicht werden sollen.

Nach der Diplomarbeit soll im Laufe der Zeit der Funktionsumfang von MUNOS erweitert werden. Mehr über die geplante Zukunft von MUNOS wird in Kapitel 10 berichtet.

### 6.4.3 Die Verpackungsstrategie

Obwohl die Methode eine auf das Projekt zugeschnittene Methode ist, kann man von einer flexiblen Phasenzuordnung sprechen. Die Tätigkeiten werden jeweils so auf die einzelnen Phasen verteilt, wie dies für das Projekt am günstigsten erscheint. Der Grund für die Anwendung dieser Strategie, ergibt sich aus der Strategie selbst.

### 6.4.4 Die Meilensteinstrategien

In diesem Projekt kommen zwei Meilensteinstrategien zum Einsatz. Einerseits ist das die Management-Review Strategie und andererseits die Risiko-gesteuerte Strategie. In die Planung der Weiterführung eines Projekts sowohl die Qualität der bisherigen Ergebnisse als auch eventuell auftretende Risiken einfließen zu lassen, scheint eine gute Kombination zu sein. Auch die Planung der Diplomarbeit wurde nach den Prinzipien dieser beiden Strategien jeweils an die aktuellen Gegebenheiten angepaßt. Beispiele hierfür finden sich in Kapitel 10, in dem unter anderem auch auf den Verlauf der Arbeit eingegangen wird.

## 7 Das Use-Case-Konzept an einem Beispiel aus der Entwicklung von MUNOS

In diesem Kapitel wird an dem Use-Case zur Erzeugung eines Liniensystems gezeigt, wie eine Use-Case in der Spezifikation von MUNOS beschrieben wird und wie diese Beschreibung sich dann im Entwurf widerspiegelt. Dazu werden Auszüge aus der Spezifikation und aus dem Entwurf von MUNOS verwendet.

Jeder Use-Case aus der Spezifikation von MUNOS wurde im Entwurf in Form von statischen Diagrammen und Interaktionsdiagrammen modelliert (vgl. 6.2.3, S. 45ff). Zum Beispiel resultierten aus dem Use-Case zur Erzeugung von Liniensystemen insgesamt ein statisches Diagramm und vier Interaktionsdiagramme. Dabei bilden die Interaktionsdiagramme den „roten Faden“ von der Spezifikation zum Entwurf, denn in den Interaktionsdiagrammen werden jeweils die Abläufe modelliert, die in den Use-Case-Beschreibungen beschrieben werden. Wenn man von einer Use-Case-Beschreibung der Spezifikation kommt, sollte man daher im Entwurf zuerst die Interaktionsdiagramme zu diesem Use-Case lesen. Parallel dazu können aus den zugehörigen statischen Diagrammen die Beziehungen entnommen werden, die zwischen den Objekten bestehen, die in den Interaktionsdiagrammen vorkommen. Der Übersichtlichkeit halber werden in den statischen Diagrammen der Use-Cases nicht alle Beziehungen gezeigt, sondern nur die, die für die Interaktionsdiagramme des jeweiligen Use-Case relevant sind. Die statischen Diagramme zu den Use-Cases sind außerdem „eingeebnet“, d.h. in ihnen werden auch ererbte Beziehungen gezeigt.

Um einen Überblick zu ermöglichen, welche Beziehungen eigentlich woher rühren, d.h. von welchem Objekttyp diese eventuell erbt wurden, und um größere Zusammenhänge zu verdeutlichen, existieren im Entwurf statische Diagramme zu den einzelnen Subsystemen, in die MUNOS untergliedert ist. Da es in diesem Kapitel primär um den „roten Faden“ der Use-Cases geht, enthält es kein Beispiel für ein Subsystem-Diagramm. Beispiele dafür finden sich aber im folgenden Kapitel, bei den Entwurfsmustern.

Die statischen Diagramme und die Interaktionsdiagramme des Entwurfs werden im Entwurf durch die Beschreibungen der in den Diagrammen vorkommenden Objekttypen und deren Operationen ergänzt. Dieses Kapitel enthält die Beschreibungen des abstrakten Objekttyps für Liniensysteme und des Objekttyps für herkömmliche 5-Linien-Systeme, sowie die Beschreibung einer Operation des abstrakten Objekttyps für Liniensysteme.

Da im Entwurf die Bezeichner in Englisch abgefaßt sind (vgl. 6.3.3, S. 53), werden nun zunächst, in alphabetischer Reihenfolge, deren Übersetzungen angegeben:

*DefaultLayout*

Das Standard-Layout.

*DefaultLayoutManager*

Der Standard-Layout-Manager.

*DefaultSubject*

Das Standard-Subjekt. Ein „Subjekt“ ist ein Objekt, das von anderen Objekten beobachtet wird.

*FiveLinesStaff*

Ein herkömmliches 5-Linien-System.

*RestRepresentation*

Die Repräsentation einer Pause.

*ToneRepresentation*

Die Repräsentation eines Tones.

*Staff*

Ein beliebiges Liniensystem.

## 7.1 Der Use-Case zur Erzeugung eines Liniensystems in der Spezifikation

### 7.1.1 Aufbau der Beschreibung eines Use-Case

Der Aufbau der Use-Case Beschreibungen, der in der Spezifikation von MUNOS zur Anwendung kommt, basiert auf den Aufbauten der Use-Case-Beschreibungen von Jacobson (vgl. Jacobson et al., 1994) und von Rumbaugh (vgl. Rumbaugh, 1995). Der Aufbau wurde außerdem um die Punkte *Notwendigkeit*, *änderungsanfällig* und *Regeln* ergänzt, wobei die Grade an Notwendigkeit aus dem ANSI/IEEE Standard für Spezifikationen entnommen sind (vgl. ANSI/IEEE, 1984). Es folgt nun, in einem Auszug aus der Spezifikation von MUNOS, eine Beschreibung des Aufbaus einer Use-Case-Beschreibung.

*Auszug aus der Spezifikation von MUNOS*

<b>Use-Case:</b>	Eindeutige Bezeichnung des Use-Case.
<b>Notwendigkeit:</b>	Ein Use-Case besitzt einen der folgenden drei Grade an Notwendigkeit: <ul style="list-style-type: none"> <li>• <i>verbindlich</i>: Der Use-Case muß verwirklicht werden.</li> <li>• <i>wünschenswert</i>: Der Use-Case würde in den eigentlichen Anwendungsbereich der Software passen und damit deren Funktionalität konsequent erweitern. Die Software kann aber auch ohne dessen Verwirklichung akzeptiert werden.</li> <li>• <i>optional</i>: Der Use-Case paßt nicht in den eigentlichen Anwendungsbereich der Software, würde diesen aber evtl. sinnvoll ergänzen. Ein Use-Case dieser Art kann am ehesten weggelassen werden.</li> </ul>
<b>verwendet:</b>	Auflistung der Abläufe von Use-Cases, die von diesem Use-Case verwendet werden. Dabei wird in der Beschreibung des Use-Case für jeden verwendeten Use-Case aufgeführt, wann dieser verwendete Use-Case zur Anwendung kommt.

*(wird auf der nächsten Seite fortgesetzt)*

*Auszug aus der Spezifikation von MUNOS (Fortsetzung)*

<b>ergänzt:</b>	Auflistung der Abläufe von Use-Cases, die dieser Use-Case ergänzt. Dabei wird in der Beschreibung des ergänzenden Use-Case für jeden ergänzten Use-Case aufgeführt, wann dieser ergänzende Use-Case zur Anwendung kommt und von wo bis wo dieser in den jeweiligen ergänzten Use-Case eingefügt wird.
<b>Zusammenfassung:</b>	Zusammenfassung des Use-Case.
<b>Akteure:</b>	Akteure, die den Use-Case durchführen können.
<b>Vorbedingungen:</b>	Der Use-Case wird nur dann durchgeführt, wenn diese Bedingungen erfüllt sind.
<b>Nachbedingungen:</b>	Bedingungen, die nach erfolgreicher Durchführung des Use-Case garantiert erfüllt sind.
<b>Im Fehlerfall:</b>	Fehlerbehandlung, d.h. Beschreibung dessen, was passiert, wenn während der Durchführung des Use-Case ein Fehler auftritt.
<b>änderungsanfällig:</b>	Gibt darüber Auskunft, ob im Lauf der Zeit Änderungen am Use-Case eher wahrscheinlich sind oder ob er eher stabil bleibt. Wenn Änderungen zu erwarten sind, wird angegeben, für welche Teile des Use-Case diese erwartet werden und wie diese aussehen könnten.
<b>Regeln:</b>	Auflistung der Regeln aus dem Anwendungsbereich, die, zusammen mit deren Ausnahmen, in dem Use-Case zur Anwendung kommen.
<b>Normaler Ablauf:</b>	Beschreibung des Ablaufs, der für das Verständnis des Use-Case am besten geeignet ist.
<b>Varianten:</b>	Beschreibung von Varianten des normalen Ablaufs, sowie von Abläufen, die zu dem normalen Ablauf verwandt sind.

## 7.1.2 Die Beschreibung des Use-Case zur Erzeugung eines Liniensystems

Es folgt nun, wieder in einem Auszug aus der Spezifikation von MUNOS, die vollständige Beschreibung des Use-Case zur Erzeugung eines Liniensystems.

*Auszug aus der Spezifikation von MUNOS*

<b>Use-Case 2:</b>	Repräsentation eines graphischen Bezugssystems für Noten erstellen.
<b>Notwendigkeit:</b>	Verbindlich.
<b>verwendet:</b>	<p><i>Im Falle des Löschens:</i></p> <ul style="list-style-type: none"> <li>• Use-Case 4: Ein Pausenzeichen löschen</li> <li>• Use-Case 5: Eine Note löschen</li> <li>• Use-Case 7: Ein Layout löschen</li> </ul>
<b>ergänzt:</b>	–
<b>Zusammenfassung:</b>	Es wird die Repräsentation eines graphischen Bezugssystems für Noten in Form eines Liniensystems erstellt.
<b>Akteure:</b>	Programm, in das die Bibliothek eingebunden wurde.
<b>Vorbedingungen:</b>	<p><i>Für den normalen Ablauf :</i></p> <ul style="list-style-type: none"> <li>• Der Layout-Manager, zu dem das Liniensystem gehören soll, muß bereits existieren.</li> </ul>

*(wird auf der nächsten Seite fortgesetzt)*

*Auszug aus der Spezifikation von MUNOS (Fortsetzung)*

**Nachbedingungen:** *Für den normalen Ablauf :*

- Es wurde die Repräsentation eines Bezugssystems für Noten in Form eines Liniensystems erstellt.
- An das Programm wurde einen Verweis auf das Liniensystem zurückgegeben.

*Im Falle von Änderungen:*

- Die Änderungen wurden korrekt und vollständig vorgenommen.

*Im Falle des Löschens:*

- Das Liniensystem wurde, mit allen Konsequenzen, erfolgreich gelöscht.

**Im Fehlerfall:**

Der Use-Case wird abgebrochen. Das Programm erhält eine Fehlermeldung zurück, in der der Fehler näher spezifiziert wird. Dabei sollen folgende Fehlerarten berücksichtigt werden:

- Eine oder mehrere Vorbedingungen waren nicht erfüllt.
- Die Repräsentation konnte aus einem bekannten Grund nicht erstellt werden bzw. eine Änderung oder das Löschen konnte aus einem bekannten Grund nicht vorgenommen werden.
- Die Repräsentation konnte aus unbekanntem Gründen nicht erstellt werden bzw. eine Änderung oder das Löschen konnte aus unbekanntem Gründen nicht vorgenommen werden.

*(wird auf der nächsten Seite fortgesetzt)*

*Auszug aus der Spezifikation von MUNOS (Fortsetzung)*

- änderungsanfällig:**
- Es werden im Lauf der Zeit noch mehr Arten von Liniensystemen zu dem hier verwendeten 5-Linien-System hinzukommen, wie z.B. verschiedene Tabulatursysteme für Saiteninstrumente. Dann wird es z.B. nötig sein, die Anzahl der Linien anzugeben, die das System besitzen soll. Bei der Eingabe der Schlüssel muß dann auch eine Interpretation für das Liniensystem angegeben werden, d.h. auf welche Linie sich der Schlüssel bezieht oder wo dieser, vertikal gesehen, im System geschrieben werden soll. an der Position des Schlüssels. Auch wird in diesem Fall das Problem zu lösen sein, wo im System, wiederum vertikal gesehen, die Pausen notiert werden sollen.
  - Es werden noch mehr Arten von Schlüsseln oder Ersatzbezeichnungen hinzukommen, wie zum Beispiel die Tabulatur-Bezeichnung.

**Regeln:** –

**Normaler Ablauf:** Das Programm beauftragt die Bibliothek die Repräsentation eines graphischen Bezugssystems für Noten in Form eines Liniensystems zu erstellen. Dazu sind folgende Angaben notwendig:

- ein Verweis auf den Layout-Manager, zu dem das Liniensystem gehören soll
- die Anordnungsnummer, die besagt, als wievielttes System das System in einer Zeile stehen soll.

Folgende Angaben können optional gemacht werden:

- Informationen über die Tonart,
- Informationen über die Taktart
- Informationen über den Notenschlüssel
- Informationen über Tonartwechsel,
- Informationen über Taktartwechsel
- Informationen über Schlüsselwechsel

*(wird auf der nächsten Seite fortgesetzt)*

**Auszug aus der Spezifikation von MUNOS (Fortsetzung)**

Die Informationen über eine Tonart bestehen in Anzahl, Art Reihenfolge und Positionen der Akzidentien. Da die Anzahl der Akzidentien kann auch null sein kann wird zwischen keiner Angabe zur Tonart und keinen Akzidentien unterschieden.

Die Informationen über einen Tonartwechsel bestehen in den Informationen über die neue Tonart und in der Angabe der Taktnummer, ab der diese gültig ist.

Die Informationen über eine Taktart bestehen aus keiner, einer oder mehreren Taktangaben in Form von Brüchen. Eine Taktangabe in Form eines Bruches besteht aus einem Zähler und einem Nenner, dem Taktmaß. Wenn es sich um einen 4/4 bzw. um einen 2/2 Takt handelt, kann zusätzlich angegeben werden, ob diese in Zahlen oder mit ihren jeweiligen Symbolen notiert werden sollen. Im Falle von mehreren Brüchen muß dazu angegeben werden, ob diese alternieren oder ob es sich um einen zusammengesetzten Takt handelt und in welcher Reihenfolge diese aufeinander folgen.

Die Informationen über einen Taktartwechsel bestehen aus den Informationen über die neue Taktart und aus der Taktnummer, ab der diese gültig ist.

Zu jeder Taktangabe kann optional angegeben werden, in welcher Taktaufteilung diese notiert werden soll. Die Angabe der Taktaufteilung erfolgt in ganzen Zahlen und bezieht sich dabei auf das Taktmaß in der Taktangabe.

Die Information über einen Notenschlüssel besteht in der Art des Notenschlüssels.

Die Informationen über einen Schlüsselwechsel bestehen aus der Art des neuen Schlüssel und aus der Taktnummer, sowie der Zählzeit im Takt, ab der dieser gültig ist. Die Zählzeit wird als Bruch angegeben, dessen Nenner ein Notenwert ist und dessen Zähler besagt, wann, auf diesen Notenwert bezogen, der Schlüsselwechsel im Takt erfolgt.

Das Liniensystem meldet sich beim Layout-Manager an.

Alle gegebenen Informationen werden bei dem Liniensystem festgehalten. Schließlich wird dem Programm ein Verweis auf das Liniensystem zurückgegeben.

*(wird auf der nächsten Seite fortgesetzt)*

*Auszug aus der Spezifikation von MUNOS (Fortsetzung)***Varianten:*****Änderungen an einem Liniensystem vornehmen***

Die Angabe des Layout-Managers entfällt und, sollte die Anordnungsnummer nicht geändert werden, entfällt deren Angabe ebenfalls.

Folgende Änderungen sind möglich:

- Tonart ändern
- Tonartwechsel ändern
- Tonartwechsel hinzufügen
- Tonartwechsel entfernen

Dasselbe gilt analog für die Taktarten und Notenschlüssel.

Außerdem kann die Anordnungsnummer geändert werden.

Der Layout-Manager wird jeweils über eine Änderung informiert. Der wiederum informiert betroffene Layouts darüber.

Bei Änderungen im Zusammenhang mit der Taktart kann es nötig werden, daß die Repräsentationen betroffener Töne bzw. Pausen geändert werden müssen, d.h. das Notenbild kann sich ändern.

***Ein Liniensystem löschen***

Ein Liniensystem kann wieder gelöscht werden. Die Angaben des Layout-Managers und der Anordnungsnummer entfallen. Der Layout-Manager wird darüber benachrichtigt, daß das Liniensystem gelöscht wurde. Der Layout-Manager veranlaßt dann, daß sämtliche Layouts, die sich nur auf dieses Liniensystem beziehen, unter Anwendung von Use-Case 7, gelöscht werden und informiert andere betroffene Layouts darüber, daß eine Änderung stattgefunden hat. Mit dem Liniensystem werden unter Anwendung der Use-Cases 4 und 5 alle Repräsentationen der Töne und Pausen gelöscht, zu denen die Noten und Pausenzeichen gehören, für die das zu löschende Liniensystem das einzige Liniensystem ist, in dem sie dargestellt werden sollen.

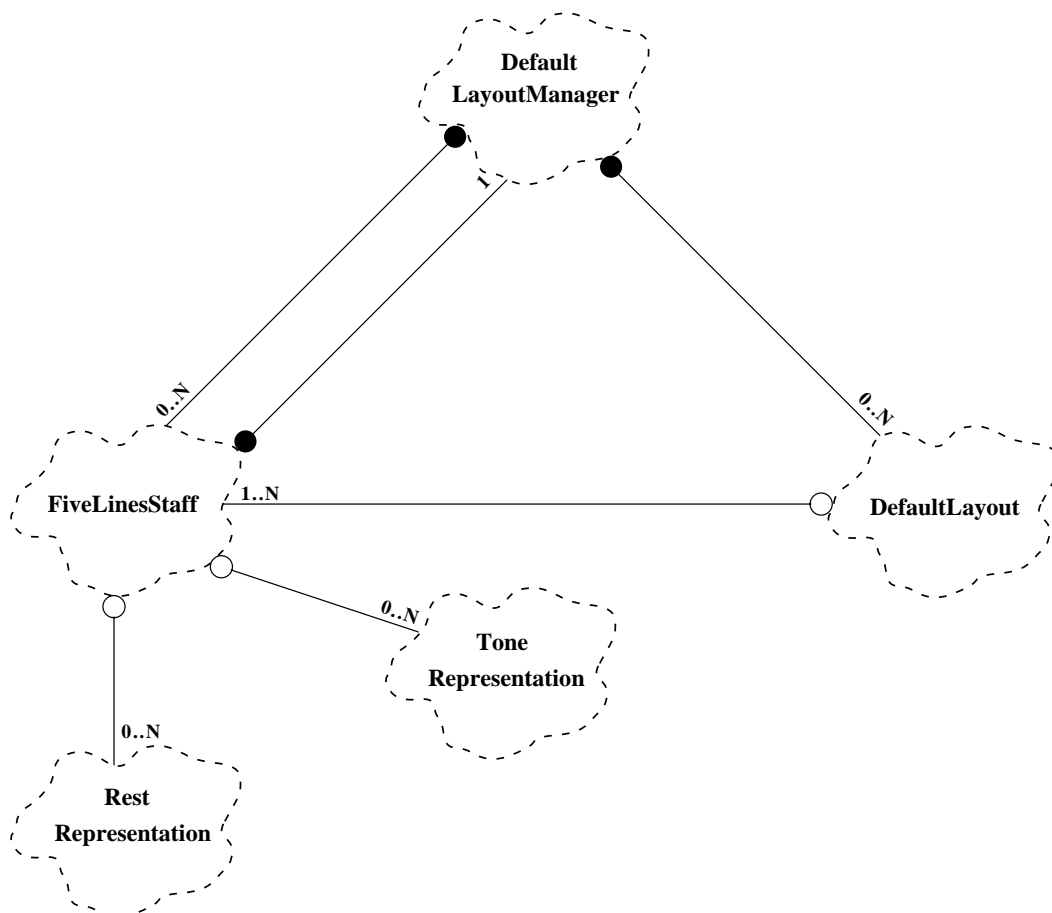
## 7.2 Der Use-Case zur Erzeugung eines Liniensystems im Entwurf

### 7.2.1 Das statische Diagramm

Aus dem in 7.1.2 beschriebenen Use-Case zur Erzeugung eines Liniensystems resultiert ein statisches Diagramm, das als Ergänzung zu den Interaktionsdiagrammen des Use-Case dient und daher parallel zu diesen gelesen werden sollte. Es folgt nun, in einem Auszug aus dem Entwurf von MUNOS, dieses statische Diagramm. Die Interaktionsdiagramme dazu finden sich dann auf den nächsten beiden Seiten.

*Auszug aus dem Entwurf von MUNOS*

**Statisches Diagramm 4.2.1**

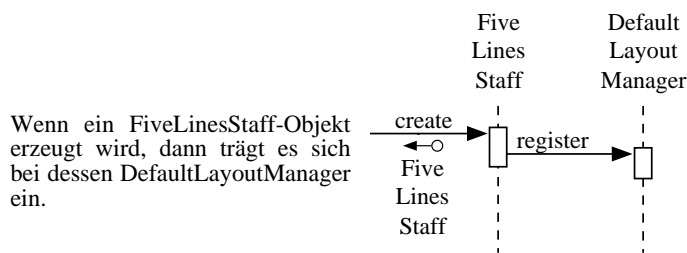


## 7.2.2 Die Interaktionsdiagramme

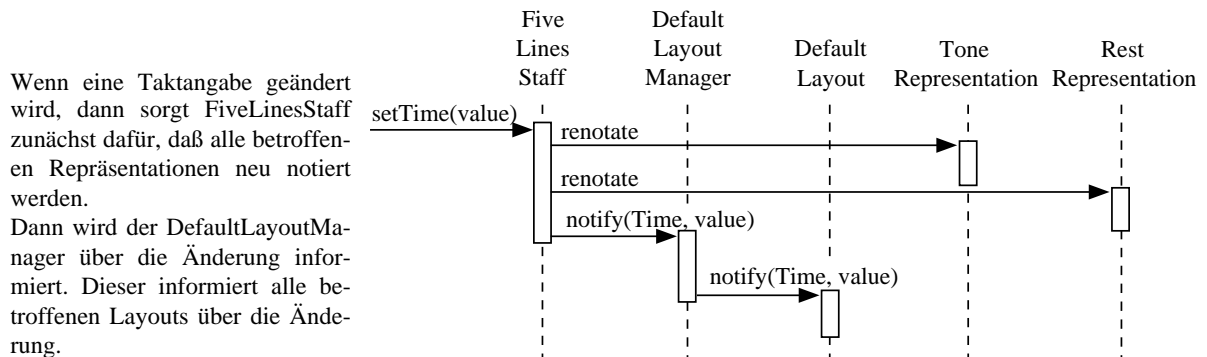
Aus dem in 7.1.2 beschriebenen Use-Case zur Erzeugung eines Liniensystems resultieren vier Interaktionsdiagramme. Es folgen nun, in einem Auszug aus dem Entwurf von MUNOS, diese vier Interaktionsdiagramme. Da die jeweiligen Abläufe direkt in den Interaktionsdiagrammen beschrieben werden, sind die Interaktionsdiagramme selbst erklärend. Die Beziehungen zwischen den Objekten lassen sich jeweils aus dem dazugehörigen statischen Diagramm (vgl. 7.2.1, S. 65) entnehmen.

*Auszug aus dem Entwurf von MUNOS*

### Interaktionsdiagramm 4.2.1 für den normalen Ablauf



### Interaktionsdiagramm 4.2.2 im Falle von Änderungen der Taktangaben

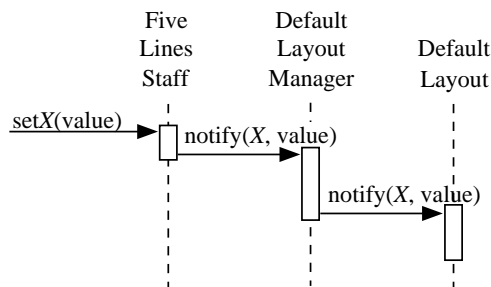


*(wird auf der nächsten Seite fortgesetzt)*

Auszug aus dem Entwurf von MUNOS (Fortsetzung)

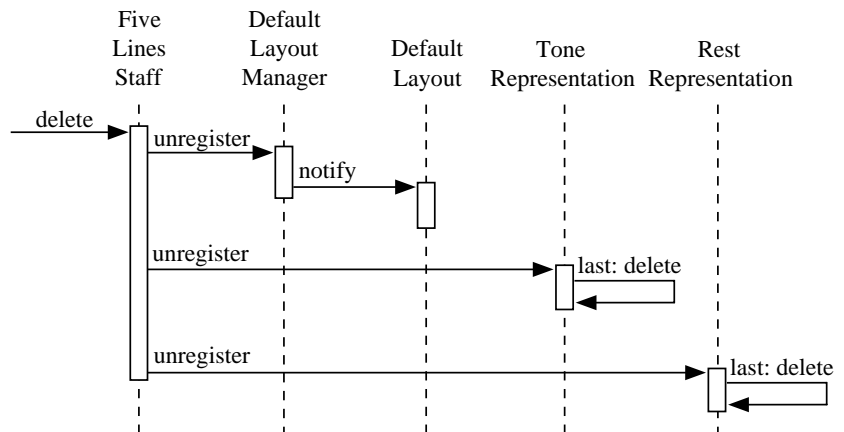
**Interaktionsdiagramm 4.2.3 für sonstige Änderungen**

Wenn in FiveLinesStaff das Attribut X geändert wird, dann wird der DefaultLayoutManager über die Änderung informiert. Der DefaultLayoutManager informiert die betroffenen DefaultLayouts über die Änderung.



**Interaktionsdiagramm 4.2.4 den Fall des Löschens**

Wenn ein FiveLinesStaff-Objekt gelöscht wird, dann meldet sich dieses bei dessen DefaultLayoutManager ab. Dieser informiert alle betroffenen Layouts über die Änderung. Dann meldet sich FiveLinesStaff bei sämtlichen betroffenen Repräsentationen ab. Wenn dies das letzte FiveLinesStaff-Objekt einer Repräsentation war, so löscht diese sich selbst und damit sämtliche Noten bzw. Pausenzeichen, die zu ihr gehören.



### 7.2.3 Aufbau der Beschreibung eines Objekttyps

Obwohl in vielen Methoden mit der Angabe von Attributen gearbeitet wird, ist dies eigentlich nicht notwendig, denn ein Attribut kann auch über die Zugriffsoperationen auf das Attribut beschrieben werden. Daher enthalten die Beschreibungen der Objekttypen im Entwurf von MUNOS keine direkten Angaben zu Attributen. Es folgt nun, in einem Auszug aus dem Entwurf von MUNOS, eine Beschreibung des Aufbaus der Beschreibung eines Objekttyps.

*Auszug aus dem Entwurf von MUNOS*

<b>Objekttyp:</b>	Eindeutige Bezeichnung des Objekttyps.
<b>Definition:</b>	Definition des Objekttyps.
<b>Subsystem:</b>	Subsystem, zu dem dieser Objekttyp gehört.
<b>Supertypen:</b>	Auflistung der direkten Supertypen dieses Objekttyps.
<b>Subtypen:</b>	Auflistung der direkten Subtypen dieses Objekttyps.
<b>Verantwortlichkeiten:</b>	Beschreibung der Aufgaben, die ein Objekt dieses Objekttyps zu erfüllen hat. Damit wird die Rolle dieses Objekttyps definiert. Die einzelnen Aufgaben werden jeweils durchnummeriert, um sich darauf beziehen zu können.
<b>kommuniziert mit:</b>	Auflistung der Objekttypen der Objekte, mit denen ein Objekt dieses Objekttyps zusammenarbeitet, um seinen Verantwortlichkeiten nachzukommen. Die Liste ist nach den Nummern der Aufgaben geordnet.
<b>Operationen:</b>	Alphabetische Auflistung der Operationen dieses Objekttyps. Dabei werden auch ererbte Operationen angegeben, wenn sie in diesem Objekttyp (evtl. noch einmal) implementiert werden. Abstrakte Operationen werden mit dem Zusatz ( <i>abstrakt</i> ) versehen.
<b>Restriktionen:</b>	Auflistung der Restriktionen für diesen Objekttyp bzw. ein Objekt dieses Objekttyps.
<b>persistent:</b>	Angabe, ob Objekte dieses Objekttyps persistent sind.
<b>Parameter:</b>	Auflistung der formalen generischen Parameter eines parametrisierten Objekttyps bzw. der konkreten generischen Parameter eines instanziierten Objekttyps.

## 7.2.4 Die Beschreibungen der Objekttypen für Liniensysteme

Auf Grund der in der Use-Case-Beschreibung geschilderten Erwartung des Hinzukommens weiterer Arten von Liniensystemen (vgl. 7.1.2, S. 62), wurde ein abstrakter Objekttyp für Liniensysteme eingeführt, dessen erster konkreter Subtyp der Objekttyp für das herkömmliche 5-Linien-System ist. Auf die Beschreibungen dieser beiden Objekttypen wird in den nächsten beiden Abschnitten näher eingegangen.

### 7.2.4.1 Die Beschreibung des abstrakten Objekttyps für ein Liniensystem

Der abstrakte Objekttyp für Liniensysteme *Staff* ist ein Teil des Notations-Modells, das im Entwurf in Form des Subsystems *Notational Model* realisiert wurde. Bevor auf der nächsten Seite die Beschreibung des Objekttyps *Staff* folgt, hier ein paar Anmerkungen, die man parallel dazu lesen sollte, um die Beschreibung besser zu verstehen:

Der Supertyp *DefaultSubject* bringt *Staff* in die Rolle eines Subjekts, das von anderen Objekten beobachtet wird. Ein Liniensystem im Notations-Modell wird jeweils von den Layouts beobachtet, die mit auf diesem Liniensystem basieren und deshalb von diesem abhängig sind. Die Rolle wurde ausgelagert, da auch Noten, Pausenzeichen, notenbezogene Texte und Stimmen beobachtet werden. Der *DefaultLayoutManager* übernimmt dann jeweils die Vermittlerrolle zwischen den Objekten, die beobachtet werden, und deren Beobachter. Die Kommunikation der Liniensysteme mit dem *DefaultLayoutManager* taucht nicht in der Beschreibung von *Staff* auf, da sie von *DefaultSubject* ererbt wird.

Der Supertyp *BarManager* versetzt *Staff* in die Lage, Objekte taktweise und, innerhalb der Takte, taktpositionsweise zu verwalten. Diese Rolle wurde ausgelagert, da eine Stimme ebenfalls dazu in der Lage sein muß.

*Staff* besitzt mit *DefaultSubject* und *BarManager* zwei Supertypen, die zwei völlig verschiedene und voneinander unabhängige Rollen implizieren. In so einem Fall läßt sich Mehrfachvererbung sinnvoll einsetzen, um die Situation realistischer zu modellieren.

*RealRepresentation* ist ein abstrakter Objekttyp, in dem ein einheitliches Interface für alle Repräsentationen von Tönen und realen Pausen definiert ist.

Es folgt nun, in einem Auszug aus dem Entwurf von MUNOS, die Beschreibung des abstrakten Objekttyps *Staff*.

*Auszug aus dem Entwurf von MUNOS*

<b>Objekttyp:</b>	Staff
<b>Definition:</b>	Abstrakter Objekttyp für ein beliebiges Liniensystem.
<b>Subsystem:</b>	Notational Model
<b>Supertypen:</b>	<ul style="list-style-type: none"> <li>• DefaultSubject</li> <li>• BarManager</li> </ul>
<b>Subtypen:</b>	FiveLinesStaff
<b>Verantwortlichkeiten:</b>	<ol style="list-style-type: none"> <li>1. Durch Staff wird ein einheitliches Interface für alle Liniensysteme definiert.</li> <li>2. Wenn ein Liniensystem gelöscht wird, so meldet sich dieses bei den Repräsentationen der Töne bzw. Pausen ab, deren Noten bzw. Pausenzeichen in dem zu löschenden Liniensystem dargestellt werden sollten.</li> </ol>
<b>kommuniziert mit:</b>	2: RealRepresentation
<b>Operationen:</b>	<ul style="list-style-type: none"> <li>• create (<i>abstrakt</i>)</li> <li>• delete (<i>abstrakt</i>)</li> <li>• getClefs</li> <li>• getKeys</li> <li>• getManager</li> <li>• getOrdinal</li> <li>• getState</li> <li>• getTimes</li> <li>• setClef</li> <li>• setKey</li> <li>• setOrdinal</li> <li>• setTime</li> </ul>
<b>Restriktionen:</b>	Dieser Objekttyp kann keine Instanzen haben.
<b>persistent:</b>	Nein
<b>Parameter:</b>	–

**7.2.4.2 Die Beschreibung des Objekttyps für ein herkömmliches 5-Linien-System**

Es folgt nun, in einem Auszug aus dem Entwurf von MUNOS, die Beschreibung des Objekttyps *FiveLinesStaff*.

*Auszug aus dem Entwurf von MUNOS*

<b>Objekttyp:</b>	FiveLinesStaff
<b>Definition:</b>	Objekttyp für ein herkömmliches 5-Linien-System.
<b>Subsystem:</b>	Notational Model
<b>Supertypen:</b>	Staff
<b>Subtypen:</b>	–
<b>Verantwortlichkeiten:</b>	1. FiveLinesStaff enthält alle Grundparametereinstellungen zur Darstellung eines herkömmlichen 5-Linien-Systems.
<b>kommuniziert mit:</b>	–
<b>Operationen:</b>	<ul style="list-style-type: none"> <li>• create</li> <li>• delete</li> </ul>
<b>Restriktionen:</b>	–
<b>persistent:</b>	Nein
<b>Parameter:</b>	–

## 7.2.5 Aufbau der Beschreibung einer Operation

Es folgt nun, in einem Auszug aus dem Entwurf von MUNOS, eine Beschreibung des Aufbaus der Beschreibung einer Operation eines Objekttyps.

*Auszug aus dem Entwurf von MUNOS*

<b>Operation:</b>	Eindeutige Bezeichnung der Operation.
<b>Objektyp:</b>	Objektyp, zu dem diese Operation gehört.
<b>abstrakt:</b>	Angabe, ob es sich um eine abstrakte Operation handelt. Eine abstrakte Operation wird in ihrem Objekttyp nicht implementiert.
<b>Argumente:</b>	Auflistung der formalen Argumente, sowie deren Datenstrukturen (z.B. <i>Liste</i> ). Erst im implementierungsabhängigen Entwurf werden hier konkrete Datentypen angegeben. Optionale Argumente werden mit dem Zusatz ( <i>optional</i> ) versehen.
<b>Rückgabe:</b>	Datenstruktur, die nach der Durchführung der Operation zurückgegeben wird. Auch hier werden erst im implementierungsabhängigen Entwurf konkrete Datentypen angegeben.
<b>Verantwortlichkeiten:</b>	Auflistung der Verantwortlichkeiten des Objekttyps dieser Operation, zu deren Erfüllung diese Operation beiträgt. Dabei wird jeweils nur die Nummer der Verantwortlichkeit angegeben, wie sie in der Beschreibung des Objekttyps dieser Operation zu finden ist. Wenn diese Operation nur einen Teil einer Verantwortlichkeit erfüllt, so wird zu der Nummer der Verantwortlichkeit angegeben, welchen Teil der Verantwortlichkeit diese Operation erfüllt.
<b>Aktionen:</b>	Beschreibung der Aktionen, die in dieser Operation ausgeführt werden, um den Verantwortlichkeiten nachzukommen. Die Aktionen werden nach den Nummern der Verantwortlichkeiten gegliedert.

*(wird auf der nächsten Seite fortgesetzt)*

*Auszug aus dem Entwurf von MUNOS (Fortsetzung)*

<b>Vorbedingungen:</b>	Auflistung der Bedingungen, die erfüllt sein müssen, damit diese Operation durchgeführt wird.
<b>Nachbedingungen:</b>	Auflistung der Bedingungen, die nach der korrekten und vollständigen Durchführung dieser Operation garantiert erfüllt sind.
<b>Ausnahmen:</b>	Auflistung der Ausnahmefälle, die diese Operation auslösen kann.
<b>Algorithmus:</b>	Beschreibung des in dieser Operation verwendeten Algorithmus.

## 7.2.6 Die Beschreibung einer Operation des abstrakten Objekttyps für Liniensysteme

Es folgt nun, in einem Auszug aus dem Entwurf von MUNOS, die Beschreibung der abstrakten *delete*-Operation des abstrakten Objekttyps *staff* (vgl. 7.2.4.1, S. 70). In dieser Beschreibung taucht das in dem Interaktionsdiagramm für den Fall des Löschens erfolgende Abmelden beim *Default-LayoutManager* (vgl. 7.2.2, S. 67) nicht auf, da dieses Verhalten von *DefaultSubject* ererbt wird (vgl. 7.2.4.1, S. 69).

*Auszug aus dem Entwurf von MUNOS*

<b>Operation:</b>	delete
<b>Objekttyp:</b>	Staff
<b>abstrakt:</b>	Ja
<b>Argumente:</b>	–
<b>Rückgabe:</b>	–
<b>Verantwortlichkeiten:</b>	2
<b>Aktionen:</b>	2: An alle Repräsentationen von Tönen bzw. Pausen des zu löschenden Liniensystems ergeht eine <i>unregister</i> -Nachricht, mit dem zu löschenden Liniensystem als Argument.
<b>Vorbedingungen:</b>	–
<b>Nachbedingungen:</b>	Das zu löschende Liniensystem wurde erfolgreich gelöscht und dabei die Verantwortlichkeit 2 des zu löschenden Liniensystems vollständig erfüllt.
<b>Ausnahmen:</b>	–
<b>Algorithmus:</b>	–

## 8 Entwurfsmuster in MUNOS

In diesem Kapitel werden zwei Beispiele für die Verwirklichung von Entwurfsmustern in MUNOS gegeben. Im ersten Beispiel dreht es sich um ein wiederverwendetes Entwurfsmuster, das dem Buch *Design Patterns* (vgl. Gamma et al., 1995) entnommen ist, in dem es als eine Variante des Entwurfsmusters *Observer* besprochen wird. Das Entwurfsmuster *Observer* bietet ein Lösungsschema für einen Abhängigkeitsmechanismus (vgl. 6.2.3.2, S. 49) an, wie er auch in Model-View-Controller-Kontexten vorkommt.

Im zweiten Beispiel geht es um ein selbst gewonnenes Entwurfsmuster, das *Server* getauft wurde. Das Entwurfsmuster *Server* läßt sich einsetzen, wenn zur Laufzeit zwischen verschiedenen alternativen Objekttypen einer ausgewählt werden muß, von dem dann eine Instanz erzeugt werden soll.

### 8.1 Ein Beispiel für ein wiederverwendetes Entwurfsmuster

Wie in 4.1 (vgl. S. 21ff) bereits beschrieben wurde, ist jede Programm-View von MUNOS wieder nach dem Model-View-Controller-Konzept aufgebaut und besteht aus einem Notations-Modell, einer oder mehreren Notations-Views und einem Notations-Controller. Diese Tatsache spiegelt sich im Entwurf von MUNOS durch das Subsystem *View* wieder, das aus den Subsystemen *Notational Model*, *Notational View* und *Notational Controller* besteht.

Wenn an einem Notations-Modell etwas geändert wird, dann müssen die davon betroffenen Notations-Views aktualisiert werden. Dabei gibt es zwei Probleme zu lösen. Einerseits muß die Abhängigkeit zwischen Objekten der Notations-Views und Objekten eines Notations-Modells modelliert werden, d.h. bei einer Änderung an Objekten eines Notations-Modells müssen die davon betroffenen Objekte der Notations-Views darüber informiert werden. Andererseits muß in einem Änderungsfall zuerst einmal festgestellt werden, welche Objekte welcher Notations-Views überhaupt von der Änderung betroffen sind und darüber informiert werden müssen.

Lösungen zu diesen beiden Probleme bieten das Entwurfsmuster *Observer* und dessen Varianten. Eine dieser Varianten wurde in dem Subsystem *View* realisiert.

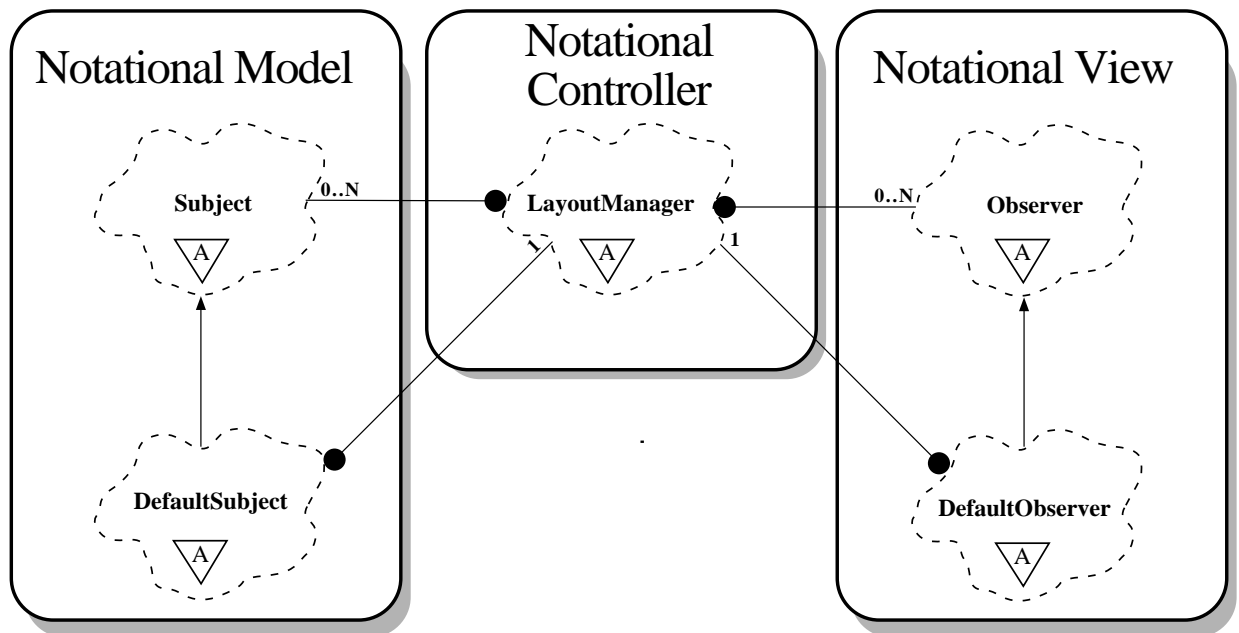


Abb. 8.1: Eine in MUNOS realisierte Variante des Entwurfsmusters *Observer*

Die Abbildung 8.1 zeigt das statische Diagramm des Subsystems *View*. Darin abgebildet sind die drei Subsysteme, aus denen das Subsystem *View* besteht, sowie die Objekttypen der Schnittstellen zwischen diesen drei Subsystemen. Die drei Subsysteme des Subsystems *View* hängen nur über diese Schnittstellen-Objekttypen zusammen, die eine Realisierung der in MUNOS verwendeten Variante des Entwurfsmusters *Observer* sind. Aus diesem Entwurfsmuster resultiert auch das gesamte Zusammenwirken der drei Subsysteme innerhalb des Subsystems *View*. Auf die Schnittstellen-Objekttypen und das Verhalten, das jeweils durch diese vererbt wird, wird nun genauer eingegangen.

Wenn von einem Entwickler nichts anderes vorgegeben wird, sind alle Objekte im Notations-Modell, deren Änderung Objekte in Notations-Views betreffen kann, letztendlich Subtypen von *DefaultSubject* und ererben dadurch die Rolle von „Standard-Subjekten“, die jeweils von den von ihnen abhängigen Objekten der Notations-Views beobachtet werden. Ein Standard-Subjekt benachrichtigt seinen Layout-Manager über jegliche Änderung, die an ihm, dem Standard-Subjekt, vorgenommen wird.

Analog dazu sind, wenn von einem Entwickler nichts anderes vorgegeben wird, alle vom Notations-Modell abhängigen Objekte in Notations-Views letztendlich Subtypen von *DefaultObserver* und ererben dadurch die Rolle eines „Standard-Beobachters“, der jeweils von den Änderungen an einem Objekt des Notations-Modells abhängig ist und daher dieses beobachtet. Ein Standard-Beobachter wird von dessen Layout-Manager über jegliche Änderung an dem Objekt im Notations-Modell, von dem der Standard-Beobachter abhängig ist, informiert.

Zwischen den „Subjekten“ des Notations-Modells und den jeweils davon abhängigen „Beobachtern“ der Notations-Views vermittelt also ein Layout-Manager. Daher ist es die Aufgabe eines Layout-Managers zu wissen, welche Änderungen am Modell welche Objekte welcher Views betreffen und diese Änderungen jeweils weiterzuvermitteln.

*Subject* und *Observer* wurden eingeführt, um einem Entwickler zu ermöglichen, in MUNOS einen anders funktionierenden Änderungsmechanismus zu realisieren. Zum Beispiel wäre es machbar, mit mehreren Layout-Managern für jedes Notations-Modell zu arbeiten, die jeweils für ganz bestimmte Änderungen zuständig sind. Daher ist bei *Subject* und bei *Observer* jeweils die Beschränkung auf genau einen Layout-Manager noch nicht vorhanden.

Zum Schluß noch eine Anmerkung zu dieser in MUNOS verwendeten Variante von *Observer*. Das eigentliche Basis-Entwurfsmuster kommt ohne einen vermittelnden Layout-Manager aus. Einen Vermittler dazwischenschalten lohnt sich dann, wenn ein komplexer Änderungsmechanismus verwirklicht werden muß, in dem zum Beispiel Änderungen voneinander abhängig sein können, wie es auch in MUNOS der Fall ist. Genau genommen ist bereits der Einsatz eines Vermittlers ein Entwurfsmuster, das den Namen *Mediator* trägt (vgl. Gamma et al., 1995).

## 8.2 Ein Beispiel für ein gewonnenes Entwurfsmuster

Bei der Entwicklung von MUNOS gab es zwei Probleme zu lösen, die sich dann durch Abstraktion auf ein Kernproblem reduzieren ließen. Das eine Problem bestand darin, ein Objekt des Notations-Modells nicht fest mit einer bestimmten graphische Darstellung des Objekts zu verbinden. Zum Beispiel soll es zur Laufzeit möglich sein, unter mehreren Alternativen für die graphische Darstellung einer Taktangabe wählen zu können, wobei die verschiedenen Alternativen unter Umständen auch in der Anzahl an Graphikobjekten, aus denen sie sich jeweils zusammensetzen, variieren können.

Das andere Problem bestand darin, die Lösung eines Problems nicht fest mit einem bestimmten Lösungsalgorithmus dafür zu verbinden. Zum Beispiel soll es zur Laufzeit möglich sein, unter mehreren alternativen Algorithmen für die Repräsentation eines Tones in Form von einer oder mehreren Noten einen Algorithmus auswählen zu können.

Wenn nun einerseits Objekte eingeführt werden, die jeweils die alternativen graphischen Darstellungen eines bestimmten Sachverhalts erzeugen können und andererseits jeder Algorithmus in ein Algorithmus-Objekt ausgelagert wird, so lassen sich die beiden Probleme auf das Kernproblem reduzieren, zur Laufzeit zwischen der Erzeugung verschiedener alternativer Objekte für etwas wählen zu können. Eine Lösung dafür besteht darin, ein „Server-Objekt“ einzuführen, an das sich andere Objekte wenden können, wenn sie ein bestimmtes Objekt unter alternativen Objekten benötigen und das für die Erzeugung eines der alternativen Objekte verantwortlich ist. Das ist dann auch die Idee, die hinter dem Entwurfsmuster *Server* steckt.

Aus semantischen Gründen wurden in MUNOS gleich zwei zentrale Server realisiert, einer für Objekte, die Graphikobjekte erzeugen können und einer für Algorithmus-Objekte. Dann wurden alle Algorithmus-Objekte mit dem dazugehörigen Server in dem Subsystem *Notational Strategies* und alle Objekte zur Erzeugung graphischer Objekte, mit dem zugehörigen Server in dem Subsystem *Shape Creators* untergebracht.

Da MUNOS in der Lage sein soll, mit mehreren Programm-Views parallel zu arbeiten, erscheint es auch sinnvoll, die beiden Server-Objekte global zugänglich zu machen, denn in jeder Programm-View sollen jeweils dieselben Graphik-objekte und Algorithmus-Objekte zur Auswahl stehen. Deshalb wurde das Objekt *Globals* eingeführt, das später über eine globale Variable allgemein zugänglich gemacht werden soll und über das auf die beiden Server zugegriffen werden kann.

Das globale Objekt bildet zusammen mit den zwei Subsystemen *Notational Strategies* und *Shape Creators* das Subsystem *Global Objekts*. Die Abbildung 8.2 zeigt ein statisches Diagramm mit den beiden Realisierungen des Entwurfsmusters *Server* in MUNOS.

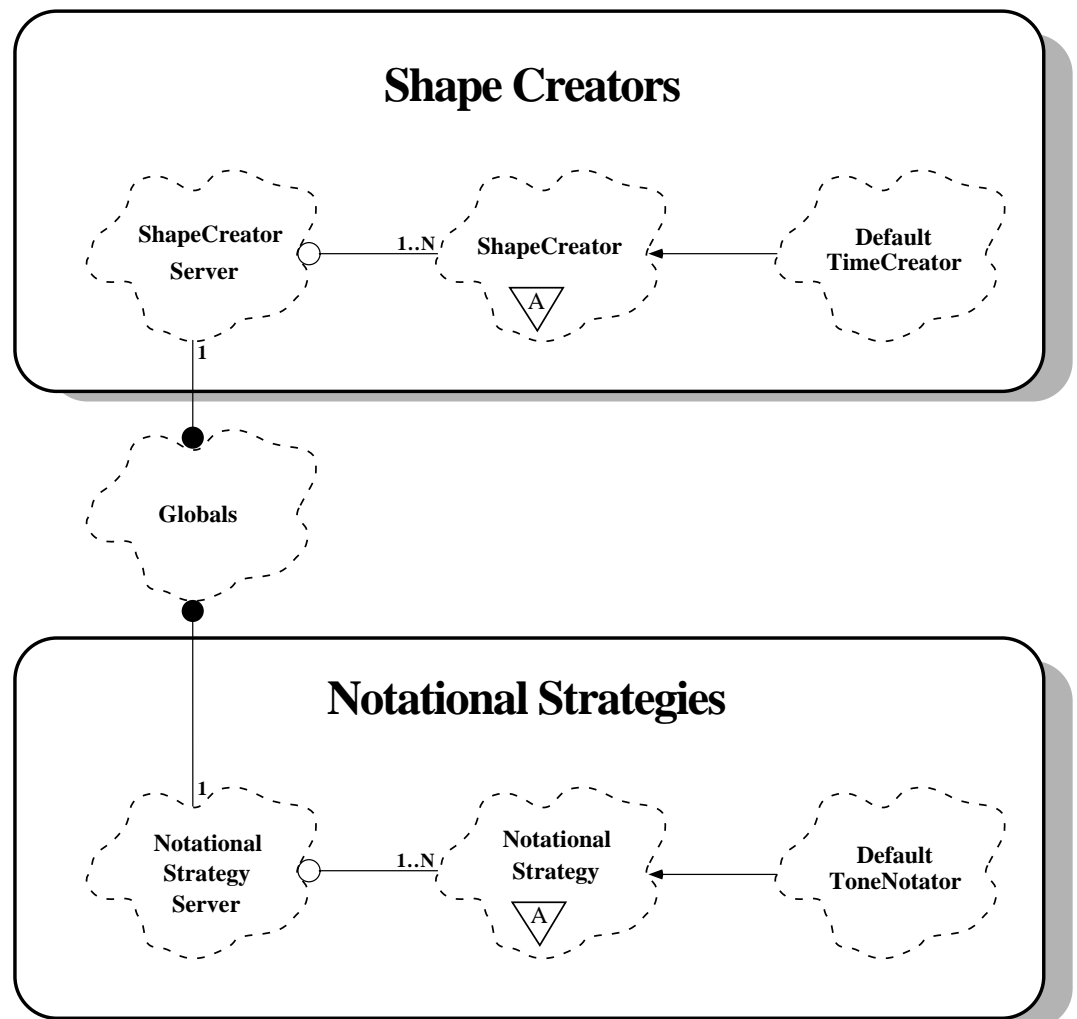


Abb. 8.2: Die beiden Realisierungen des Entwurfsmusters *Server* in MUNOS

Darin sind die beiden Server-Objekte jeweils in *Globals* enthalten und so global zugänglich. Durch den abstrakten Objekttyp *ShapeCreator* wird ein gemeinsames Interface für alle Objekte, die Graphikobjekte erzeugen, festgelegt. In dem Interface sind in erster Linie die Operationen definiert, die der zugehörige Server benötigt. Als Beispiel für ein Objekt, das alternative graphische Darstellungen eines Sachverhalts erzeugen kann, ist der *DefaultTimeCreator* abgebildet, der für die Erzeugung der graphischen Darstellungen von Taktarten zuständig ist. Analog dazu wird durch *NotationalStrategy* ein gemeinsames Interface für alle Algorithmus-Objekte festgelegt. Auch hier werden in dem Interface in erster Linie die Operationen definiert, die der zugehörige Server benötigt. Als Beispiel für ein Algorithmus-Objekt ist der *DefaultToneNotator* abgebildet, der einen Algorithmus für die Notation von Tönen in Form von Noten enthält.

Wenn nun zum Beispiel das Repräsentations-Objekt eines Tones den Algorithmus braucht, durch den bestimmt wird, aus wievielen und welchen Noten die Repräsentation seines Tones bestehen soll, so greift das Repräsentations-Objekt über *Globals* auf den *NotationalStrategyServer* zu und fordert bei diesem das Objekt mit dem benötigten Algorithmus an. Daraufhin erzeugt *NotationalStrategyServer* das gewünschte Algorithmus-Objekt und gibt dieses an das wartende Repräsentations-Objekt zurück. Dann beauftragt das Repräsentations-Objekt das Algorithmus-Objekt, die Anzahl und Art der Noten zu bestimmen, die die Repräsentation seines Tones haben muß. Wenn das Algorithmus-Objekt die für die Repräsentation benötigte Anzahl und Art der Noten berechnet hat, gibt es das Ergebnis an das wartende Repräsentations-Objekt zurück und löscht sich anschließend selbst.

Dieses Beispiel war zwar sehr spezifisch, doch ich hoffe damit auch die generelle Funktionsweise eines Servers deutlich gemacht zu haben. Allgemein läßt sich sagen, um einen Server zu realisieren, benötigt man den Server selbst, einen abstrakten Objekttyp, in dessen Interface die Funktionalität definiert ist, die der Server benötigt, sowie die Objekttypen, die jeweils ein Subtyp des Interface-Objekttyps sind und auf die der Server zugreifen kann. In dieser ursprünglichen Version kennt ein Server alle Objekte, die er erzeugen kann. Eine denkbare Variante wäre daher ein Server, der die Objekte die er erzeugt, vorher nicht zu kennen braucht.

In MUNOS wird einem Entwickler durch den Server für Objekte, die graphische Darstellungen erzeugen können, die Möglichkeit gegeben, seine eigenen Objekttypen zu Erzeugung graphischer Darstellungen definieren und verwenden zu können, um damit die graphische Darstellungen von MUNOS an seine jeweiligen Bedürfnissen anzupassen.

Analog dazu wird einem Entwickler durch den Server für Algorithmus-Objekte die grundsätzliche Möglichkeit eröffnet, eigene Algorithmus-Objekttypen definieren und verwenden zu können, um dann in MUNOS mit seinen eigenen Algorithmen arbeiten zu können, wo es ihm beliebt.

## 9 Vergleich mit anderen Lösungen

Abgesehen von MUNOS gibt es auch noch andere Lösungen, um an eine graphische Darstellung von Noten zu kommen. Diese unterscheiden sich in ihrem Charakter allerdings wesentlich von MUNOS, zu dem mir bislang kein Pendant bekannt ist (vgl. 2.2, S. 13). In diesem Kapitel werden die verschiedenen Alternativen zu MUNOS vorgestellt. Wo es sinnvoll erscheint, wird dabei als Beispiel das Generalbaßprogramm verwendet (vgl. 2.1, S. 12). Am Schluß des Kapitels wird dann in der Zusammenfassung beschrieben, wo diese Lösungen im Vergleich zu MUNOS anzusiedeln sind.

### 9.1 Die vermeintliche Lösung

Eine denkbare Lösung besteht darin, die Daten zwischen dem eigenen Programm und einem Notensatzprogramm bzw. einem Sequenzerprogramm über eine Software-Schnittstelle, die von dem Notensatzprogramm bzw. in dem Sequenzerprogramm zur Verfügung gestellt wird, zur Laufzeit auszutauschen, wie es Abbildung 9.1 zeigt.

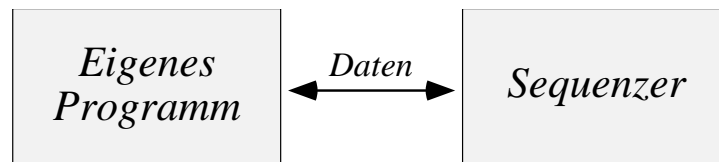


Abb. 9.1: Ein direkter Datenaustausch zwischen dem eigenen Programm und einem Sequenzerprogramm

Zum Beispiel vertreibt die Firma Emagic neben ihrem Sequenzerprogramm *Logic* noch diverse andere Programme, wie *SoundDiver* zum Editieren und Verwalten von Klängen, die an sich zwar eigenständig sind, die aber auch über *AutoLink* (vgl. Emagic, 1995) mit *Logic* gekoppelt werden können. Allerdings, so ergab eine Nachfrage, wird *AutoLink* nur firmenintern genutzt. Um *AutoLink* allgemein zugänglich zu machen, müßte der Programmcode herausgegeben werden. Auch zu *Enigma*, mit dem von der Firma Coda in ihren Programmen, wie zum Beispiel dem Notensatzprogramm *Finale*, gearbeitet wird, war nichts erhältlich. OMS von der Firma Opcode ist in diesem Zusammenhang in die Kategorie MIDI-Teiber (vgl. 9.2, S. 83ff) einzuordnen.

So lange keine konkreten Lösungen dieser Art für jeden Entwickler zugänglich sind, wird dies nur eine „vermeintliche“ Lösung des Problems bleiben.

## 9.2 Die MIDI-Treiber Lösung

Eine denkbare Lösung besteht darin, die Daten im MIDI-Format über einen standardisierten MIDI-Treiber zwischen dem eigenen Programm und einem Notensatzprogramm, das Eingaben über MIDI verarbeiten kann, bzw. einem Sequenzerprogramm auszutauschen, wie es Abbildung 9.2 zeigt. Als Voraussetzung für diese Lösung muß einerseits im Rechner ein standardisierter MIDI-Treiber seinen Dienst tun und andererseits muß das Notensatzprogramm bzw. der Sequenzer mit diesem MIDI-Treiber arbeiten. Standardisierte MIDI-Treiber gibt es zum Beispiel von Apple oder von Microsoft, die jeweils einen MIDI-Treiber als Bestandteil ihres Betriebssystems mit ausliefern. Notensatzprogramme bzw. Sequenzer, die für diese Plattformen angeboten werden, bieten in der Regel auch die Option den Standard-MIDI-Treiber zu verwenden.

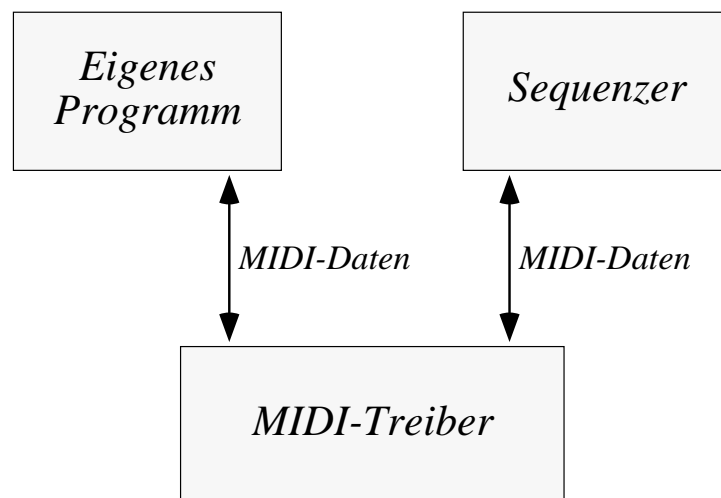


Abb. 9.2: Ein Austausch von Daten im MIDI-Format über einen MIDI-Treiber zwischen dem eigenen Programm und einem Sequenzerprogramm

Der Vorteil dieser Lösung liegt darin, das Problem in ein anderes Programm auslagern zu können. Dadurch ist der Aufwand im eigenen Programm nicht mehr so groß und läßt sich auf die Umwandlung der Datenformate und den Datenaustausch beschränken.

Der Nachteil dieser Lösung liegt darin, nur sehr eingeschränkt interaktiv arbeiten zu können, denn einerseits muß immer vorher manuell eingegriffen werden, wenn im Sequenzprogramm geänderte Daten wieder zurückgegeben werden sollen und andererseits ist es unmöglich, bestimmte Noten in der Darstellung im Sequenzerprogramm vom eigenen Programm aus zu markieren.

Diese Lösung eignet sich für Fälle, in denen das reine Notenbild an sich, sowie dessen Bearbeitung von Interesse ist. Zum Beispiel ist diese Lösung für das Generalbaßprogramm geeignet, wenn nur dessen Ergebnisse in Form von Noten dargestellt werden sollen.

Diese Lösung eignet sich nicht für Fälle, in denen im eigenen Programm eine musiktheoretische Semantik hinter der graphischen Darstellung von Noten steckt und mit dieser Semantik dann über das Notenbild interaktiv gearbeitet werden soll. Zum Beispiel ist diese Lösung für das Generalbaßprogramm ungeeignet, wenn das Generalbaßprogramm auf die vom Benutzer im Notenbild erfolgte Auswahl einer bestimmten musikalischen Wendung entsprechend reagieren soll oder wenn es dem Generalbaßprogramm selbst möglich sein soll, bestimmte musikalische Wendungen im Notenbild zu markieren.

### 9.3 Die MIDI-Standardfile Lösung

Eine, zu der in 9.2 vorgestellten MIDI-Treiber Lösung, sehr ähnliche Lösung, ist die MIDI-Standardfile Lösung, wie sie die Abbildung 9.3 auf der nächsten Seite zeigt. Diese Lösung besteht darin, die gewünschten Daten durch das eigene Programm in Form eines MIDI-Standardfiles abzuspeichern, dieses MIDI-Standardfile mittels eines geeigneten Notensatzprogramms oder Sequenzerprogramms einzulesen und dann in diesem in Form von Noten darstellen zu lassen. Die Vermittler-Funktion des MIDI-Treibers aus 9.2 wird in diesem Fall von einem MIDI-Standardfile übernommen.

Der Vorteil dieser Lösung liegt, wie bei 9.2, in dem relativ niedrigen Programmieraufwand beim eigenen Programm, da das Problem der Notendarstellung wieder komplett ausgelagert wird. Darüber hinaus bietet diese Lösung den Vorteil, nicht von einem bestimmten MIDI-Treiber abhängig zu sein.

Der Nachteil dieser Lösung liegt, wie bei 9.2, darin, nur sehr eingeschränkt interaktiv arbeiten zu können. Dazu kommt in diesem Fall allerdings noch der erhöhte Aufwand, der später ständig für den Datenaustausch betrieben werden muß.

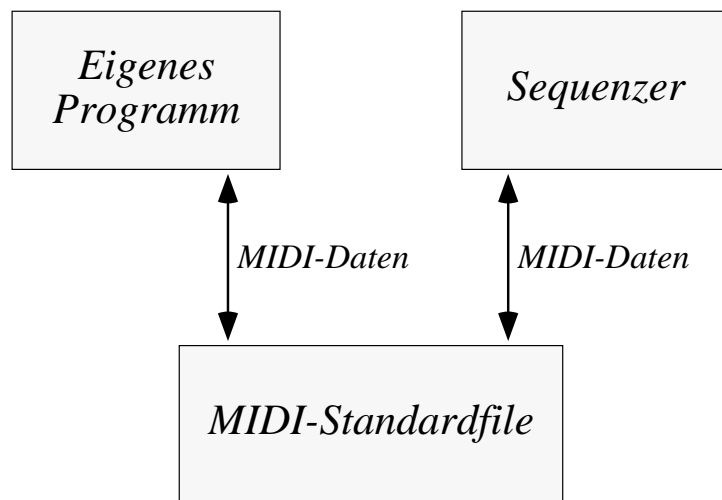


Abb. 9.3: Ein Austausch von Daten im MIDI-Format über ein MIDI-Standardfile zwischen dem eigenen Programm und einem Sequenzerprogramm

Wenn relativ häufig Daten ausgetauscht werden müssen, so ist es daher auf längere Sicht gesehen besser, einmal bei der Programmierung einen größeren Aufwand zu betreiben, um damit später den ständigen Aufwand des Datenaustausches einzusparen.

Diese Lösung eignet sich, wie 9.2, für Fälle in denen das reine Notenbild an sich, sowie dessen Bearbeitung, von Interesse ist. Das gilt in diesem Fall allerdings nur, wenn die Daten nicht zu häufig oder unabhängig vom jeweiligen MIDI-Treiber ausgetauscht werden müssen. Zum Beispiel ist diese Lösung für das Generalbaßprogramm geeignet, wenn nur dessen Ergebnisse in Form von Noten dargestellt werden sollen und das Generalbaßprogramm nicht ständig in Gebrauch ist.

Diese Lösung eignet sich, wie 9.2, nicht für Fälle, in denen im eigenen Programm eine musiktheoretische Semantik hinter der graphischen Darstellung von Noten steckt und mit dieser Semantik dann über das Notenbild interaktiv gearbeitet werden soll.

Darüber hinaus ist diese Lösung ungeeignet, wenn ein häufiger Datenaustausch stattfinden soll. Zum Beispiel ist diese Lösung für das Generalbaßprogramm ungeeignet, wenn das Generalbaßprogramm auf die vom Benutzer im Notenbild erfolgte Auswahl einer bestimmten musikalischen Wendung entsprechend reagieren soll, wenn es dem Generalbaßprogramm selbst möglich sein soll, bestimmte musikalische Wendungen im Notenbild zu markieren oder wenn das Generalbaßprogramm ständig verwendet wird.

## 9.4 Die fragwürdige Lösung

Grundsätzlich ist es auch möglich, den Programmcode zur Notendarstellung eines anderen Programms an das eigene Programm anzupassen. Neben dem Problem an den Code zu kommen, ist es immer vom Einzelfall abhängig, ob diese Lösung überhaupt gangbar ist und ob man sich damit einen großen Gefallen tut.

Der Vorteil dieser Lösung liegt in dem vorliegenden Programmcode für eine Notendarstellung. Die Nachteile dieser Lösung liegen meist in einer sehr langen Einarbeitungszeit in den Programmcode und in dem nicht unerheblichen Aufwand der Anpassung dieses Programmcodes an das eigene Programm. Da in den meisten Fällen die Nachteile überwiegen, wird auf diese auch Lösung nicht weiter eingegangen.

## 9.5 Der letzte Ausweg

Die letzte Möglichkeit besteht darin, die Notendarstellung für das eigene Programm selbst zu schreiben. Dies ist zwar der flexibelste, aber auch der aufwendigste Weg.

Der Vorteil dieser Lösung liegt in ihrer optimalen Anpassung an die eigenen Bedürfnisse. Die Nachteile dieser Lösung liegen einerseits in dem beträchtlichen Aufwand dafür, eine Notendarstellung zu schreiben und andererseits in dem dazu nötigen Fachwissen, wie eine korrekte Notendarstellung im Detail aussieht. Zum Beispiel gibt es unter „Stechern“, also Leuten, die Noten drucken, feste Regeln dafür, wie Balken korrekt notiert werden (vgl. Roemer, 1973).

Diese Lösung eignet sich für Fälle, in denen der Aufwand nicht gescheut wird und genügend Zeit, sowie das nötige Fachwissen, mitgebracht werden. Diese Lösung eignet sich nur bedingt für Fälle, in denen zwar der Aufwand nicht gescheut wird und die nötige Zeit auch vorhanden ist, aber das nötige Fachwissen nicht mitgebracht wird. Es hängt dann von den Qualitätsansprüchen ab, denen die Notendarstellung genügen soll, ob dieser Weg gegangen wird oder nicht. Diese Lösung ist ungeeignet, wenn man denn Aufwand scheut oder die Zeit, um diesen betreiben zu können, gar nicht zur Verfügung steht.

## 9.6 Zusammenfassung

Den ersten drei Lösungen liegt jeweils die Idee zu Grunde, das Problem der Notendarstellung aus dem eigenen Programm in ein anderes Programm zu verlagern, indem dazu ein Notensatzprogramm oder ein dafür geeignetes Sequenzerprogramm verwendet wird. Notensatzprogramme und Sequenzerprogramme sind allerdings nicht für eine derartige Anwendung gemacht und man tut sich dementsprechend schwer. Darin zeigt sich letztendlich auch der Unterschied zwischen Notensatzprogrammen bzw. Sequenzerprogrammen und MUNOS, denn Notensatzprogramme bzw. Sequenzerprogrammen sind Anwenderprogramme, die für den Endbenutzer gedacht sind und MUNOS ist eine Bibliothek, die für Software-Entwickler gedacht ist (vgl. 2.3, S. 13ff).

In den letzten beiden Lösungen hat man es jeweils mit der Notendarstellung eines Programms zu tun, die genau auf die Bedürfnisse des Programms und dessen Entwickler zugeschnitten ist. Das macht in diesen Fällen den Hauptunterschied zu MUNOS aus, denn MUNOS ist als eine Bibliothek eher auf Flexibilität und damit auf möglichst breite Anwendbarkeit ausgelegt.

## 10 Kritischer Rückblick und Ausblick

In diesem Kapitel wird zunächst ein kritischer Rückblick auf den Verlauf dieser Arbeit geworfen. Dann wird auf den aktuellen Stand des MUNOS-Projekts eingegangen, wozu mit dieser Arbeit der Grundstein gelegt wurde, und ein Ausblick auf dessen Weiterführung gegeben.

### 10.1 Kritischer Rückblick auf den Verlauf der Arbeit

Im Rahmen dieser Diplomarbeit wurde das MUNOS-Projekt begonnen, das anschließend an der Musikhochschule weitergeführt werden soll (vgl. 10.2, S.89). Die Arbeit umfaßte die erste Iteration bis zum Entwurf. Dabei entstanden eine Spezifikation von mehr als 80 Seiten und ein Entwurf von mehr als 140 Seiten. MUNOS besteht derzeit aus über 90 Klassen.

Im Verlaufe der Arbeit stellte sich beim Umgang mit Use-Cases ein Problem heraus, das generell beim Umgang mit Use-Cases besteht. Use-Cases spiegeln zwar Anwendungsfälle wieder und sagen damit auch etwas über den Funktionsumfang eines Systems aus, sie können aber sehr über den Arbeitsumfang der Realisierung dieses Funktionsumfangs hinwegtäuschen. Je höher der Aufwand an internen Berechnungen ist, die nicht direkt von einem Akteur außerhalb des Systems angefordert werden, sondern indirekt zur Realisierung eines Use-Case benötigt werden, desto schwieriger wird es, den dafür benötigten Arbeitsaufwand anhand der Use-Cases abzuschätzen.

Im Falle von MUNOS ist der Teil zum Erstellen eines Layouts mit vielen internen Berechnungen verbunden, wogegen im gesamten Rest eher wenig interne Berechnungen ablaufen (vgl. Kapitel 4, S.24ff). So ist „ein Layout erstellen“ der umfangreichste Use-Case, dessen Beschreibung um ca. 50% länger ist, als die des zweitgrößte Use-Case. Aber die Realisierung von „ein Layout erstellen“ ist im Entwurf umfangreicher ausgefallen, als alle anderen Use-Cases zusammengenommen, und damit wesentlich umfangreicher als allgemein erwartet wurde. Durch dieses krasse Mißverhältnis war auch keine sinnvolle Kürzung des Funktionsumfangs machbar. Zwar hätte man beispielsweise auf Stimmen verzichten können, aber das wiederum hätte sich nicht sonderlich auf den Verlauf der Arbeit ausgewirkt.

Im nachhinein betrachtet, wäre vielleicht eine Aufteilung in zwei Arbeiten am sinnvollsten gewesen. Eine Arbeit für das Notations-Modell und eine Arbeit für die Notations-View, wobei der Notations-Controller die gemeinsame Schnittstelle gewesen wäre (vgl. 4.1, S.88). Aber das läßt sich auch erst zu diesem Zeitpunkt sagen, denn diese Aufteilung ist im Verlauf dieser Arbeit entstanden. Deshalb wäre zusätzlich auch einiges an Vorarbeit nötig gewesen, um zunächst einmal bis zu dieser Aufteilung zu kommen.

Als sich der unterschätzte Arbeitsaufwand herauskristallisierte, wurde deshalb beschlossen, weiterhin auf Qualität vor Quantität zu setzen, indem zugunsten des Entwurfs auf eine Teil-Implementierung verzichtet wurde. Das war im Gesamtzusammenhang des MUNOS-Projekts gesehen besser für das Projekt, in dessen weiteren Verlauf noch vollständige Implementierungen geplant sind (vgl. 10.2). Anhand der im Entwurf verwendeten statischen Diagramme und Interaktionsdiagramme (vgl. 7.2, S. 65ff) läßt sich außerdem die Realisierbarkeit des Entwurfs relativ gut abschätzen. Diese Tatsache erleichterte die Entscheidung zugunsten des Entwurfs.

## 10.2 Ausblick

Da von Seiten der Musikhochschule Interesse an MUNOS existiert, wird dort versucht werden, das Projekt fortzusetzen (vgl. 2.1, S.12). Allerdings herrscht an der Musikhochschule bislang kaum eine Infrastruktur dafür, denn es gibt keinen offiziellen Zweig für Musik-Programmierung und es werden deshalb auch keine wissenschaftlichen Mitarbeiter in dieser Richtung beschäftigt. Die Hoffnung ruht eher auf den Informatik-Studenten mit Nebenfach Musiktheorie, die an der Musikhochschule im Rahmen des Nebenfachstudiums Musik-Software realisieren sollen.

Nach dieser kurzen Beschreibung der allgemeinen Situation, nun wieder zurück zum MUNOS-Projekt selbst. Es wird zunächst eine Version 2 des Entwurfs erstellt, in der noch einige Teile ergänzt werden. Parallel dazu wird ein implementierungsabhängiger Entwurf für Windows-Plattformen realisiert. Anschließend sollen zwei Implementierungen in C++ erfolgen, eine für Macintosh-Umgebungen und eine für Windows-Umgebungen.

Die Implementierungen von MUNOS und eventuell auch die Spezifikation und die Entwürfe sollen dann über Internet frei verfügbar gemacht werden. Dazu wird es nötig sein, englische Dokumentationen zu verfassen und eventuell englische Fassungen der Entwürfe und der Spezifikation anzufertigen.

Schließlich ist noch geplant, den Funktionsumfang von MUNOS im Laufe der Zeit zu erweitern und dabei auf die Wünsche von Entwicklern einzugehen. Es bleibt spannend, wohin das letztendlich führen wird.

# A Glossar

## **Abstrakte Operation**

Eine Operation, die in einem abstrakten Objekttyp definiert, aber dort nicht implementiert wird. Sie muß daher in Subtypen des abstrakten Objekttyps implementiert werden (vgl. *abstrakter Objekttyp, Operation*).

## **Abstrakter Objekttyp**

Ein Objekttyp, der keine Instanzen haben darf (vgl. *Objekttyp*).

## **Abstrakter Use-Case**

Ein Use-Case, der keine Akteure hat (vgl. *Akteur, Use-Case*, sowie 6.2.1, S. 42).

## **Akkolade**

Klammer, durch die mehrere Liniensysteme als gleichzeitig erklingend zusammengefaßt werden.

## **Akteur (engl. actor)**

Eine Person, eine Software oder eine Hardware außerhalb des Systems, die das System in einer bestimmten Weise benutzt. Dabei ist der Begriff des Akteurs nicht an die Person, die Software oder Hardware selbst gebunden, sondern an die Rolle, die diese übernimmt, wenn sie das System auf diese Weise benutzt (vgl. 6.2.1 S. 41).

## **Akzidentien**

Sammelbegriff für Vorzeichen und Auflösungszeichen.

## **Analytisches Modellieren (engl. analysis modeling)**

Eine Phase der objektorientierten Entwicklung (vgl. 5.3.2, S.35). Das analytische Modellieren besteht im wesentlichen darin, eine Spezifikation zu erstellen.

**Attribut**

Ein Datenbehältnis in einem Objekt, in dem ein Teil des Zustands oder des Wissens dieses Objekts festgehalten wird. Ein Attribut wird rein semantisch betrachtet und noch nicht mit einer konkreten Datenstruktur in Verbindung gebracht. Über die Datenstruktur eines Attributs gibt dessen Attributtyp Aufschluß.

**Attributtyp**

Der Typ eines Attributs (vgl. *Attribut*, *Typ*).

**Auflösungszeichen**

Ein Auflösungszeichen hebt die Funktion eines bzw. mehrerer Vorzeichen auf.

**Balken**

Gruppierung von Noten, mittels eines oder mehrerer horizontaler Striche, den *Balken*, die der besseren Lesbarkeit der Noten dient. Die Anzahl der Striche hängt von den Notenwerten der Noten der Balkengruppe ab. Dabei wird zwischen zwei Balkentypen unterschieden: *schräge* Balken und *gerade* Balken. Bei schrägen Balken sind zwei Richtungen möglich: *aufsteigend* und *absteigend*.

**Bezifferung**

Zu einer Note können eine oder mehrere Bezifferungen gehören, die über den harmonischen Kontext Aufschluß geben, in dem diese Note zu sehen ist. Eine Bezifferung besteht aus bestimmten Ziffern und Zeichen, die unterhalb eines Liniensystems unter der Note, auf die sie sich beziehen, geschrieben werden. Wenn zu einer Note mehrere Bezifferungen gehören, so wechselt der harmonische Kontext der Note noch während diese andauert, d.h. die Dauer der Gültigkeit einer Bezifferung muß nicht mit der Dauer der Note, auf die sie sich bezieht, identisch sein. Die Bezifferungen werden in diesem Fall jeweils an der Taktposition unterhalb der Note geschrieben, ab der sie gelten.

**Beziehungstyp**

Der Typ einer Beziehung (vgl. *Typ*).

**Controller**

Ein Controller hat im Model-View-Controller-Kontext die Aufgabe, Benutzereingaben, die das zugehörige Modell betreffen, in Aufrufe an das Modell umzusetzen. Wenn die Benutzereingaben nicht das Modell, sondern eine oder mehrere Views davon betreffen, so setzt der Controller die Benutzereingaben in Aufrufe an die betroffenen Views um (vgl. *Model-View-Controller-Konzept*, sowie 3.1, S. 15ff).

**Dynamisches Modell**

In einem dynamischen Modell werden die dynamischen Abläufe in einem System beschrieben. Ein Beispiel dafür sind Interaktionsdiagramme (vgl. dort).

**Entwurfsbezogenes Modellieren** (engl. *design modeling*)

Eine Phase der objektorientierten Entwicklung (vgl. 5.3.2, S.35). Das entwurfsbezogene Modellieren besteht im wesentlichen darin, einen implementierungsunabhängigen Entwurf zu erstellen. In diesem Zuge werden auch strukturelle Maßnahmen zur Verbesserung der Produkt- und der Gebrauchsqualität getroffen.

**Entwurfskomponente** (engl. *design component*)

Eine Zusammenfassung von Objekttypen, die eine einzelne Funktionseinheit bilden. Wird oft synonym zu Subsystem gebraucht (vgl. *Subsystem*).

(„*This is a collection of object types which create a single unit of functionality*“ - Hutt/OMG 1994, S. 197).

**Entwurfsmuster** (engl. *design pattern*)

Ein Entwurfsmuster ist ein Muster auf einer Entwurfs-ebene. (vgl. *Muster*, sowie 6.2.2., S. 44ff).

**Ergänzender Use-Case**

Ein Use-Case, der einen anderen Use-Case erweitert (vgl. *Use-Case*, sowie 6.2.1, S. 42).

**Generalbaß**

Ein Generalbaß diente im Barockzeitalter (ca. 1600-1740) hauptsächlich als harmonisches Grundgerüst für Improvisationen, anhand dessen auch musiktheoretische Kenntnisse vermittelt wurden. Ein Generalbaß besteht aus einer Folge von Noten für ein Baßinstrument. Wenn für diese Noten eine Begleitung gemacht wird, so nennt man das „den Generalbaß aussetzen“. Dabei müssen zu jeder Note eines Generalbasses üblicherweise drei weitere Noten als Begleitung gefunden werden, um so insgesamt einen vierstimmigen Satz zu erhalten. Es gab damals eine feste Regel dafür, wie einzelne Baßnoten innerhalb einer Tonart zu verstehen waren. Bei Ausnahmen zu dieser Regel wurde die entsprechende Baßnote mit einer sogenannten „Bezifferung“ versehen, die darüber Auskunft gab, wie diese Baßnote zu interpretieren war, d.h. in welchem harmonischen Kontext diese zu sehen war.

**Generischer Objekttyp**

Synonym für parametrisierten Objekttyp (vgl. dort).

**Hilfslinien**

Notenlinien, die sich außerhalb eines Systems befinden.

**Implementation**

Das Ergebnis der Implementierung.

**Implementierung**

Unter Implementierung versteht man die Erstellung eines Programms oder Programmteils. Eine Implementierung ist auch eine Phase der objektorientierten Entwicklung (vgl. 5.3.2, S. 35).

**Implementierungsbezogenes Modellieren**

(engl. *implementation modeling*)

Eine Phase der objektorientierten Entwicklung (vgl. 5.3.2, S. 35). Das implementierungsbezogene Modellieren besteht darin, den implementierungsunabhängigen Entwurf an implementierungsabhängige Gegebenheiten anzupassen, d.h. an eine bestimmte Plattform, die verwendete Programmiersprache, die zum Einsatz kommende Klassenbibliothek, Datenbank, etc.

**Improvisation**

„Das gleichzeitige Erfinden und klangliche Realisieren von Musik. ...“ (Brockhaus/ Riemann, 1995).

**Improvisieren**

Das Ausführen einer Improvisation.

**Instanz** (engl. *instance*)

In der OMG-Terminologie ist Instanz ein Synonym für Objekt (vgl. dort).

**Instanziieren**

- 1) Eine Instanz erzeugen.
- 2) Bei einem parametrisierten Objekttyp werden dessen formalen Parameter durch tatsächliche Parameter ersetzt.

**Instanziierter Objekttyp**

Konkrete Ausprägung eines parametrisierten Objekttyps (vgl. *parametrisierter Objekttyp*). Wenn bei einem parametrisierten Objekttyp die formalen Parameter durch tatsächliche Parameter ersetzt wurden, liegt ein instanzierter Objekttyp vor.

**Interaktionsdiagramm**

Mit Hilfe eines Interaktionsdiagramms werden die dynamischen Abläufe in einem System dargestellt (vgl. 6.2.3, S. 45ff).

**Iteration**

Im Kontext einer Entwicklung ist eine Iteration ein einzelner Durchlauf aller Entwicklungsphasen (vgl. 5.3.2, S. 35, sowie 5.3.3.2, S. 37).

**Iterationsstrategie** (engl. *iteration strategy*)

Durch eine Iterationsstrategie wird vorgegeben, wie oft die einzelnen Phasen durchlaufen werden und welche Qualität die Ergebnisse der einzelnen Iterationen jeweils besitzen sollen (vgl. 5.3.3.2, S. 37).

**Kategorie**

Eine Menge von Objekttypen bzw. Operationen und/oder Kategorien, die in einem logischen Zusammenhang stehen. Das können beispielsweise alle Interface-Objekttypen oder alle Zugriffsoperationen auf Attribute sein. Kategorien werden teilweise synonym zu Subsystem definiert (vgl. *Subsystem*). Es sollte aber besser zwischen den beiden Termini unterschieden werden, denn ein Subsystem läßt sich immer als ein solches verwirklichen, eine Kategorie nicht unbedingt, obwohl beide oft deckungsgleich sind.

**Klasse** (engl. *class*)

Eine Implementation eines Objekttyps (vgl. *Implementation*, *Objekttyp*).

(„*A class is a description of a group of objects (instances) with similar properties, and common behavior, semantics, and relationships. A class represents a particular implementation of an object type.* ...“ - Hutt/OMG 1994, S. 203).

**Liniensystem**

Ein graphisches Bezugssystem für Noten. Ein herkömmliches Liniensystem besteht aus fünf Notenlinien und vier Zwischenräumen zwischen diesen Notenlinien.

**Meilensteinstrategie** (engl. *checkpoint strategy*)

Durch eine Meilensteinstrategie wird festgelegt, wie der Umgang mit Meilensteinen gehandhabt wird (vgl. 5.3.3.4, S. 39).

**Metrum**

Rhythmischer Grundschatz. Eine Note, deren Notenwert dem Taktma entspricht, hat die absolute Dauer von einem Grundschatz.

*Beispiel: Eine 4tel-Note in einem 3/4-Takt besitzt die absolute Dauer von einem Grundschatz. Ein 3/4-Takt hat demnach drei Grundschatzschlge.*

**MIDI** (engl. *Musical Instrument Digital Interface*)

Standard-Schnittstellenprotokoll der Musikindustrie fr den Datenaustausch zwischen Musikinstrumenten, Effektgerten, Musikprogrammen, etc. Darin sind die Hardware-Schnittstelle, die Daten-Formate und die bertragungsprotokolle definiert.

**MIDI-Daten**

Daten, die in einem MIDI-Format vorliegen (vgl. *MIDI*).

**MIDI-Standardfile**

Eine Datei in einem Standard-Dateiformat fr den Austausch von MIDI-Daten (vgl. *MIDI-Daten*, *MIDI*).

**Model-View-Controller-Konzept**

Das Ziel des Model-View-Controller-Konzept ist es, die Unabhngigkeit der Daten eines Programms von dessen Benutzeroberflche zu erreichen. Dazu wird innerhalb eines Programms eine Aufgabenteilung in ein Modell, eine oder mehrere Views und einen oder mehrere Controller vorgenommen. Fr diese Aufgabenteilung gibt es inzwischen zahlreiche Varianten, aber das Prinzip, das dahinter steckt, ist immer dasselbe geblieben. Ursprnglich stammt es aus der Klassenbibliothek der Programmiersprache Smalltalk und ist inzwischen eines der bewhrtesten Entwurfsmuster (vgl. *Entwurfsmuster*, sowie 3.1, S. 15ff).

**MUNOS**

Name der Bibliothek, die im Rahmen dieser Diplomarbeit entstanden ist. MUNOS ermöglicht anderen Programmen die graphische Darstellung von Noten am Bildschirm. MUNOS steht für ***Music Notation on Screen***. Der einfacheren Lesbarkeit wegen, wird der Name in Kapitälchen geschrieben.

**Muster** (engl. *pattern*)

Ein Muster beschreibt ein bestimmtes, in einem gegebenen Kontext immer wiederkehrendes Problem, sowie ein vordefiniertes Schema zu seiner Lösung (vgl. 6.2.2, S. 44ff).

**Note**

Graphische Repräsentation eines Tones.

**Notensatzprogramm**

Ein Notensatzprogramm ist eine Art Textverarbeitungsprogramm für Noten. Reine Notensatzprogramme sind eher selten, denn meist verfügen diese noch zusätzlich über Sequenzer-Funktionen (vgl. *Sequenzer*). Andererseits wurden die meisten Sequenzerprogramme im Laufe der Zeit um die Fähigkeit zum Notensatz erweitert. Deshalb ist der Übergang vom Notensatzprogramm zum Sequenzerprogramm fließend.

**Notenschlüssel**

Ein Notenschlüssel gibt darüber Auskunft, wie die Tonhöhen der Noten in einem Liniensystem zu interpretieren sind, d.h. welche Notenlinie bzw. welcher Zwischenraum welche Tonhöhe bedeutet.

## Notenwert

Der Notenwert einer Note gibt Auskunft über deren relative Dauer, d.h. deren Dauer im Bezug auf andere Notenwerte und damit auf die anderen Noten. Die absolute Dauer einer Note ergibt sich aus deren relativer Dauer in Abhängigkeit vom Metrum (vgl. Metrum).

*Beispiel: Eine Halbe ist normalerweise halb so lang wie eine Ganze, unabhängig davon, in welchem Tempo ein Stück gespielt wird. Das Tempo, in dem eine Stück gespielt wird, ist nur für die absoluten Dauern relevant.*

## N-Tole

Eine N-Tole wird durch ein Dauernverhältnis  $N:M$  und einen Notenwert, auf den sich dieses Dauernverhältnis bezieht, definiert, d.h.  $N$  Noten eines bestimmten Notenwerts dauern im Fall der N-Tole so lange wie sonst  $M$  Noten desselben Notenwerts unter normalen Umständen dauern würden. Üblicherweise ist  $N$  größer als  $M$ .

*Beispiel: Eine 4tel-Triole besteht aus drei 4tel-Noten mit dem Dauernverhältnis 3:2, d.h. drei 4tel-Noten dauern in diesem Fall so lange wie normalerweise zwei 4tel-Noten dauern würden. Die Gesamtdauer einer 4tel-Triole entspricht demnach der Dauer einer halben Note. Dieser Zeitraum wird im Falle der 4tel-Triole auf drei Noten aufgeteilt. Die Dauer einer einzelnen 4tel-Note innerhalb einer 4tel-Triole, entspricht demnach  $1/3$  mal der Dauer einer halben Note.*

## Object Management Group

Eine Gruppierung, die es sich zum Ziel gemacht hat, möglichst rasch Standards für den objektorientierten Bereich zu entwickeln. Deren zur Zeit bekanntester Standard ist CORBA („**The Common Object Request Broker Architecture**“), in dem die Architektur für objektorientierte Programme vorgegeben wird, die auf verteilten Systemen zusammenarbeiten sollen und zwar unabhängig von Programmiersprachen, Betriebssystemen und Netzwerkstandards.

**Objekt** (engl. *object*)

Ein Objekt ist eine Sache. Es wird als eine konkrete Ausprägung eines Objekttyps erzeugt. Jedes Objekt besitzt eine eigene Identität. Auch wenn zwei Objekte in ihren Eigenschaften völlig identisch sind, besitzt doch jedes eine eigene Identität. Diese ist also nicht von den charakteristischen Eigenschaften eines Objekts abhängig und unterscheidet sich von jeder dieser Eigenschaften. Jedes Objekt stellt ein oder mehrere Operationen zur Verfügung.

*(„An object is a thing. It is created as the instance of an object type. Each object has a unique identity that is distinct from and independent of any of its characteristics. Each object offers one or more operations.“ - Hutt/OMG 1994, S.165).*

**Objekttyp** (engl. *object type*)

Die Definition einer Menge von Objekten, die ein ähnliches Verhalten aufweisen. Ein Typ ist eine semantische Eigenschaft (vgl. *Typ*). In der OMG-Terminologie wird hier zwischen der semantischen Definition (*Objekttyp*), die häufig ebenfalls als Klasse bezeichnet wird, und deren syntaktischer Repräsentation innerhalb eines Programms (*Klasse*) unterschieden (vgl. *Klasse*).

*(„An object type represents a definition of some set of object instances with similar behavior. A type is a semantic property. Characteristics of object type are: its definition, any extension of its definition, its supertypes and its subtypes. ...“ - Hutt/OMG 1994, S. 167).*

**OMG**

OMG steht für *Object Management Group* (vgl. dort).

**Operation** (engl. *operation*)

Funktion, die von einem Objekt zur Verfügung gestellt wird.

(„*Objects support operations which may take one or more arguments, may return a result, may cause a change of state, and may trigger exceptions. An operation has a signature which contains: operation name, argument types, results types, and optionally a definition of semantics (e.g., the effect on object state).*“)- Hutt/OMG 1994, S.168).

**Parametrischer Objekttyp**

Ein parametrischer Objekttyp dient als Vorlage (engl. *template*) für andere Objekttypen. Um daraus einen Objekttyp zu instanziiieren werden die formalen Parameter des Objekttyps durch tatsächliche Parameter ersetzt (vgl. *instanziiieren, instanziiierter Objekttyp*).

**Parametrisieren**

Mit formalen Parametern versehen (vgl. *instanziiieren*).

**Pause**

„...das zeitweilige Schweigen einzelner oder aller Stimmen eines Satzes. ...“ (Brockhaus/Riemann, 1995). Wenn zum Beispiel ein Instrument während eines Musikstücks gerade nicht spielt, so hat es Pause. Wird häufig synonym zu *Pausenzeichen* (vgl. dort) verwendet.

**Pausenwert**

Der Begriff des Pausenwerts wird analog dem des Notenwerts gebraucht. Auch die Dauern sind identisch, d.h. die Dauer einer halben Pause entspricht immer der Dauer einer halben Note. Die einzige Ausnahme bildet in manchen Taktarten (2/4, 3/4, 7/4, 3/8, 6/8 und 9/8 -Takt) die ganze Pause, die dort aus dem relativen Gefüge herausgelöst ist und immer einen ganzen Takt lang Pause bedeutet, statt der Gesamtdauer zweier halber Pausen zu entsprechen.

**Pausenzeichen**

Graphische Repräsentation einer Pause.

**Persistentes Objekt**

Ein Objekt, dessen Lebensdauer über die Lebensdauer seines Erzeugers hinausgehen kann.

**Phasenübergreifende Konzepte**

Die Ausgangsbasis einer objektorientierten Entwicklung bilden die phasenübergreifenden Konzepte einer Methode, die im Referenzmodell der OMG unter „*object modeling*“ zusammengefaßt werden (vgl. 5.3.1, S. 34ff). Dazu gehören einerseits die Konzepte, die eine Methode bietet, um Objekte zu modellieren, wie z.B. die Beziehungen, die zwischen Objekten ausgedrückt werden können oder die Kommunikationsarten, die zwischen Objekten aufgezeigt werden können und eventuelle graphische Ausdrucksmittel dafür, d.h. eine Notation. Andererseits gehören dazu auch die Konzepte einer Methode, um die Analyse und den Entwurf innerhalb der Methode miteinander zu verbinden. Das können Transformationskonzepte für Modelle sein oder noch weiter reichende Konzepte, mit deren Hilfe ein roter Faden durch eine gesamte Entwicklung gebildet werden kann, wie z.B. das Use-Case-Konzept, das in diesem Projekt verwendet wurde.

**Point**

Angelsächsische typographische Maßeinheit im DTP-Bereich, die u.a. auch bei PostScript verwendet wird:  
1 point = 1/72 inch.

**Primärbalken**

Der am weitesten außen liegende Balken, d.h. der Balken, der von den Noten am weitesten weg ist. Wenn nur ein Balken vorhanden ist, so ist das der Primärbalken.

**Progressionsstrategie** (engl. *progression strategy*)

Durch eine Progressionsstrategie wird festgelegt, wie die Ergebnisse der einzelnen Entwicklungsphasen generell zusammenhängen (vgl. 5.3.3.1, S. 36).

**Punkt**

Verlängerung der Dauer eines Notenwerts bzw. Pausenwerts um jeweils die Hälfte.

**Satz**

„...allgemein als »das (mehrstimmig) Gesetzte«...“ (Brockhaus/Riemann, 1995). Ein Beispiel für einen vierstimmigen Satz ist ein Generalbaß (vgl. dort).

**Schlag**

Damit ist ein Grundschatz gemeint (vgl. *Metrum*).

**Sekundärbalken**

Ein Balken, der nicht Primärbalken ist (vgl. *Primärbalken*).

**Sequenzzer**

Eine Art Tonbandmaschine für MIDI-Daten. Die meisten Sequenzerprogramme verfügen darüber hinaus noch über Notensatz-Fähigkeiten (vgl. *Notensatzprogramm*).

**Statisches Diagramm**

Mit Hilfe eines statischen Diagramms werden die Strukturen und Beziehungen innerhalb eines Systems dargestellt (vgl. 6.2.3, S. 45ff).

**Statisches Modell**

In einem statischen Modell werden die Strukturen und Beziehungen innerhalb eines Systems beschrieben. Ein Beispiel dafür sind statische Diagramme (vgl. dort).

**Stimme**

Traditionell: „... horizontaler Verlauf von Tönen und Konstituens der Zusammenklänge ...“ (Brockhaus/Riemann, 1995). Es gibt grundsätzlich zwei Betrachtungsweisen. Vertikal gesehen bilden mehrere Töne, die gleichzeitig erklingen, einen Zusammenklang. Horizontal gesehen gehört jeder Ton eines Zusammenklangs zu einer nacheinander erklingenden Folge von Tönen, also einer Stimme. Deshalb ist eine Stimme ein konstitutiver Bestandteil eines Zusammenklangs, d.h. ein Zusammenklang setzt sich aus mehreren Tönen zusammen, von denen jeder zu einer anderen Stimme gehört.

In MUNOS ist eine Stimme als eine beliebige, zeitlich geordnete, Menge von Tönen und Pausen definiert, die im Gegensatz zur üblichen Auffassung von Stimme nicht durchgängig und zu anderen Stimmen disjunkt sein muß. Es kann also eine Stimme an manchen Taktstellen auch Lücken aufweisen und ein Ton bzw. eine Pause kann durchaus mehreren Stimmen gleichzeitig angehören.

**Strategisches Modellieren** (engl. *strategic modeling*)

Eine Phase der objektorientierten Entwicklung (vgl. 5.3.2, S. 35). Das strategische Modellieren besteht im wesentlichen darin, ein Projekt zu planen.

**Subsystem**

Eine Menge von Objekttypen und/oder Subsystemen, die in einem funktionalen Zusammenhang stehen. Wird synonym zu Entwurfskomponente gebraucht (vgl. *Entwurfskomponente* ).

**Subtyp**

Ein Objekttyp an den Eigenschaften vererbt werden.

**Supertyp**

Ein Objekttyp von dem Eigenschaften geerbt werden.

**System**

Im musikalischen Kontext ist System ein Synonym für Liniensystem (vgl. *Liniensystem*).

**Taktart**

Die Taktart wird in Form eines Bruches angegeben. Im Zähler steht die Anzahl von Noten eines bestimmten Notenwerts, die einen ganzen Takt ausfüllen. Im Nenner steht das Taktmaß, das angibt auf welchen trivialen Notenwert sich der Zähler bezieht.

**Taktmaß**

Der Nenner der Taktart (vgl. *Taktart*).

**Ton**

„...bezeichnet im heutigen Sprachgebrauch die Kategorie T. als Element der Musik; im Vordergrund steht die Bedeutung Einzelton. ...“ (Brockhaus/Riemann, 1995). Wenn zum Beispiel ein Trompeter auf seiner Trompete spielt, so gibt dabei die Trompete einzelne Töne von sich.

**Triviale Note**

Eine Note, die einen trivialen Notenwert besitzt.

**Triviale Pause**

Eine Pause, die einen trivialen Pausenwert besitzt.

**Trivialer Notenwert**

Eine trivialer Notenwert ist:

eine doppelte Ganze, eine Ganze, eine Halbe, eine Viertel, eine Achtel, eine Sechzehntel, eine Zweiunddreißigstel, eine Vierundsechzigstel oder eine Hundertachtundzwanzigstel.

**Trivialer Pausenwert**

Eine trivialer Pausenwert ist:

eine doppelte Ganze, eine Ganze, eine Halbe, eine Viertel, eine Achtel, eine Sechzehntel, eine Zweiunddreißigstel, eine Vierundsechzigstel oder eine Hundertachtundzwanzigstel.

**Typ**

Durch einen Typ wird für eine beliebige Menge eine semantische Eigenschaft festgelegt, über die dann alle Elemente verfügen, die zu dieser Menge gehören.

**Use-Case** (eigentlich *use case*)

In einem Use-Case werden ähnlich gelagerte Anwendungsfälle beschrieben. Ein Use-Case kann als Objekttyp für Anwendungsfälle gesehen werden (vgl. 6.2.1, S. 41ff).

**Verpackungsstrategie** (engl. *packaging strategy*)

Durch eine Verpackungsstrategie wird vorgegeben, wie die einzelnen Tätigkeiten einer Entwicklung in die einzelnen Entwicklungsphasen „verpackt“ werden, d.h. wie die Zuordnung der einzelnen Tätigkeiten auf die Entwicklungsphasen gehandhabt wird (vgl. 5.3.3.3, S. 38). Eine Verpackungsstrategie ist eine organisatorische Maßnahme des Managements und deshalb meist von den äußeren Randbedingungen eines Projekts abhängig. Beispielsweise kann die Verpackungsstrategie davon abhängen, was wann wo verfügbar ist.

**Vertrieb** (engl. *delivery*)

Eine Phase der objektorientierten Entwicklung (vgl. 5.3.2, S. 35). In dieser Phase werden erste Testversionen verteilt bzw. wird das fertige Produkt ausgeliefert.

**Verwendeter Use-Case**

Ein Use-Case, der von einem anderen Use-Case benutzt wird (vgl. *Use-Case*, sowie 6.2.1, S. 42).

**View**

Im Model-View-Controller-Kontext erzeugt eine View eine Darstellung eines Modells, bei der das Modell auf eine ganz bestimmte Art und Weise interpretiert wird. Die Interpretation äußert sich darin, was von dem Modell wie dargestellt wird. Oft wird durch eine View nur ein bestimmter Teil des Modells dargestellt, der einem ganz bestimmten Gesichtspunkt entspricht. Verschiedene Views auf ein Modell ermöglichen so, das Modell aus verschiedenen Blickwinkeln heraus zu betrachten (vgl. *Model-View-Controller-Konzept*, sowie Kapitel 3.1, S. 15ff).

**Vorzeichen**

Eine Vorzeichen ändert die Tonhöhe einer Note. Ein *b* erniedrigt eine Note um jeweils einen Halbton, ein *#* erhöht eine Note um jeweils einen Halbton.

**Warnungsakzidens**

Ein Akzidens, das nur zur Erinnerung angebracht wird. Ein Warnungsakzidens wird deshalb oft in Klammern geschrieben.

**Zwischenraum**

Der Zwischenraum zwischen zwei Notenlinien eines Liniensystems.

## B Literatur

- ANSI/IEEE (1984): *An American National Standard: IEEE Guide to Software Requirements Specifications*. **ANSI/IEEE Std 830-1984**.
- Apple (1994): *Inside Macintosh: QuickDraw GX **Environment and Utilities***. Addison-Wesley, Reading, Massachusetts.
- Apple (1994): *Inside Macintosh: QuickDraw GX **Graphics***. Addison-Wesley, Reading, Massachusetts.
- Apple (1994): *Inside Macintosh: QuickDraw GX **Objects***. Addison-Wesley, Reading, Massachusetts.
- Apple (1994): *Inside Macintosh: QuickDraw GX **Typography***. Addison-Wesley, Reading, Massachusetts.
- Booch G. (1994): ***Object-Oriented Analysis and Design with Applications***. Benjamin/ Cummings, Redwood City, California, 2nd Ed.
- Brockhaus, Riemann (1995): ***Brockhaus Riemann Musiklexikon***, Schott, Mainz, und Piper, München, 2.Auflage.
- Deining M., Lichte H., Ludewig J., Schneider K. (1993): ***Studien-Arbeiten – ein Leitfadens zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik***. vdf, Zürich, und Teubner, Stuttgart, 2.Aufl.
- Gamma E., Helm R., Johnson R., Vlissides J. (1995): ***Design Patterns – Elements of Reusable Object-Oriented Software***. Addison-Wesley, Reading, Massachusetts.
- Hutt A./OMG (1994): ***Object Analysis and Design – Comparison of Methods***. Wiley-QED, New York.
- Jacobson I., Christerson M., Jonsson P., Övergaard G. (1993): ***Object-Oriented Software Engineering – A Use Case Driven Approach***. Addison-Wesley, Reading, Massachusetts Revised Printing.
- Jacobson I., Christerson M. (1995): *Modeling With Use Cases. A growing consensus on use cases*. **Journal of Object-Oriented Programming, March-April**, S. 15-19.
- Papurt D. M. (1995): ***Inside the Object Model – The Sensible Use of C++***. SIGS, New York.

- Parker R.C. (1995): **Looking Good In Print** – Grundlagen der Gestaltung für Desktop Publishing. (An europäische Verhältnisse angepaßte deutsche Fassung des amerikanischen Originals von 1990) Midas, St. Gallen, Schweiz
- Roemer C. (1973): **The Art Of Music Copying** – The Preperation of Music for Performance. Roeric Music Co., Sherman Oaks, California
- Rumbaugh J. (1994): *Modeling & Design. Getting started: Using use cases to capture requirements.* **Journal of Object-Oriented Programming, September**, S. 9-12.
- Schneider Wolf (1991): **Deutsch Für Profis** – Wege zu gutem Stil. Goldmann / Stern-Bücher, Gruner & Jahr, Hamburg, 9. Aufl.
- Sommerville I. (1992): **Software Engineering**. Addison-Wesley, Wokingham, England, 4th Ed.

## **C Erklärung zum selbständigen Arbeiten**

Ich versichere, diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet zu haben.

(Christoph Weiß)

Ein Musterexemplar liegt bei der die Arbeiten entgegennehmenden Stelle zur Einsicht aus.