View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Helsingin yliopiston digitaalinen arkisto

Kai Lassfolk

Music Notation as Objects

An Object-Oriented Analysis of the Common Western Music Notation System

Academic dissertation to be publicly discussed, by due permission of the Faculty of Arts at the University of Helsinki in Auditorium XIV, on the 20th of November, 2004 at 10 o'clock.

Music Notation as Objects

Acta Semiotica Fennica
Approaches to Musical Semiotics

Editor
Eero Tarasti
Associate Editors
Paul Forsell
Richard Littlefield

Editorial Board (ASF)
Honorary Member:
Thomas A. Sebeok †

Pertti Ahonen
Henry Broms
Jacques Fontanille
André Helbo
Altti Kuusamo
Ilkka Niiniluoto
Pekka Pesonen
Hannu Riikonen
Kari Salosaari
Sinikka Tuohimaa
Vilmos Voigt

Editorial Board (AMS)
Daniel Charles
Márta Grabócz
Robert S. Hatten
Jean-Marie Jacono
Costin Miereanu
Raymond Monelle
Charles Rosen
Gino Stefani
Ivanka Stoianova

Music Notation as Objects

An Object-Oriented Analysis of the Common Western Music Notation System

Kai Lassfolk

Acta Semiotica Fennica XIX
Approaches to Musical Semiotics 7
Studia Musicologica Universitatis Helsingiensis XI

International Semiotics Institute at Imatra
Semiotic Society of Finland
Department of Musicology, University of Helsinki
2004

This book is a publication of

The International Semiotics Institute

http://www.isisemiotics.fi/

Telephone orders +358 5 681 6639
Fax orders +358 5 681 6628
E-mail orders maija.rossi@isisemiotics.fi

Copyright © 2004 Kai Lassfolk
All Rights Reserved

Printed by Hakapaino, Helsinki 2004

ISBN 952-5431-07-X
ISSN 1235-497X Acta Semiotica Fennica XIX
ISSN 1458-4921 Approaches to Musical Semiotics 7
ISSN 0787-4294 STUDIA MUSICOLOGICA UNIVERSITATIS
HELSINGIENSIS XI

Acknowledgments

It took me more than ten years to complete this study. During that time I received help and encouragement from an enormous number of people, to all of whom I am deeply indebted. Here, I wish to mention a few of them as well as the institutions, whose help was of utmost importance to the completion of this book.

There are two persons to whom I will be forever grateful, but who unfortunately passed away before I could complete the task: my mother Laila, who spared no effort whenever I needed help of any kind, and Erkki Salmenhaara, who was my original supervisor and for whom I hold the greatest respect.

I want to thank my final supervisors, Eero Tarasti and Alfonso Padilla, for all their help and encouragement throughout my studies and this research project. I also thank Kimmo Iltanen, my collaborator at the start of this dissertation project. His input was crucial at the initial stages of this study. I would also like to thank all my colleagues at the Department of Musicology, University of Helsinki for their support. In particular, I wish to thank Erkki Pekkilä, Erja Hannula, Merja Hottinen, Jaakko Tuohiniemi, and Irma Vierimaa. Many thanks are also due to Anna Pienimäki and Esa Lilja for proofreading my texts, to Richard Littlefield for improving the English language of this text, and to Paul Forsell for his help at the final hectic stages of preparing this book for printing.

I am grateful to Andrew Bentley, Paavo Heininen, and Yrjö Hjelt, who granted valuable interviews to Kimmo and me at the early stages of our research. Andrew Bentley also deserves my warmest thanks for his advice and help at various stages of this project. I thank Eleanor Selfridge-Field, Vesa Välimäki, and Mika Kuuskankare for kindly providing me with valuable reference material and Jean-Baptiste Barrière for his generous advice. I also thank my closest col-

leagues, Pauli Laine, Mikko Ojanen, Jaska Uimonen, Jukka-Pekka Kervinen, and Kalev Tiits for all their support and friendship.

The person who has had the greatest influence on this study is Timo Lehtinen. Countless conversations and brainstorming sessions with him during the last nearly twenty years have been a tremendous source of inspiration for me. Many decisions made during this research project had their origins in these discussions. In particular, it was Timo, who first brought linear logic to my attention. Also, he was the first to suggest the idea of presenting an analogy between a linear object system and a hierarchical computer file system. Timo, I owe you my deepest gratitude.

I want to thank my wife Hellevi and my sons Elias and Salomo for creating an inspiring atmosphere at home. Many thanks are also due to my father Lenni for constantly reminding me to keep on working with this study.

Finally, I thank the Finnish Cultural Foundation, the Niilo Helander foundation, and University of Helsinki for funding this study.

Espoo and Helsinki, October 2004

Kai Lassfolk

Contents

Acknowledgments v
List of Figures xiii

Chapter 1: Introduction 1

1.1 Computer-based music notation 2
1.2 Research methods 3
1.3 Application of methodology and general goals 4
1.4 Results 5
1.5 Overview of contents 6

Chapter 2: Computer-based music notation 9

2.1 Common Music Notation 9
2.2 Graphical aspects of music notation 11
2.3 Dynamic and evolutionary aspects 11
2.4 Computer applications 12
2.5 Capabilities of music notation programs 14
2.6 Types of information 17
2.7 Rules and ambiguities 20
2.8 Music notation and computer graphics 22
2.9 Music notation terminology 23
2.10 Some conclusions 25

Chapter 3: Computer-based musical data representation systems 27

3.1 Music representation 28
3.2 Design requirements 30
3.3 Music notation as a representation 33
3.4 Digital audio signals 34
3.5 The Music V sound synthesis language 35
3.6 MIDI and MIDI Files 38
3.7 MusicKit 39
3.8 The SCORE music notation program 41
3.9 Lime and Tilia 45
3.10 The Music preprocessor 46
3.11 NIFF: Notation Interchange File Format 47
3.12 Music notation markup languages 49
3.13 Object-based music representations 51
3.14 General-purpose graphical representations 51
3.15 Comments on data representation systems 52

Chapter 4: Object-oriented software engineering 55

4.1 Basic concepts and terminology 56
4.2 Object-oriented programming languages 59
4.3 Object-oriented software engineering methods 61
4.3.1 The Coad & Yourdon OOA method 62
4.3.2 The Booch/ROSE method 65
4.3.3 The Object Modeling Technique 68
4.3.4 The Unified Modeling Language 70
4.4 UML principles and terminology 71
4.5 UML class and object diagrams 72
4.5.1 Class and its properties 73

4.5.2 Association and aggregation 74
4.5.3 Inheritance 76
4.5.4 Other relationships and features 77
4.5.5 Object diagrams 78
4.6 Text representation of object systems 79
4.7 Object identification and classification techniques 79
4.8 Summary and discussion 80

Chapter 5: Refining the methodology with linear logic 83

5.1 Linear logic 84
5.2 Computational applications 85
5.3 Considerations 86
5.4 Application in object-oriented analysis 88
5.5 Formal rules for a linear object system 89
5.6 Implications for UML notation 90
5.7 Toward a systematic analysis process 91

Chapter 6: Analysis principles for music notation 93

6.1 Existing systems 93
6.1.1 Sound Processing Kit 94
6.1.2 Adobe Illustrator 97
6.1.3 SCORE 98
6.1.4 Tilia and MusicXML 100
6.1.5 Object-oriented representations of music notation 101
6.2 Basic criteria for a new object representation 103
6.3 Categorization principles 107
6.4 Scope of the analysis 109
6.5 Application of linear logic 110

6.6 General requirements 110
6.7 Preliminary examples 112

Chapter 7: The analysis model 115

7.1 General definitions 115
7.2 The CMNSymbol class 116
7.3 The top-level aggregation structure 118
7.4 Staff 121
7.5 DurationalSymbol 122
7.6 Note 123
7.7 Environment modifiers 125
7.8 Attachments 127
7.8.1 Connector 128
7.8.2 Marks 129
7.9 Beams 130

Chapter 8: Object diagram examples 133

8.1 Systems and Staves 133
8.2 CoreSymbols 133
8.3 Chords and Stems 136
8.4 Beams 136
8.5 TremoloBeams 137
8.6 Ties and Slurs 139
8.7 An XML example 140

Chapter 9: Discussion 145

9.1 Commentary on the analysis 145
9.2 A low-level graphics system 146
9.3 Notes, stems and chords 148
9.4 Beams 149
9.5 Design issues 150
9.6 Implementation issues 152
9.7 Score processing and dynamic behavior 153
9.8 Logical and performance information 156
9.9 Macro statements 157
9.10 Extendibility of the model 158
9.11 Representational aspects 158

Chapter 10: Conclusions 161

References 165

List of Figures

Figure 2-1: A simple expression using common music notation 18
Figure 3-1: An example score 37
Figure 4-1: Class 73
Figure 4-2: Association 74
Figure 4-3: Aggregation 75
Figure 4-4: Aggregation with a multiplicity adornment 75
Figure 4-5: Association with role adornments 76
Figure 4-6: Composition 76
Figure 4-7: Inheritance 77
Figure 4-8: Multiple inheritance 77
Figure 4-9: Object diagram 78
Figure 6-1: Signal flow diagram of a Schroeder reverberator 95
Figure 6-2: UML class diagram of the SPKit Schroeder reverberator 96
Figure 6-3: SCORE's inheritation structure 99
Figure 6-4: Examples of tied notes 113
Figure 7-1: The CMNSymbol class 117
Figure 7-2: Top-level aggregation structure 119
Figure 7-3: System-level structure 120
Figure 7-4: Staff-level structure 122
Figure 7-5: DurationalSymbol class structure 123
Figure 7-6: Note class structure 124
Figure 7-7: Environment modifiers 126
Figure 7-8: Types of barlines 127
Figure 7-9: Connectors 128

Figure 7-10: Marks 130
Figure 7-11: Class diagram of Beam and its subclasses 131
Figure 8-1: Example of a system with staves and grand staff 134
Figure 8-2: A two-bar note example and equivalent object diagram 135
Figure 8-3: Sample chord 136
Figure 8-4: Example of two beamed notes 137
Figure 8-5: Examples of tremolo beams 138
Figure 8-6: Examples of tied notes 139
Figure 8-7: Examples of tied notes 140
Figure 9-1: An example low-level graphics model 147
Figure 9-2: A simplified top-level aggregation structure 151
Figure 9-3: Sample sequence diagram: Deletion of a note from a staff 156

Chapter 1
Introduction

Western music notation has played an important role in preserving musical works created over several hundred years. Although electronic and computer technology has provided new means of storing musical information, music notation is still used extensively. The limitations of Western music notation have been criticized, especially in the 20th century, and alternative notations have been developed. In spite of this, the so called “standard” or “common practice” music notation remains an important form of storing and publishing music. As one consequence, specialized computer applications have been developed to create, process and print music notation.

The Western music notation system has continuously evolved throughout its existence. One reason for the evolution has been the need of composers to represent new musical structures or expressions. Another reason has been the need of music publishers to engrave and print musical scores in an economical way. Although a similar economy and level of standardization as in printing texts has not been achieved, enough commonly accepted principles and systematic rules exist to make it possible to develop computer-assisted tools for many notation-related tasks.

Music notation is a graphical language, which is comprised of different types of symbols and from the placement of these symbols in relation to each other. Compared to other written languages, Western music notation is exceptionally complex. However, it is efficient enough to allow a skilled musician to perform a musical work even at first sight. A computer music notation program needs an explicit description of these rules in order to interpret or process a musical score efficiently. One central aspect of the rules is the structure of a musical score. This does not mean the musical structure of any particular work but rather the structure of music notation as a system of symbols for representing a musical work.

The purpose of this study is to analyze the Western music notation system as a set of objects that act as computational representatives for notation symbols. The primary analytical method is the object-oriented analysis used in computer science. To improve coherency in the resulting analysis, I have extended standard object-oriented methodology with linear logic, a method also originating

from computer science. The result of the study is a *structural object model* that describes the roles and relationships of objects found in music notation. This model can be used as a basis for computer software design.

The intention of the study is to represent a theoretical description of music notation that can be used in the design process of a music notation computer program. Previous studies on computer-based music notation have shown that the processing of music notation involves different types of information. Existing musical data representations serve as current examples of how the various types of information can be encoded and of the kinds of difficulties involved. This study is based on the assumption that a consistent object model can be formed by including only those objects that have an explicit visual appearance; information belonging to levels or domains other than the visual one can be included as properties of visual objects.

1.1 Computer-based music notation

Music notation presents a challenge for software developers for many reasons. Firstly, music notation programs typically have to process several different types of information. (At least three basic types of information have been identified by previous research: graphical, logical and performance. Sometimes, a fourth type, the analytic, is referred to as well.) Secondly, the rule set of music notation is extremely complex compared to, for instance, word processing or other commonly-known computer applications. Third, the market for commercial music notation application is considerably smaller than that of, say, word processing programs. These facts mean that the development process of a notation program is typically difficult and slow, while the potential outcome of a commercial endeavor may be small.

The computational and representational difficulties concerning music notation have also been subject to scientific research in both musicology and computer science. There has been active scientific research on many areas concerning music notation. These include user interfaces, forms and means of data input, different forms of displaying music notation (e.g. music printing, online publication, and dynamic, interactive-graphical presentation systems), automatic spacing algorithms, algorithms for generating a musical performance from a notated score, optical recognition of printed and handwritten music notation, etc. One central area of research is the computer representation of music notation, which also lies within the scope of this study.

Computer representations have been developed for various purposes related to music notation. These include data input, data output, the internal memory structures of notation programs, file formats for data storage, and file formats for

data interchange between notation programs. This study does not present a concrete and detailed representation for any of these particular uses. Instead, an abstract, general-purpose analysis model of the common Western music notation system is presented. This model, comprised of hierarchically organized objects, can be used as a basis for designing concrete representations, for example, for the uses mentioned above.

Representational issues of music notation are related to computer representation of musical data in general. Many notation programs also process data that is not strictly notation-specific. Such data might, for example, be an outcome of an interpretation process of a notated musical score. For this reason, a computer representation of music notation must also take into account other forms of musical data.

1.2 Research methods

Computer science offers systematic methods for various tasks and stages in software engineering. These tasks include the selection and use of programming languages and the preparation of specifications for programming tasks. In addition, formal methods have been presented for analyzing a system, or for a problem to be implemented in or solved by computer software.

Object-oriented analysis is a part of object-oriented software engineering methodology. Object-oriented software engineering uses objects as a basic unit of software construction. An “object” is a combination of data and a set of operations for manipulating the data. The use of objects provides a formal method of decomposing a large and complex system into smaller parts that can be written and tested separately. An object can hide its internal data structures from other objects so that the data is protected from undesired use. Thus, the object’s data structures may also be redesigned and changed without requiring changes to other objects.

Object-oriented analysis is a method whereby a system to be implemented in software is decomposed into objects, and the basic relationships of these objects are defined. The result of the analysis is presented as graphical diagrams and textual descriptions.

Several formal object-oriented methods have been presented. The basic principles of these methods are more or less similar. The main differences concern terminology and graphical notation conventions. In the late 1990’s, the Unified Modeling Language (abbr. UML) became the dominant formal object-oriented method. Thus UML notation was chosen for use in this study. However, UML is primary a modeling language, not a formal method for performing an analysis. In this study, formal analysis methodology is approached by the examination of

existing methods for object-oriented engineering. To systematize further the analysis process, linear logic was used as a complementary method.

Linear logic is an alternative logic to classical logic. Linear logic is based on the principle of limited resources as opposed to the principle of unlimited resources in classical logic. For software engineering linear logic offers advantages for simplified management of memory and other computing resources. Although software systems based on linear logic have been criticized as computationally inefficient, the theoretical principles remain useful as a means of modeling real-life situations. When applied to object-oriented analysis, linear logic provides a systematic method for evaluating various analytical decisions. Furthermore, linear logic provides basic guidelines by which to make structural decisions during the analysis process.

1.3 Application of methodology and general goals

The basic goal of this study is to present a model of the Western music notation system, such that this model, in turn, can be used as a basis for computer software design. The model should be general enough to be applicable to many types of computer applications requiring music notation. At the same time, the model should be independent enough not to require a particular area of application. Also, it should be independent of the computer hardware or software environments and of the implementation programming language.

The original impetus for the present study arose from a technical interest in applying object-oriented techniques to the design of music notation software. Previous personal experience with music representations and on music composition software design had shown that modularization can lead to an effective and efficient musical computing environment while requiring a relatively small amount of programming code (see, e.g., Kervinen and Lassfolk 1993). Then came further experience in designing and implementing an audio signal processing system (Lassfolk 1995 and Lassfolk 1999). This experience showed that an object-oriented approach provided the means of building an easily-maintainable and expandable framework, one that could also be accomplished with a relatively modest amount of programming effort. This leads to experiments with applying similar techniques to the design of music notation software. However, music notation proved to be a much more difficult problem than the one faced in signal processing or the manipulation of musical events. Experiments with different program prototypes showed that a theoretical study, separated from the practical problems of programming, was needed in order to form an objective view of the problem domain.

The present study focuses more on data representational aspects of computer-based music notation, and less on algorithms involved with musical data processing. More specifically, my focus is on high-level structural abstractions rather than on low-level data structures and optimization of storage space.

My study concentrates on defining the different types of objects that can be found in common music notation. This can be seen as the first necessary step in a process of object-oriented software development. What this study does *not* address is another important step in software development: the definition of rules for handling the objects. Among these rules are conventions that affect the placement of notes such a way as to make a score readable and visually pleasing. These rules are described in great detail in several books on music theory and engraving. Hence, duplicating them here would have been both redundant and unnecessary.

One goal of the study was to achieve a simple and coherent model of a system that in itself appears to be extremely complex. To help achieve this goal, I applied systematic methodology beyond conventional object-oriented techniques. In the light of the complexity of music notation as a communication system, the amount of detail that could be handled had to be kept small. Limiting the amount of detail also helps to keep the model and its presentation simple.

1.4 Results

The results of the study are centered around a structural analysis model of music notation. The model is presented as a set of UML class diagrams with accompanying text descriptions. The model presents a structure of object classes and their relationships. One aspect of the analysis involves the classification of musical symbols. Another aspect is the definition of various relationships between the classes. One central set of relationships is the decomposition of a musical score; this starts from the abstraction of an entire score and proceeds down to its smallest symbolic constituents: note heads, lines, dots, letters, numbers, etc.

This is not the first project to apply object-oriented techniques to computer-based music notation. In fact, several ongoing notation programs have been written with object-oriented programming languages. Nevertheless, few systematic scientific studies have been published that deal with what music notation is in terms of objects (i.e., classification of music notation symbols). The key contribution of this study is not the new-object model itself. Rather, object-oriented modeling is used here as a means for gaining a new perspective on music notation and on its computer representation. *Above all else*, the object modeling applied in this study helps to isolate and organize the different types of musical

information mentioned above, and to examine their roles and mutual relationships.

What separates this study from previous ones on the same area is that it presents a general-purpose model of music notation rather than one aimed at some particular computer program or application. Here, object-oriented analysis, further refined with the principles of linear logic, is used as a systematic research method. Moreover, the object-model of music notation is presented as a result of using this methodology, rather than as a mere case study of some software engineering method. The main aim of this study is to present a description of music notation that can be used as a basis for software design. In addition to this practical aim, the present work can be approached as a theoretical study that presents a systematic categorization of the symbols used in Western music notation.

The principal value of this study does not lie in the application of object-oriented methods to music notation in general. Rather, its main value concerns the way that music notation is approached and how object-oriented methods are applied and developed further. Here, music notation is approached primarily as a graphical system, which contrasts with some recently developed representations of music notation. Object-oriented methodology is extended with linear logic, which provides a strict set of rules to help in forming an object structure for representing music notation.

Like representations in general, my analytic model is a result of interpretation. It does not represent absolute and universal truth. Rather, the analysis model is an instrument for retrieving information from its target. For example, it can reveal potential problems concerning computer representation or music notation in general. With the analysis model proposed here, these problems can be taken into account and possibly solved in a software design process.

1.5 Overview of contents

This text includes descriptions and discussions of the following questions: What is common Western music notation? What is required of a high-quality musical representation? What types of musical data representations exist? Do they relate to music notation, and if so, then how? What is object-oriented software engineering? How can it be applied to the development of music software? Finally, how can music notation be represented in a form that is realizable as a computer program, while remaining true to the traditional semantics of music notation?

The text is divided into 10 chapters. Following this introduction, Chapter 2 describes various issues in computer-based music notation and defines basic terminology. Ways of examining and categorizing music notation programs are

described. Also discussed are general principles and problems involved with computer-data representation of music notation.

Chapter 3 describes the terminology and criteria of computer-based musical data representation, with a focus on music notation. A selection of existing computer-based musical data representations and their design criteria are also described. Among the representations are file formats, music-related programming languages, and data structures of notation programs. Their applicability for representing various forms of musical data, particularly music notation, is discussed. Short examples are given to demonstrate their syntax and semantics. These particular representations were chosen because they are typical representatives of their genre and are therefore documented in sufficient detail. Less emphasis was put on their availability, popularity among users, or commercial success.

Chapter 4 describes central aspects of object-oriented, software engineering methodology. Basic object-terminology as well as the historical and philosophical background of that methodology are presented. A discussion of object-oriented software engineering methodology is included, and basic principles of the Unified Modeling Language (UML) are described.

Chapter 5 discusses linear logic and its application as a complementary method for object-oriented analysis. Chapter 6 discusses the application of object methodology to the analysis of music notation. General principles and objectives for analysis of music notation are also described.

Chapter 7 presents an analysis model of the common music notation system. The model is illustrated with a set of UML class diagrams and commented on in the text. Chapter 8 shows examples of the model as UML object diagrams.

Chapter 9 contains a discussion of the advantages and disadvantages of the analytic model as well as the application and modification of the model for the purposes of software design. The model's application for software design receives discussion, leading to Chapter 10, which presents a summary and concluding remarks of the study, as well as possible directions in future research.

Chapter 2
Computer-based music notation

In this chapter, basic terminology of music notation is defined, including the term *music notation*, or more specifically, *common Western music notation*. Also, various aspects and problems of computer-based music notation are discussed.

At least since the 1980's, computer-based music notation has been an object of ongoing scientific research. Moreover, the 1980's saw a dramatic increase in the availability of both commercial and academic notation programs. In 1987 Walter Hewlett and Eleanor Selfridge-Field (1987: 35-73) published notation examples made with 17 different computer-based notation systems, including both proprietary systems and commercial software). In 1989 the number of systems demonstrated had grown to 23 (Hewlett & Selfridge-Field 1989: 57-105). This increasing trend has continued in those authors' later publications.

Research areas involving music notation have included data structures, spacing algorithms, means of data entry, and user interfaces, among other subjects. Below, overviews of some of these studies are presented. Specific computational and representational problems concerning computer-based music notation receive special discussion.

2.1 Common Music Notation

Western music notation is a graphic system used to encode music so that it can be interpreted and reconstructed by people. The entry on the topic in the *Grove Dictionary of Music and Musicians* (1980) says music notation can be described as either a description of sound events or as "a set of instructions for performers". According to Nelson Goodman and Kari Kurkela, music notation can be examined both syntactically and semantically, as in the case of a natural language – although music notation differs in many ways from natural languages (Goodman 1985; Kurkela 1986).

Besides *Western music notation*, the system is also called *conventional music notation*, *common (Western) music notation* (sometimes abbreviated as *CMN*), *common-practice music notation*, and *staff notation*. No official or formal stan-

dard designation for this form of music notation exists. However, the term *standard music notation* is used to denote a loosely restricted set of Western music notation symbols and conventions. In this study, the terms *common music notation* or just *music notation* will mainly be used.

Roads defines common music notation as follows: "Common music notation (CMN) is the standard music notation system originating in Europe in the early seventeenth century." (Roads 1996: 708.) In the context of this study, this definition can be further refined as *the set of music notation conventions established by music publishers and musical education institutions in the 20th century for the representation of 18th and early 19th century Western art music.* Although common music notation is even used to represent contemporary art or popular music, it does not include special symbols or conventions developed for contemporary music.

Donald Byrd's SMUT notation program was aimed at "handling virtually all Western music written from about 1600 to 1945" (Byrd 1984: 6-7). According to Byrd:

> *CMN ("conventional" music notation) includes any arrangement of the symbols in general use by composers in the European art music tradition from about 1700 to 1935, used with the meanings that were standard: (1) if the notation was made between 1700 and 1935, at the time it was made; (2) if the notation was made before 1700, with the meanings of 1700; or (3) if the notation was made after 1935, with the meanings of 1935.*

Byrd admits, however, that "the endpoints 1700 and 1935 are somewhat arbitrary." (Byrd 1984: 13.) Byrd illustrated his description of conventional music notation by dozens of examples extracted from scores of the standard repertoire of Western art music. In particular, Byrd focused on special cases that are potential sources of problems for computer-assisted music notation.

As a language, Western music notation is highly complex, more so than any written natural language or even Western mathematical notation (Roads 1996: 708). Donald Byrd acknowledged this complexity by comparing music notation with both mathematical notation and the written Chinese language (Byrd 1994). The complexity of music notation is caused by, not only the number of different symbols, but the complex rules that govern the coexistence the symbols.

As an example of the number of different kinds of symbols in music notation, *The Essential Dictionary of Music Notation* by Tom Gerou and Linda Lusk (1996) lists 79 different topics. Several individual notation symbols receive their own topics, while some groups of related symbols are grouped under a common topic. Topics have also been given to concepts or practices such as "spacing". Gerou and Lusk's text is still not a comprehensive manual or guidebook of music notation, but is rather intended as a concise encyclopedia. Kurt Stone

(1980), in turn, describes at more length the use of common music notation augmented with specialized techniques of contemporary music notation. Gardner Read's book (1982) is yet another example of a description of music notation practices.

Guitar and lute tablatures form another kind of music notation that also is sometimes regarded as part of common music notation. Both specialized contemporary music notation techniques and tablatures are excluded from this study.

2.2 Graphical aspects of music notation

Music notation is not only complex in terms of the amount of symbols or rules of encoding information. As a graphical system it also has rules and conventions that govern the visual layout. The main purpose of these conventions is to improve the readability of a musical score so that it can be interpreted as fast and correctly as possible. This is particularly important in a concert performance of a musical work.

In addition to rules derived from music theory, visual layout is controlled by conventional esthetic considerations. Entire text books are devoted to teaching the conventions of musical manuscript writing and engraving (e.g., Ross 1970; Heussenstamm 1987). Many music publishers have developed their own layout conventions, sometimes called "house styles". Many individual music engravers have also developed their own personal visual styles.

George Heussenstamm (1987) describes layout conventions and also gives detailed instructions on how to draw notation symbols by hand. Ted Ross (1970) describes in great detail how to engrave publication-quality musical scores. He discusses, not only the commonly accepted and established rules on how to stack notes and adjust the direction of stems, but also, how much space each symbol should be given on a staff, how to place staves on a page, and the like. Despite providing these detailed instructions, Ross acknowledges that there exists no single standard for music engraving. Engraving and layout practices vary between individual music publishers and engravers. Engravers even disagree on the role and placement of common symbols such as barlines (Ross 1970: 151).

2.3 Dynamic and evolutionary aspects

Western music notation, from its earliest forms up to those of the twenty-first century, has appeared in several different representational forms and their minor

variants. The time period of common music notation, as defined by Byrd (see above), represents only a fragment of this long evolutionary process. Even common music notation itself has not remained a static system. For example, the conventions of writing manuscripts of eighteenth-century music differ, in many details, in the twenty-first century from what they were in the nineteenth century. The music publishing industry has obviously played an important role in establishing many conventions used in modern music notation. It can be expected that the use of computers will also have an evolutionary effect on the conventions of music notation. Manual tools, such as stamps or specially designed rulers used by music engravers, also affect the graphical layout of scores and the shape and maximal degree of variance among notation symbols. The engraver is tempted to use tools that give the desired result with the least effort. A similar phenomenon can be seen in the design and implementation of music notation programs. Programmers are tempted to solve problems with as little programming or designing effort as possible in order to save time in the software development process.

The increasing use of computers in music publication and the like has affected, and will continue to affect, music notation itself. Techniques that are easy to accomplish with a computer become more common, while difficult techniques are avoided. This phenomenon may shape and restrict the expressiveness of music notation and thus have an evolutionary effect. Composers who write music using a notation program that has a limited set of capabilities easily fall into restricted notational expression. In addition, a difficult learning phase of a notation program or of some notational feature can form an obstacle to its use, which is yet another potential cause of evolution.

2.4 Computer applications

Computer programs that can process music notation can be categorized in several ways. At least the following criteria can be listed:

1. Intended use or user base
2. Functionality or feature set
3. Type of user interface
4. Type of data representation

Intended use or *user base* refers to the type of use or users the program is designed for and marketed to. A user base may be defined by the kind of musical background (e.g., popular music or classical music), educational background, level of expertise (professional, amateur, student, etc.), or even the special needs of a particular instrument (e.g., the guitar). Intended user base and intended use

are partly correlating criteria. According to Glendon Diener, music notation programs have three basic types of use (1990: 6-7):

1. Compositional
2. Archival
3. Analytic

Diener based his description on Hugo Cole's (1974) earlier categorization, which included four types of use. The only significant difference between Cole's and Diener's categories are that Cole specified distinct categories for both composition and arranging, whereas Diener described both of these uses as "compositional" (Diener 1990: 6-7).

The uses of notation programs can be also divided into (a) publication of music notation and (b) other types of music production. In the publication of printed music, graphical layout capabilities and quality of graphical output are of great importance. In other types of music production, the notation may serve only as a user interface or as an intermediate form of communication between music producers and performers. In such cases, aspects other than the quality of graphical output may be more important to users. Nevertheless, some acceptable degree of graphical quality is required of all programs that handle music notation – at the very least, the notation produced should be readable and understandable.

The user interface of a notation program often also reflects the kind of users the program is intended for. A shallower learning curve, but with fewer capabilities or less control of detail, is typical of programs intended for non-experienced or casual users, whereas programs intended for professional users may be hard to learn but typically offer a relatively greater deal of control, with several alternative means of data entry. The capabilities of several professional music notation programs have been examined and compared by Alan Belkin (1994), and will be discussed further below.

When categorized on the basis of functionality, programs may be divided into general-purpose and special-purpose programs. A high-quality, general-purpose program is capable of handling several types of notational uses needed by composers, arrangers, music students, and others. Some general-purpose programs are even used by music engravers and publishing companies. Special-purpose programs, on the other hand, are designed or optimized for some specific task, such as music publishing, music education, guitar or lute tablatures, medieval notation, or music analysis.

On the basis of user interface, programs may be divided into interactive and non-interactive ones. Interactive and non-interactive methods of data input and editing are both described in section 2.5.

The data representation of a notation program shows how data is organized within, imported to, or exported from a notation program. A representation also reflects the capabilities of the program that uses it. On the basis of data representation, programs may be divided, for example, into language-based, data-structure-based, and object-oriented. Since non-interactive programs require an external program for data entry and processing, they must support a well-documented data representation. The representation is often a text-based, specialized programming or formatting language that can be written by a general-purpose text editor. Alternatively, the representation may be a binary file format. Musical data representations, including computer representations of music notation, are described in Chapter 3.

2.5 Capabilities of music notation programs

The capabilities of music notation programs can be divided into three basic categories: data entry, data output, and editing. Data entry refers to ways of creating notation. Data output refers to ways of displaying scores or parts of scores, and to ways of exporting information in various forms. Editing refers to ways of manipulating notation that has been previously created by some form of data entry.

Alan Belkin describes the problems involved in music notation software development. Belkin observes that some of the problems are caused by music notation itself. Conventional word processors are very similar to each other in features and deal with a relatively simple flow of data. In contrast, the processing and organization of music notation data is much more difficult, and music notation programs must also include a large set of features to accommodate different kinds of users (Belkin 1994: 53-54).

To be called a “notation program”, a program requires at least one kind of graphical output and at least one form of data input. Other than this, any combination of the capabilities mentioned above is possible. For example, editing capabilities are not necessarily needed if the program supports an input language that can be created and edited with an external program.

As already mentioned above, music notation programs support many different ways of inputting data. A particular program may support from one to several input methods. At least the following input method categories can be listed:

1. Input language
2. Point and click entry
3. Computer keyboard entry
4. Piano-style keyboard entry
5. File import from other programs
6. Data generation from event file formats
7. Automatic composition/arrangement/orchestration

An input language is typically written manually with a conventional text editor or some other external program. Point and click entry refers to data entry by an interactive, typically graphical user interface that is operated by a mouse, touch pad, or other pointing device. Computer keyboard entry refers to an interactive user interface, in which notes are created by the typing of commands on a computer keyboard. Many interactive notation programs allow note entry via a piano-style keyboard. In that case, music can be entered, for example, by one's playing the notes in real time (against a metronome click played by the program) or in "step time", by the choice of pitches from the MIDI keyboard and entry of durations with a computer keyboard.

File import from other programs may be provided, for example, in the form of a dedicated, notation-exchange file format or by interpreting the native file format of another notation program. The generation of notation imported from event files, from MIDI files in particular, is supported by several notation programs. MIDI files are discussed in the next chapter.

A notation program may also provide automatic composition, arrangement, or orchestration capabilities. Also, a macro-facility may be used for automating routine note-entry tasks.

Other, less common input methods include handwritten notation, which uses a graphics tablet (see Forsberg *et al.* 1998) and optical recognition of printed or handwritten music notation. Also, information retrieval from acoustical signals by automatic transcription could be applied as a method of inputting data (e.g., see Klapuri 2003; McNab *et al.* 1996). These methods could be implemented as external programs, or as integral parts of notation programs, such as extensions (e.g., "plug-ins") of notation programs.

Data output methods of notation programs include the following:

1. Computer display
2. Printing
3. Synthesized audio playback
4. File export to other notation programs
5. Export to event files (e.g. MIDI Files)

Printing and computer display are the most common forms of data output. A synthesized audio playback capability is typically provided for proof-reading purposes. In a more sophisticated form of synthesized playback, algorithmic phrasing methods can be used to make the playback sound less mechanistic than a rhythmically rigorous conversion from graphical or logical notation data. File export may also be provided to other notation programs as well as to MIDI files or other event-file formats.

The available editing methods of notation programs include the following:

1. Interactive point-and-click editing
2. Keyboard shortcuts or macros
3. Interactive command language
4. Editing of input language (i.e. in batch-based, non-interactive programs)

A point-and-click-based, interactive, user-interface is perhaps the most commonly offered editing method in commercial, general-purpose notation programs. To speed up common tasks and to automate routine tasks, interactive programs often offer keyboard shortcuts or user-definable macro commands (as in the case of data entry). In some notation programs, a command language can be used either instead of or in addition to point-and-click-type operations. In the case of an input language, editing is performed in some external program rather than within the notation program itself.

As in the cases of both input and output methods, a particular notation program does not necessarily provide or need all the editing methods listed above. For example, programs with no interactive data entry or editing do not need operations for such tasks as selecting, moving, copying, or deleting notation symbols.

Alan Belkin compared the capabilities of commercial notation programs available for the Apple Macintosh. Belkin presented an extensive list of features divided into the following main categories (Belkin 1994: 65-67):

1. Note entry
2. Entry of slurs, articulation, dynamics, etc.
3. Selection in regional editing operations
4. Editing
5. Special, customized notations
6. Lyrics
7. Midi playback
8. Entry layout
9. Page layout
10. Part extraction
11. File operations
12. Interface and overall ease of use

All the programs compared by Belkin were based on a graphical, interactive user interface typical of Macintosh software in general. Belkin noticed a tendency towards unification of features between the programs, although their origins differed considerably. Belkin also noted that a previous contradiction between ease of use and amount of flexibility was becoming less obvious as the programs became more mature (Belkin 1994: 54). He cited some "standard" requirements, such as viewing of user-selected parts of a score, cut and paste editing, and MIDI playback. However, he concluded that "currently, no available notation software meets all of these requirements" (ibid.: 53-55). Belkin restricted his review to the Apple Macintosh platform. Thus, music notation languages designed for text-based user interfaces were not discussed. It can be assumed that there has been considerable progress in the capabilities of notation programs since the time of Belkin's survey. Some of the programs available in 2004 might well meet Belkin's requirements. There are, however, also new requirements, such as music publishing via Internet, that would have to be considered, if a similar survey were to be conducted in the year 2004 or later.

Belkin saw a need for the transfer of notation data between programs. He proposed a list of requirements for a "standard notation file format" (SNFF). Partly from Belkin's initiative, a group of developers began development of the Notation Interchange File Format (NIFF).

2.6 Types of information

The Notation Interchange File Format specification (NIFF 2002) states that music notation contains three types of information *components*: *graphical*, *logical*, and *performance*. These components are loosely connected. This principle was first presented by Ornstein and Maxwell (1983; see also, Maxwell & Orn-

stein 1984). A similar principle forms the architectural basis of the Standard Music Description Language (SMDL). The latter groups different types of musical information into *domains*, one of which is called *visual* and another *gestural*. The SMDL equivalent for the logical domain is called *Cantus*. SMDL also specifies a fourth domain called *analytic* (Sloan 1997).

It is difficult to separate and define these components in a precise manner. Eleanor Selfridge-Field (1997b) describes the relationships between logical, "physical" and "practical" information by using an analogy to geographical maps. As she points out, a map may represent physical, practical, and logical information, or some combination of these. For example, one kind of map can show borders of states, and a road map represent physical presence, but both maps also constitute logical information.

In a computer program, the graphical information represents instructions for displaying visual symbols on a computer screen or on printed paper. Logical information includes invisible connections and relationships between individual graphical symbols. This includes terms such as "voices" in a polyphonic structure or the durational "value" of a note. Performance information represents musical interpretation of a score. This typically includes the precise timing of notated events, phrasing, intonation, detailed use of vibrato, and so on.

The difference between performance, logical, and graphical information can be demonstrated with an example. "Quarter note middle C" can be regarded as logical expression describing musical information. The expression "one second long, 261.62 Hertz tone" can be regarded as a performance-oriented or physical expression. A corresponding graphical expression might in turn be as shown in Figure 2-1.

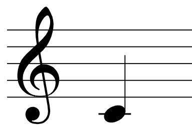


Figure 2-1: A simple expression using common music notation

The notation example of Figure 2-1 could also be expressed verbally, as "a staff containing a treble clef and a quarter note with an upward stem on the first lower ledger line" or "five horizontal lines, one vertical line, a spiral-like shape, and a filled ellipse with a short horizontal line crossing it". Both expressions can be

regarded as graphically-oriented, but the first uses musical vocabulary, whereas the second uses general graphical vocabulary. This shows that even graphical types of information may be interpreted and expressed on different semantic levels.

The expression of pitch is often an indicator of the orientation of a representation. If a pitch is coded by the note's vertical position on a staff, it is an indication that the representation is (likely to be) graphically-oriented. By contrast, if a note name and octave range are used, then the representation can be considered as more logically-oriented. A combination of both ways of specifying pitch indicates that the representation is designed to express both types of information explicitly.

A purely logically-oriented representation primarily represents logical information explicitly, and graphical information implicitly (either fully or in part). If a notation program uses a logically-oriented representation, it must typically calculate at least some of the placements of notation symbols automatically. A logically-oriented representation may, however, contain either optional or mandatory graphical parameters to aid in the calculation process. A graphically-oriented representation, in turn, encodes graphical information explicitly. A performance-oriented representation may also allow optional expression of logical or graphical information, although its typical purpose is to express performance information in an explicit way.

In common music notation, graphical information is explicit. Logical information is implicit, and must be derived through analysis or interpretation of the graphical symbols. Performance information, for its part is mainly expressed implicitly and can be generated by interpreting the graphical information.

As noted above, graphical information may be further subdivided into several levels of abstraction. The "lowest" of these may be called the "physical" level. In a printed score, this level would consist of ink and paper. In a computer software implementation, it might consist of primitive computer graphics (such as points, lines, curves, etc.) or of individual pixels on a computer display – these may be regarded as virtual equivalents of paper and ink. On a higher level, graphical information can be represented as complex symbols, such as notes, rests, key signatures, or staves. Some of these symbols can consist of physically separable parts. On an even higher level of abstraction, a whole musical work may be regarded as a single graphical symbol.

Performance information, too, may be represented on many abstract levels, including a stream of events or a sampled audio signal. When a musical score is performed, each note is given an exact beginning and duration in time as well as nuances such as vibrato, tone color, and more. In performance, graphical details

(such as stem direction) and logical information (such as tempo) either disappear or are translated into a combination of several notes or parameters.

Similarly, a detailed analysis of logical information in music notation can be expected to reveal different levels of abstraction. How SMDL distinguishes between logical and analytic domains is an indication of this. Also musical structure can be regarded as either graphical, logical or analytic information. This raises an additional issue to be considered in evaluating the relationships and roles of the different information types. Roger Dannenberg discusses hierarchy and musical structure in music representation. According to him, it is impossible to represent musical structure sufficiently in terms of a single hierarchy. For example, beams and slurs form two different structural constructs, which often intersect (Dannenberg 1993: 20-21). Therefore, the encoding of musical structures as parts of relationships could result in a complex set of interweaving hierarchies.

2.7 Rules and ambiguities

The basic rules of music notation have long been described in music theory books, some of which were mentioned above. Among these, George Heussenstamm's book (1987) describes basic music notation techniques, and Gardner Read (1982) presents an even more detailed guide to music notation.

In even more detail, Ted Ross describes the rules that govern the graphical placement of notation symbols. For example, the correct beaming of two consecutive eight notes is described with illustrations of more than 270 discrete interval combinations. In these illustrations, the correct stem direction, stem length, and beam angle is shown for each interval (Ross 1970: 104-110). A direct conversion into algorithmic form of Ross's examples of "correct" and "incorrect" notation would create an extensive list of conditions. Although Ross's book is written for human music engravers and not for software designers, it shows how much detailed consideration must be paid to even the simplest engraving tasks.

Donald Byrd (1994) takes examples from works of J. S. Bach, Brahms, Debussy and Ravel to demonstrate situations which are likely to cause difficulties for computer software. In his Ph.D. thesis, Byrd presented a large amount of examples extracted from standard Western art music repertoire, such examples showing even more ambiguities and difficulties (Byrd 1984). Both Byrd and Dannenberg reached the conclusion that several tasks require manual work, such as transcription of musical performances and layout adjustments necessary when instrument parts are extracted from an orchestral score (Dannenberg 1993).

Automatic spacing has been studied as one of the most difficult areas in notation software design. Blostein and Haken (1991) discussed the difficulties of designing spacing algorithms. Earlier studies, made by Gourlay (1987) and Rouch (1988), on the spacing of monophonic music notation provided solutions for spacing single staves or simple homophonic material. Blostein and Haken (1991), however, went on to address polyphonic and multi-staff notation.

Blostein and Haken described a complex, two-pass spacing algorithm used in the LIME notation program. In the first pass, two types of spacing, “textual” and “durational”, are computed. In textual spacing the width for each chord, individual note, clefs, key signature, lyric syllable, and so on are computed in parallel on all staves in the system. Textual spacing does not take into account the duration of notes. For this, the durational spacing algorithm, in turn, uses a nonlinear function of note length to compute spacing. In the second pass, textual and durational spacings are compared and combined while the desired width of the music is taken into account (Blostein & Haken 1991: 95).

SCORE uses a semi-automatic approach to spacing. In the data input stage, symbols are placed horizontally according to the durations of notes and rests. When the input of a whole system has been completed, the user must type a command that causes SCORE to make fine adjustments to spacing. SCORE’s spacing scheme is divided into two processes: “lineup” and “justify”. The user may either execute both processes at the same time, using a single command (named “LJ” for “lineup and justify”), or perform each operation separately (i.e., either “L” or “J”). The lineup and justify processes have functions approximately similar to Blostein’s and Haken’s textual and durational passes, respectively. SCORE’s lineup process aligns every note and rest belonging to the same beat in vertical groups. The justify process spaces the symbol groups according to their durational values (SCORE 1991b: 243-244).

SCORE provides an additional command for justifying a previously spaced system to accommodate a lyric line. Its purpose is to handle situations in which long syllables do not fit between their respective notes. If the result is still not pleasing, the user may move symbols manually. A detailed description of SCORE’s spacing algorithms has not yet been published.

Kai Renz presented an improved version of Gourlay’s spacing algorithm. He also stated that most published spacing algorithms were minor modifications of Gourlay’s scheme. Renz demonstrated the weaknesses of Gourlay’s algorithm, especially in the spacing of tuplets in multi-stave systems. He also demonstrated that the spacing algorithms in the Finale and Sibelius notation programs suffer from deficiencies similar to those in Gourlay’s algorithm (Renz 2002.)

The descriptions of the above spacing algorithms concern the placement of symbols within a staff or a system. An even higher-level problem exists. An

engraver might, for example, deliberately misalign barlines on nearby systems to enhance readability of the score or an instrument part. This situation has also been noted by Gerou and Lusk (1996: 52). Also, vertical spacing between staves and systems may have to be adjusted to avoid collisions between symbols on neighboring staves. Furthermore, facile placement of page turns may be difficult even for a human engraver (see Ross 1970: A3-A5).

2.8 Music notation and computer graphics

Music notation programs use computer graphics as their primary means of data output. Notation may be shown on a computer display, or it may be printed on paper. Many programs use an interactive graphics system as their user interface. The graphics-processing capabilities needed by a notation program may be divided into general-purpose graphics algorithms and notation-specific capabilities.

Kimball Stickney divided the implementation techniques of notation symbols into two categories: “iconic” and “algorithmic” (1987: 129-131). There, iconic symbols can be seen as computer equivalents of metal stamps used in manual music engraving. In a score, extensively used symbols that do not have to be varied in shape or size can be conveniently implemented as “iconic” symbols. For example, these can include note heads, clefs, and symbols for rests and accidentals. Many computer graphics systems offer fonts as a means to implement iconic symbols. “Algorithmic” symbols, by contrast, are calculated individually for each symbol and often vary in shape. These symbols include slurs, beams, braces, and stems. Algorithmic objects offer more graphical flexibility than do iconic symbols, but they are computationally less efficient.

Many general-purpose graphics systems, such as Adobe PostScript, Apple MacOS, and Microsoft Windows, support both iconic and algorithmic graphics. For example, the Adobe Sonata PostScript font (Grosvenor *et al.* 1992: 354) is specifically designed for music notation. It includes common notation symbols such as note heads, rests, clefs, stems, flags, and accidentals.

General-purpose graphics systems provide means for drawing lines, curves, circles, ovals, rectangles, polygons, and more. For example, stems, staff lines or even beams, may be flexibly drawn with general-purpose graphics routines. Some notation-specific algorithmic symbols are more difficult to produce. These include braces and slurs. Few graphics systems offer ready-made primitives for drawing these symbols. PostScript (Adobe 1986), for example, includes a cubic Bezier curve primitive, but a single Bezier curve does not suffice to draw a shape with varying width, such as a music notation slur or brace.

Special-purpose processing is also required when fonts and algorithmic shapes have to be joint or jointly adjusted. For example, to join perfectly a note head with a stem may cause difficulties for a music notation program. In that case, the exact dimensions of the note head must be known in order to adjust the starting point of the stem. Also, the width of the stem must be taken into account so that the stem starts precisely beneath (i.e., from inside of) the respective note head.

2.9 Music notation terminology

According to Donald Byrd (1984), common music notation represents four basic parameters: pitch, time, loudness, and timbre. Of these, the representation of pitch and time are explicit or fairly explicit; representation of time is mostly implicit; and timbre is both explicit and implicit. Loudness may be expressed explicitly by verbal comments but is mostly implicit. Timbre is expressed explicitly by naming instruments or ways of playing (e.g., "sul ponticello"), but mostly the expression of timbre is implicit (Byrd 1984: 10-11).

A *note* is a central term in most music representations. The semantics of note, however, has different meanings among various representations. In performance-oriented representations, note is typically equivalent to a sound event. Performance-oriented representations include sound synthesis programs and MIDI, as described in the next chapter. In music notation, note is typically equal to a graphical symbol and has various parts, such as a note head and a stem. Alternatively, a note may refer to a logical unit, which may have several alternative appearances in sonic or visual domain. In a visual domain, e.g. in common music notation, a logical note might appear as either a single graphical note symbol or, if its context requires, as two successive note symbols connected by a tie.

Pitch is an ambiguous term. Its meaning and representation differ among musical data representations. In performance-oriented systems, pitch may mean a fundamental frequency of a sound event or a symbolic parameter referring to, for example, a key on a piano keyboard, where each key produces a sound with a predefined pitch. In logically-oriented music representation systems, pitch is often represented by a name that contains a pitch class and an octave. In music notation, pitch is affected by several parameters, including the note-head's vertical position on a staff, a preceding clef symbol, a preceding key signature, preceding accidentals, preceding barlines, and, in some cases, even the kind of instrument.

Duration of a note is also ambiguous. In performance-oriented systems, duration refers to the length in time of a sound event. In music notation, a com-

mon form of expressing duration is in fragments of a "whole" note; e.g., half note, quarter note, eighth note, dotted quarter note, etc. A referential duration for these relative note durations is defined with tempo expressions. Logically-oriented representations often use a similar way of expressing duration.

Time has a partially similar meaning in performance, logical representation, and notation. In performance, time can be measured in real time (e.g., minutes and seconds). Logical time is measured in beats and measures. Some performance representations also use a beat-based expression of time instead of, or as an alternative to, real time. In notation, time flows roughly from page to page, from the top-most to the bottom-most system on a page, and from left to right on each system or staff. Barlines are used as a means of synchronization. Notes that are vertically aligned are usually also played simultaneously. There are, however, several exceptions to this general principle. In some cases, notes cannot be aligned perfectly vertically, but are still intended to be played simultaneously. On the other hand, an arpeggio sign indicates that vertically aligned notes are not to be played exactly simultaneously. Also, instrumental limitations may require breaking a chord such that all the notes will not be played at exactly the same time.

In notation, *dynamics* and *loudness* are expressed by written comments and by graphical symbols (e.g. "wedges" or "hairpins" indicating increase or decrease of loudness). In performance representations they can be expressed as signal amplitude values or as instrument-specific parameters, such as key stroke velocity values for keyboard instruments. Logical representations may use both implicit and explicit expression of dynamics and loudness.

The staff is an organizational unit that can be regarded as both graphical and logical. A staff is usually explicitly visible as a one or more (typically five), equally spaced, solid horizontal lines. Additionally, the staff also forms a coordinate system within which notation symbols are placed. In music engraving, staff space and staff step are commonly used units of measuring distance (both vertical and horizontal). Staff space is the distance between two consecutive staff lines. A staff step is one half of a staff space. *A system* is a way to connect staves to indicate that they are performed in parallel. A system may be regarded as a unit for both logical and graphical organization. A group of staves connected with a systemic barline is unambiguously recognized as a system (see Gerou and Lusk 1996: 308).

Voice, *part*, and *measure* (a.k.a. *bar*) are used for the logical or analytical organization of notes or other symbols. In notation, voices and parts correlate partly with staves. For example, a part (e.g., that of an instrument) is often written on a dedicated staff or group of staves. However, also more than one part may be written on one staff. A voice is often used to refer to a monophonic mel-

ody line within a polyphonic structure. A voice may also refer to either the part of a solo performer (a human voice or an instrument) or a group of performers singing or playing in unison. In music notation, voice is displayed implicitly by stem direction and, sometimes, by slight horizontal misalignment of notes instead of their being grouped into a chord. In many computer representations, a voice is an organizational feature within a part; i.e., a voice could also be called a “subpart”. Visually, there is no dedicated explicit symbol that represents a voice. Instead, a voice is implied by relative placement or by stem direction of notes. Measure or bar is a horizontal section of a staff or system enclosed by barlines. Visually, a measure can be regarded as an implicit construct of a staff or system and surrounding barline symbols.

2.10 Some conclusions

In this chapter, various difficulties involved with computer-based music notation as well as representational and terminological aspects of music notation were discussed. The issues in notation software design concern identification of the different types of information described above, the definition of their respective roles, and the finding of suitable representations for them. The representation of program data must be capable of supporting not only the various types of information, but also the complex rules of both correct notation and pleasing visual layout.

Of the three basic types of information, only graphical information is mandatory for producing music notation. Without the ability to process graphical information, on some level, a computer program cannot be called a music notation program. Therefore, issues concerning graphical data-processing have to be addressed in the design process of music notation software. In this respect, logical and performance information are only of secondary importance. For a computer program, the ability to store and process logical or performance information is a benefit, and may in many kinds of applications even be a necessity. For example, a purely graphics-based notation program may be unable to provide adequate assistance for users. Moreover, if the notation program is intended to be used by a musician rather than a graphics designer, it should operate on or at least demonstrate itself to the user on an understandable semantic level. A distinction should be made, however, between the user interface of a program and its scheme for internal data representation. In some programs, Leland Smith’s SCORE in particular, the internal data structures are visible to the end user, while in many others, they are hidden under a user-interface layer.

A considerable part of data processing in any music notation program is devoted to graphics processing. Questions concerning general-purpose graphics

are under on-going research (e.g. see Feiner *et.al.* 1992). Questions concerning the application of those techniques are part of the specialized design tasks of music notation software.

As described above, there are several alternative methods of inputting, outputting, and editing data in computer-based music notation. A general-purpose, high-quality computer representation should be applicable to any of those methods and, ideally, be adaptable to new, as yet uninvented methods. To achieve this, the representation should be independent of any particular input, output, or editing method.

From the above discussions, the requirements for an "ideal", all-purpose notation program can be summarized as follows: (1) the program should be applicable to any of the basic uses listed by Diener; (2) it should provide a high degree of graphical freedom (as required by Byrd); (3) it should provide guidance for correct notation (i.e., implementation of the rules presented by Heussenstamm or Ross); and (4) it should provide assistance in automating notation tasks (as suggested by Belkin). These requirements can be difficult to achieve in a single program. Yet, in designing new representations for computer-based notation, one should take these criteria into account, if only to assure that the representation does not prevent any of the requirements being met, if not now, then during future developments of the system.

Chapter 3
Computer-based musical data representation systems

Computer-based musical data processing involved with music production may be divided into three main areas: (1) synthesis and processing of audio signals, (2) manipulation of musical events, and (3) music notation. Specialized programs, with a wide variety of user interfaces, are available for each area. Also, general-purpose or multi-purpose programs exist which can handle more than one of these tasks. Specialized data representations have also been developed for all three of these areas.

An example of an early data representation system for musical sound synthesis is the Music V scoring language (Mathews et al. 1969). Music V provides a specialized programming language for describing a detailed sound synthesis process. Music V and its predecessor, Music IV, have greatly influenced other music synthesis programs as well as programs developed for the other two main areas of application. The Music V program produces a digital audio signal, which is typically stored in a pulse-code modulated (PCM) form.

The introduction of the Musical Instrument Digital Interface (MIDI) in 1983 was an important factor in the development of programs that process musical events. In turn, the development of interactive, graphical user-interfaces for personal computers during the 1980's affected the development of interactive music notation software. Most modern musical notation programs used for commercial and general purposes have adopted the interactive "point-and-click" user interface used in the Apple Macintosh and Microsoft Windows computing environments. User interface principles have also affected and placed constraints on the representation of musical data.

This chapter presents an overview of some musical data representation systems with emphasis on music notation. With some exceptions, the discussion concentrates on text-based representations. Text-based file formats and languages are often used as a means of data input or data interchange between musical programs. In order to be usable, such a format needs to be thoroughly documented. Less information is available for internal data representation of notation programs – especially commercially distributed ones. There are two

exceptions for which usable documentation has been published: SCORE and Lime. Their representations are also described in this chapter. My intention here is not to present a thorough listing or description of even those notation-specific representations that have been published. The focus is instead on basic architectural features of musical representations and on their semantic differences. General questions concerning representation are also discussed.

3.1 Music representation

According to David Huron (1992: 7), "to represent something is to create a mapping between a source object to be signified, and an independent target object or signifier". Huron mentions four basic types of representations: binary, alphanumeric, graphic, and sonic.

A terminological distinction can be made between representation and format. A computer data representation is a semantical definition of parameters used to describe (musical) information. "Format" refers to how the parameters are entered into computer memory. In particular, "file format" is a definition of encoding data in a computer file.

Many representations are bound to a single format. In some cases, however, there may be multiple (file) formats for a single representation, or a representation may be independent of any fixed format (see, e.g., Selfridge-Field 1997b: 565-572). For example, the Time Stamped Music File Format (Lassfolk & Lehtinen 1988) includes a definition of a binary file format but also allows the same information to be stored in three different types of ASCII text formats. The binary format was optimized for storage efficiency while the ASCII formats were intended to aid in data transmission and editing.

Representation and format are often used as synonyms, however, and an exact distinction between the two is sometimes difficult to make. For example, a binary file may have to be converted to ASCII text for E-mail transmission or for archival purposes. The result can be called an "ASCII encoded binary file". In that case, ASCII character strings represent binary codes. The distinction between "format" and "representation" may also be seen as a cultural one. Format (especially, file format) is a commonly-encountered term in computer science, whereas humanists or musicologists are more eager to talk about representation. A further distinction could be that representation refers to the logical content of the document, whereas format refers to the physical form or encoding in which content is stored.

In general, digital music representations may be divided into four basic categories, according to intended usage:

1. Input
2. Output
3. Storage
4. Data transfer

Input representations are optimized for entering information into a program. Output representations are products of computer processing tasks. For example, a notation program might use the PostScript graphics language as an output representation of a score, or it might employ a MIDI file as an output representation of a musical performance of the score. Storage representations are used to store programs' data structures in computer memory or into files. Transfer representations are used to transfer data between different programs or program versions.

As discussed in the terminological section of the previous chapter, the concept of “note” often demonstrates the semantic differences between various systems of representation. In the language of musical synthesis, “note” refers to a sound event that has an explicitly definable pitch, duration, onset time, and color (timbre). In MIDI, a note refers to a key of a keyboard instrument (such as a piano or organ). In MIDI, onset time and duration are explicitly definable, but exact pitch and tone are implicit. In music notation, as pointed out by Kurkela (1986: 101), a graphical symbol of a note is dependent of other objects, which Kurkela calls qualifiers. In representational systems for computer-based music notation, several forms of information (graphical, logical, and performance oriented) may be included in the same document.

Curtis Roads uses two adjectives for characterizing computer representations: *iconic* and *symbolic*. Roads bases his categorization on Thomas A. Sebeok’s definition of iconic and symbolic signs. Roads quotes Sebeok: “A sign is said to be iconic when there is a topological similarity between the signifier [the sign] and its denotata [what it represents]” (in Roads 1985: 405; Sebeok 1975: 242). Further, according to Sebeok and Roads “a symbol can be defined as ‘a sign without either similarity or contiguity but with only a conventional link between its signifier and denotata’” (Roads 1985: 406; Sebeok 1975: 247). According to Roads, a digital audio signal can be described as iconic while

music notation falls somewhere in between iconic and symbolic (Roads 1985: 406).

3.2 Design requirements

Huron (1992: 15-37) specifies a list of twelve properties of good representations. These properties may be used to characterize representations and to point out differences between representations. Huron lists the properties as follows:

1. Unique
2. Mnemonic
3. Consistent
4. Reversible
5. Terse
6. Non-cryptic
7. Structurally isomorphic
8. Context-free
9. Idiomatic
10. Explicit (interpreted)
11. Optional
12. Extendable

By *unique*, Huron means that "no two signifieds may share the same signifier" (ibid.: 16). This property guarantees error-free (or ambiguity-free) translation or interpretation of the representation. *Mnemonic* refers to associational relationships or mappings between signifiers and signifieds, said relationships meaning to help the user to learn or remember the representation. Huron mentions several types of mnemonic relationships. An example of literal mapping is the relationship between a graphic music notation symbol and its written name (for example, a fermata symbol and the word "fermata"). An "initial" mnemonic relationship occurs when a signifier is a letter that equals the written name of the signified; for example, when the letters W, H, and E represent durational values of a whole note, half note, and eighth note, respectively. "Pictorial" relationship refers to (approximate) visual resemblance between signifiers and signifieds; for example, the characters | and & as signifiers for a barline and a treble clef, respectively. Northern Indian tabla notation is an example of "onomatopoeic" mapping (Huron 1992: 20; Bel & Kippen 1992: 208).

Other types of mnemonic mappings listed by Huron are operational, semantic convention, isotonic convention, and topological correspondence. Isotonic convention describes the mapping of two quantitatively ordered parameters. For example, in MIDI, numbers 0 to 127 are mapped to the keys of a piano or syn-

thesizer keyboard. Isotonic mapping of dynamic markings would be achieved by assigning consecutive integer values for each dynamic level from *ppp* (or *pppp*, if desired) through *fff* (or *ffff*).

A representation is *consistent* when general rules and conventions of the representation are followed and applied without exceptions. Consistency is harder to achieve in terse representations than in verbose ones. This is due to limits placed upon the size and thereby by the number of possible signifiers for a given parameter. For example, the DARMS representation mixes initialism and isotonic convention in the encoding of durations (Huron 1992: 23).

In a completely *reversible* representation, the signifiers may be derived from the signifieds with the same effort required to derive the respective signifieds from the representation's signifiers. According to Huron, however, reverse mapping from signifieds to signifiers may be less mnemonic than is the opposite mapping (Huron 1992: 24).

A *terse* representation is often more efficient to use (for example, to write or to read) than a "verbose" representation. Huron uses a programming language to exemplify the advantages of terse representation (1992: 25). According to this example, it is more convenient to keep (local) variable names short rather than unnecessarily verbose. Here, "verbose" means that variable names are full, descriptive names rather than single letters or abbreviations as used in terse representations. The same principle is discussed by Kernighan and Pike (1999: 3). A terse representation is more economical in terms of space and often faster to comprehend, especially by an expert user or programmer. However, Kernighan and Pike point out that short names should be used for local variables whereas descriptive names should be used for global ones (ibid.).

Terseness is beneficial especially for a text-based representation intended for data input. The less keystrokes, the faster the encoding process. However, terse representations become easily – but not necessarily – more cryptic than verbose ones do. Obviously, *non-cryptic* representations are easier to understand and to process manually than are cryptic ones.

Structural isomorphism means that the signifiers are structurally organized similarly to the signifieds in the system to be represented. For example, a structurally isomorphic computer representation of music notation would somehow maintain the two-dimensional graphical structure of music notation. In *context-free* representations signifiers are self-contained and independent of other signifiers.

A representation is *idiomatic* if it takes advantage of practical features, "idioms", provided by computing environments. One such feature is the ASCII

character set. Another is the principle in which computer memory is physically organized as 8 bit bytes and other power-of-two quantities.

According to Huron, a representation is a result of interpretation. A representation is *explicit* when signifieds are mapped to signifiers so that they form a correct and sufficient interpretation of the target system for a particular application or purpose. For example, a different kind of explicit mapping may be needed in a representation intended for music printing purposes than for music analysis. According to Huron, it is important that the signifiers are well-suited (i.e., provide a correct interpretation) to the application for which they are used (Huron 1992: 31-34).

A representation is *optional*, if it allows the user to omit those attributes that are unnecessary for a particular task. For example, a user might want to omit pitch when encoding only a rhythmic structure. An *extendable* representation is not restricted to the set of signifiers defined by the designer of the representation.

It may be difficult for a music representation to meet all of the above requirements. For example, to be both terse and non-cryptic, may be impossible unless it is for a fairly simple or restricted target system. Further, if a representation is too idiomatic, then expressiveness, extensibility or even storage capability may suffer. Many binary formats, for instance, use a fixed number of bits for storing data lengths or other information about amounts. For example, the RIFF file format, which is used for many music and video applications, uses 32-bit data-length fields, which limits the storage capacity of RIFF to 4 gigabytes. This makes a single RIFF incapable of storing lengthy, high-resolution video or multi-track audio recordings.

Huron´s requirements provide a usable set of features for characterizing and assessing music representations. Still, some additional criteria and complementary remarks can be brought forward.

First of all, a representation should be complete (i.e., provide a signifier for every signified). This completeness may be difficult, if not impossible, to achieve in a dynamic target system. This situation emphasizes the importance of extensibility. Extensibility addresses the issue of completeness indirectly. To be usable, however, a representation should be complete enough to serve the purpose for which it is intended. An overly restricted initial set of signifiers may result in the use of terse or idiomatic signifiers that will later lead to inconsistencies when more signifiers must be added.

As mentioned by Huron, a central issue in representation is mapping between signifiers and signifieds. In general, four different types of mapping can be defined: (1) one-to-one, (2) one-to-many, (3) many-to-one, and (4) many-to-many. In one-to-one mapping, there is always exactly one signifier for each sig-

nified and vice versa. In one-to-many type mapping, a signifier is mapped to a group of signifieds. In many-to-one mapping, in turn, a group of signifiers is mapped to a single signifier. In many-to-many mapping the mapping is done between a group of signifiers and a group of signifieds. Of these, one-to-one mapping is the most explicit and unambiguous. One-to-one mapping also makes reverse mapping easier to achieve.

3.3 Music notation as a representation

The signifiers in music notation are graphical symbols, but what are their respective signifieds? If notation is regarded as a "visual analogue of sound" (see Grove 1980), then the signifieds can be regarded as sound events. To achieve a sound event from a note, an interpretation process is needed, wherein information other than the note symbol itself also have an effect on the produced sound event. Therefore, music notation is symbolic, rather than iconic, in respect to sound.

Although Huron´s requirements were intended for describing computer representation, they also can be used for "analogue" representations such as music notation. At least the following characterizations can be made of music notation as a representation.

In music notation, many signifiers (notes in particular) are unique only within a specific context. Thus music notation is highly context-dependent. For example, a dot may be interpreted as an augmentation dot, as part of a repeat sign, or as a staccato symbol, depending on context. Music notation contains both pictorially and literally mnemonic signifiers. Tempo and dynamic markings are examples of literally mnemonic signifiers. Pitch and time are represented roughly and pictorially with a two dimensional coordinate system, where the vertical axis represents pitch and the horizontal axis represents time. Thus, music notation is also structurally isomorphic in respect to time and pitch.

Music notation is reversible. A musical score may be created by transcribing a musical performance. However, for trained musicians, music is generally easier to read than to write. Music notation is terse. This can be seen from computer representations of music notation, which show the large amount of information embedded in the individual signifiers of music notation.

Music notation is both cryptic and partly inconsistent. Its signifieds are hard to understand without prior musical education. Among the inconsistencies are ways of representing durations of notes as a combination of several different symbols, which include noteheads, stems, flags, beams, and augmentation dots. Among the benefits of music notation are that it is both extendable and optional.

It is also explicit enough for many applications. One proof of this assertion is the large corpus of works of Western art music stored by means of such notation.

Computer representation of music notation can be regarded as "representation of representation" or *meta-representation*. Depending on the degree of uniqueness and explicitness of the particular computer representation, many aspects of music notation apply to the respective computer representation as well. Therefore, it is difficult, if not impossible, to design a computer representation of music notation that meets all of Huron's requirements. For example, if a computer representation is explicit and unique, can it be context-free at the same time? After all, music notation itself is not context-free. However, if uniqueness and explicitness are sacrificed, a context-free representation can be developed.

3.4 Digital audio signals

A digital audio signal is a technically simple way of storing music. A common technique for representing an audio signal is pulse code modulation (PCM). In a PCM encoding process, an analogue audio signal resulting, say, from a recorded musical performance, is sampled at a constant rate. Each sample represents an amount of air pressure, or a respective level of electronic voltage, at a specific point in time. Resolution, and thus audio signal quality, may be altered by a change of the sampling rate or of the amount of bits reserved for each sample.

A PCM signal does not interpret the content of the signal. It may originate from any kind of sound source, such as a solo musical instrument, a complete orchestra, speech, sounds of animals, or noise. Even inaudible signals, or static air pressure, can be encoded in PCM format. A PCM signal can therefore be described as non-symbolic in respect to musical content.

Methods of technically more advanced audio signal encoding have been developed for network transmission purposes, in order to reduce the amount of data relative to the amount of transmitted information content. For example, the MPEG-1 Audio Layer 3 format (MP3) uses spectrum analysis methods and psychoacoustic modeling to enable data compression by reducing inaudible components emitted from an audio signal (Fraunhofer 2004a). MP3 and similar techniques can be used to optimize transmission of acoustic signals. However, their encoding techniques are based on the physical behavior of hearing, not on the source or origin of the sound signal.

An alternative approach is offered by the Structured Audio encoding format of the MPEG-4 system (Fraunhofer 2004b; Lazzaro & Wawrzynek 2004). Structured Audio (MPEG-4 SA) is based on a virtual sound synthesis engine, which is controlled by a score or by MIDI data. An MPEG-4 SA document decoder produces an audio signal, but the content of an SA document is represented on a

symbolic level. The operational principle MPEG-4 SA is derived from sound synthesis languages used in computer music. The Music V program, described in the next section, can serve as an example of a synthesis language.

Many music notation programs offer some form of sound output, which may be at some stage converted into a digital audio signal. Few notation programs contain a sound synthesis engine for directly producing an audio signal. Instead, many programs can produce symbolic performance information, such as MIDI data, that can be used to drive an external hardware synthesizer or a synthesis program.

3.5 The Music V sound synthesis language

Through its data representation system, Music V (Mathews 1969) enables the user to construct synthetic instruments and supply them with a score that describes a detailed musical performance. The representation of Music V is a specialized programming language which provides instructions for synthesizing sound.

Music V instruments are constructed by the combining of *unit generators*, which can be regarded as software equivalents of analog synthesizer modules. Each unit generator performs a primitive signal generation or modification task. Among available unit generators are an oscillator, a set of filters, a random noise generator, and an envelope generator. For example, oscillators may be combined to modulate each other's frequency or output amplitude. Moreover, the output of several oscillators may be mixed to create complex sounds. In the Music V representation, each instrument is given a unique identification number so that it can be referred to during a performance. In principle, Music V imposes no limits on either the amount of unit generators per instrument or the number of instruments in an orchestra.

The musical performance is controlled by a set of timed events called *notes*. Each note consists of timing information (such as the onset time and duration of the note) and parameter data specific to the individual instruments. The use of nearly all of the note's parameters must be explicitly defined by the instrument.

Music V stores its data in a set of *records*. Each record is written on a single line of text and is divided into a set of *fields,* which are ordered from left to right. There, the first field represents the record type (also called an "operation code"). Individual types are reserved for each unit generator and for note events. The interpretation of other fields depends on the record type and/or instrument.

In note events the record type is "NOT". The second field contains the number of the instrument that plays the note. The third field contains the start time, in beats, of the note and the fourth field contains the duration of the note. The

rest of the fields may contain values for various synthesis parameters depending on the instrument's construction. An arbitrary amount of notes of any duration can be played simultaneously.

Typically, a note event contains parameters to specify pitch and amplitude specifically. Also, parameters might be added for controlling tone, vibrato rate and depth, and so forth.

Below is a sample program written for Music V. These programs are called scores in Music V terminology (Mathews 1969: 44). A Music V score consists of two parts: instrument definitions that comprise an orchestra and a set of notes to be played by the instruments.

```
INS 0 1 ;
OSC P5 P6 B2 F1 P30 ;
OUT B2 B1 ;
END ;
GEN 0.00 1 1 1 ;
NOT 0.00 1 1.00 1000 3.03 ;
NOT 1.00 1 1.00 1000 3.82 ;
NOT 2.00 1 1.00 1000 5.54 ;
NOT 3.00 1 3.00 1000 3.03 ;
TER 6.00 ;
```

In Music V scores, each record contains a *data statement*. A record is terminated by a semicolon. Records consist of fields that are separated by spaces. Fields are ordered from left to right and can be referred to by a numeric identifier. For example, the leftmost field's identifier is P1, the next field's identifier is P2 etc. Field P1 contains an *operation code* and other fields contain parameters for the data statement.

The first four records form a definition for instrument number 1. The INS operation code begins an instrument definition. P2 of the INS statement specifies the time at which the statement is executed. P3 specifies a numeric identifier for the instrument. In the example, the instrument definition starts at time 0 (i.e., immediately at the start of the performance). The instrument is given 1 as its ID number. The OSC record defines an oscillator that will be used as a sound generator. The parameters of the OSC statement are amplitude, pitch, output, waveform function, and sum, respectively. In the example, amplitude and pitch values are controlled by, respectively, the P5 and P6 fields of note records. In other words, each note specifies its own amplitude and pitch.

The oscillator output is copied to a memory block named `B2`. The oscillator waveform function is F1, which is defined in the `GEN` record on the fifth line. `P30` is used as a temporary storage space for the oscillator. The `OUT` record on

line 3 is used for connecting the oscillator output signal, stored in `B2`, to a main signal output memory block `B1`. The END record terminates the instrument definition. The `GEN` record defines a waveform function as one period of a sine wave.

In Music V oscillators, pitch is specified by a coefficient that determines a fundamental frequency. The frequency can be calculated as: $f_0 = (f_s * p) / N(F)$, where f_0 is the fundamental frequency in Hertz, f_s is the sampling rate of the resulting audio signal, p is the pitch coefficient and $N(F)$ is a memory "block size", i.e., the amount of samples reserved for storing wavetable functions (Mathews 1969: 127). In the example, a sampling rate (f_s) of 44100 and a block size of 512 are assumed. Setting p to 3.03 would yield a frequency of 260.98 Hz, which is close (with a small round-off error) to a middle C in an equally tempered scale with tuning reference level of A = 440 Hz. Adding more decimals to p would yield a more precise definition of pitch.

The `NOT` records in the score will play four consecutive tones on the instrument 1. A respective score in music notation is shown in figure 3-1.The `TER` record terminates the score at beat 6.00.

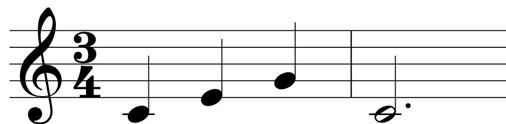


Figure 3-1: An example score

Semantically, a note in Music V is equivalent to a sound event. Each note has an explicitly defined onset time and duration. The only external variables that affect a note are the digital audio signal's sampling rate, the memory block size and performance tempo. If and how the sound event is audible is determined by the instrument definition. All parameters in instruments are explicitly defined and deterministic except when a random function (Mathews 1969: 128-129) is used to produce sound or to control a parameter of an instrument. Music V does not include a signifier for a "rest". A rest is produced implicitly by not defining a note for a desired length of time.

The Music V score is a symbolic representation although it provides fairly explicit definition of sound events through notes and instrument definitions. For

example, Music V scores do not contain detailed descriptions of the sound synthesis algorithms needed when the score is translated into a sound file. Therefore, it is possible, that different implementations of Music V-compatible synthesis programs might produce slightly different-sounding translations of the same score.

The influence of Music V and its predecessors is demonstrated by the large amount of other unit-generator-based synthesis languages. Roads, for example, lists 20 synthesis languages, including Mathews' Music III, IV and V (Roads 1996: 789-790). A modern and widely used member of this family is Csound by Barry Vercoe (Boulanger 2000).

3.6 MIDI and MIDI Files

MIDI (Musical Instrument Digital Interface) was designed in the early 1980's as a means of controlling music synthesizers and other electronic keyboards. The original MIDI 1.0 specification (MIDI 1985 [1983]: 114-126) contained definitions for both a hardware interface and a data format. The data format consists of MIDI messages that are transmitted between instruments or other devices equipped with MIDI hardware interfaces. Later versions of the specification have included extensions to the data format and a file format for storing MIDI data.

Two central MIDI messages for controlling musical instruments are called Note On and Note Off. Both messages contain three parameters: a MIDI channel, a key number, and a key-stroke velocity value. The key number, an integer between 0 and 127, specifies a key on a chromatic piano keyboard. MIDI thus allows the use of a 128-key keyboard, at the maximum. The key number 60 is specified as "middle C". No means of explicit definition of pitch was given in the 1983 specification. In later revisions, a way of defining different tuning systems was included.

A MIDI Note On event consists of three bytes: one *Status Byte* and two *Data Bytes*. As an example, let us say a Note On event contains the decimal values 144, 60, and 64. There, the first value (144; i.e., the status byte) specifies the MIDI message type and, in the case of a Note On event, also the MIDI channel. Status byte values of 144 through 151 are reserved for Note On on channels 1 through 16, respectively. In the example message, the data byte values 60 and 64 specify the key number and velocity, respectively. A MIDI channel, ranging from 1 to 16, is encoded in the status byte.

An example Note Off message contains the decimal values 128, 60, and 64. There, 128 is a Status byte specifying a Note Off for MIDI channel 1. (Status bytes 128 through 143 are reserved for Note Off on channels 1 though 16,

respectively.) The values 60 and 64 specify, respectively, the key number and a Note Off velocity (the speed at which the key is released).

MIDI was originally a real-time system. That is, MIDI messages were intended to be transmitted and performed instantly. In later revisions of the specification, a file format, Standard MIDI Files, was included for attaching timing information to events. Still, the MIDI communication protocol itself remains mostly real-time.

Besides event timing, MIDI Files enables the storage of logical information that is not part of the MIDI protocol itself. Examples of these include time signature and lyrics. Still, as pointed out by Hewlett and Selfridge-Field (1997: 68) as well as Haken and Lippold (1993: 43), MIDI or MIDI Files offers insufficient detail to be used as a music notation representation. Extensions have been proposed for including notation information in MIDI (e.g., Nordli 1997; Cooper *et al.* 1997). These extensions have, however, not been included in the official MIDI specification.

MIDI is a terse representation. This is an important requirement, because events are transmitted in real time between devices, and there should not be a noticeable delay in the reaction time of a receiving instrument to the messages sent by a MIDI keyboard or other controller.

In the MIDI protocol, all data is encoded in groups of 8-bit bytes, where one bit is reserved for indication of status or data byte. The remaining 7 bits are left for storing the actual data. This leads to difficulties, when large data entities have to be folded into 7 bit chunks. MIDI does not even specify a generic way of encoding and decoding typical 16 bit, 32 bit, or 64 bit computer data types. Therefore, each application or extension of MIDI, must specify its own way of solving this common problem.

3.7 MusicKit

MusicKit (Boynton & Jaffe 1991) is an object-oriented software system originally distributed as part of the system software of NeXT computer workstations. Music Kit contains an application programming interface (abbr. API) with classes for sound synthesis as well as for processing and storing musical events. MusicKit also included a text-based scoring language, which is a mixture of techniques used in MIDI and synthesis languages. In the NeXT system software, a separate class system, called SoundKit was provided for audio signal storage and editing.

Syntactically, the MusicKit score language resembles a modern statement and expression -based programming language, such as C or Pascal. This is a departure from the record/field-type syntax of Music V and many other synthe-

sis languages. Semantically, MusicKit borrows features from both synthesis languages and MIDI.

In MusicKit, Note is semantically equivalent to a sound event. Onset times and durations of sound events are determined by “noteOn” and “noteOff” events, or alternatively by “noteDuration” events which are encoded in a timed event stream. Pitch may be specified either as a MIDI key number or as a fundamental frequency value. The score may be used to control both external MIDI instruments and the computer’s internal sound synthesis engine.

Below, an example of a MusicKit score is presented. The first line contains comment, preceded by a “`//`” delimiter. The rest of the score consists of *statements.* A statement ends with a semicolon character (`;`) and may be written on one or more lines. The first statement in the sample score specifies a performance tempo in beats per minute. The next statement defines a named “part”, `p1`. The part is given an instrument in the next statement. There, the expression `synthPatch:"midi"` specifies, that a MIDI instrument is used to perform the part.

The `BEGIN` statement marks the start of a stream of timed events which ends with an `END` statement. The event stream contains a noteOn statement and a noteOff statement. The noteOn statement is preceded with a timing statement “`t 0`” that sets the time of the performance to 0 beats. This means that all succeeding events until the next timing statement are performed at beat 0. The noteOn event belongs to the previously defined part `p1`. In the parenthesis following the noteOn expression, is a numeric identifier for the note. Each note may be given a unique identifier so that the note may be referred to in other events. In the noteOn event, pitch is specified with the expression `keyNum:60` as a MIDI note number. Velocity is specified as a MIDI value with the expression `velocity:64`.

With the timing statement “`t +1`” time is advanced by one beat (i.e. 1/60th of a second as defined by the preceding tempo setting). Next, a noteOff event for the note ID 1 is given. Unlike in MIDI, a key number does not have to be specified since the note already has a unique identifier.

```
// A sample MusicKit score

info tempo:60;

part p1;

p1 synthPatch:"midi";
BEGIN;
t 0;
p1 (noteOn 1) keyNum:60 velocity:64;
t +1;
p1 (noteOff 1);
END;
```

A MusicKit note is an extension of the MIDI Note concept. As one important extension, MusicKit supports uniquely identifiable notes. This enables one to change parameters of a note in between noteOn and noteOff events. For this purpose MusicKit includes a specialized event type called noteUpdate. MusicKit allows the specification of pitch or related information in alternative ways. One way, a key number, is used in the above example. Other alternatives are fundamental frequency and key name. A key name allows one to specify a name on a note (c, d, e, etc.) and an octave range. For example, the key name c4 is equivalent to MIDI key 60. Unlike with MIDI key numbers, key names enable explicit definition of enharmonic variants. For example, the key name cs4 (c sharp, 4th octave) is logically different from df4 (d flat, 4th octave). When used to control a MIDI instrument both are interpreted as key number 61, but if the representation is translated into music notation, the enharmonic distinction can be preserved. Still, a MusicKit score does not provide sufficient information to be used as a translation format or input language for music notation programs. Among the lacking parameters are key and time signature, lyrics, and stem direction.

MusicKit is more literal than MIDI and, on the other hand, less terse. MusicKit is also somewhat cryptic and idiomatic in respect to the synthesis algorithms available in the NeXT workstations. These features become apparent in scores that are more complex than the above example and which use the internal synthesis algorithms instead of, or in addition to, MIDI control of external instruments.

3.8 The SCORE music notation program

SCORE, designed by Leland Smith (SCORE 1992a, 1992b, Smith 1997), is an exceptionally well-documented commercial notation program, especially con-

cerning the contents of internal data structures and operational logic. The SCORE system is implemented for IBM PC compatible computers running the MS-DOS operating system. SCORE is based on an earlier program for mainframe computers, called MSS (Smith 1972). Hereafter, SCORE, written in all capital letters, refers to the SCORE program as a distinction from the general musical term "score" or from a class "Score" (with an initial capital letter) as described in Chapter 7.

As its data representation system, SCORE uses a database approach similar to that of Music V. Whereas the database of Music V consists of sound synthesis parameters, SCORE's database consists of music notation parameters.

SCORE is written in the FORTRAN programming language. The limited data structure capabilities of early FORTRAN languages are reflected in SCORE's data representation and behavior. Due to the lack of user-definable data structures in FORTRAN 66 and 77, SCORE stores most of its data in arrays of floating point numbers. Text strings are stored separately from the floating point arrays. Parameters are accessed through their position (i.e., index) in an array. As in Music V, parameter fields are named P1, P2, etc. according to their index number.

As in Music V, SCORE's data representation system can be described as a simple database where data is stored in *records*, with each containing a set of fields. In a record, the first field, named P1, is a "code" number. Individual code numbers are given to notes, rests, clefs, staves, time and key signatures, etc. Some closely related symbols are given a shared code. For example, slurs, ties, tuplet brackets and endings share a common code number. The second, third and fourth fields are common to all record types. The second field, named P2, is the number of the staff associated with the symbol. Fields P3 and P4 contain the horizontal and vertical coordinates of the symbol, usually relative to the associated staff's location on the score. Other fields contain parameters specific to the type of record.

SCORE supports a text-based input language for entering notation from the computer terminal or from a file. The graphical layout may be adjusted manually by editing the numeric values of the record-based data storage format. SCORE also has a set of commands for automatically adjusting the spacing of notes, height of stems, length of beams etc.

Internally, each SCORE record is stored in an array of floating point numbers. Even the record types are internally stored as numbers. Lyrics and other text symbols are an exception to this principle.

Below is a representation for the note example presented in Figure 3-1 as a SCORE Parameter List File (SCORE 1992b: 225). Each SCORE symbol is printed on a separate line. Above the parameter list is a line that lists names of

the parameter fields from P1 to P9. The symbol code number is stored in P1 followed by a staff number, horizontal and vertical (if needed) coordinates, in P2, P3, and P4, respectively. Coordinates are expressed in staff steps and are relative to the position and size of the staff that the symbols are printed on. Coordinates may use the decimal part of the floating point number to reach higher resolution than that of a staff step. Fields P5 through P9 contain parameters specific to the individual code numbers.

P1	P2	P3	P4	P5	P6	P7	p8	P9
8.	1.0	.000	.00	.00	70.00			
3.	1.0	1.500						
18.	1.0	8.999	.00	3.00	4.00			
1.	1.0	16.499	1.00	10.00	.00	1.0000		
1.	1.0	26.678	3.00	10.00	.00	1.0000		
1.	1.0	36.857	5.00	10.00	.00	1.0000		
14.	1.0	47.456	1.00					
1.	1.0	50.788	1.00	10.00	1.00	3.0000	.00	10.00
14.	1.0	70.000	1.00					

The example contains dedicated records for one staff (code 8), a clef (code 3), a time signature (code 18), four notes (code 1), and two barlines (code 14). SCORE's coordinates are expressed in staff steps.

In SCORE parameter list files, parameter field values are printed for each code, up to the last field having a nonzero value. Zero values are printed as "`.00`".

Below is a list of the code numbers used in SCORE version 3. The code numbers range from 1 to 18 with the exception of number 13 which is not used in SCORE version 3. Leland Smith marked code number 13 as "reserved" (Smith 1997: 257).

1. Notes
2. Rests
3. Clefs
4. Lines and hairpins
5. Slurs, ties, tuplet brackets, and endings
6. Beams and tremolandi
7. Trills, ottavas, and pedal marks
8. Staves
9. Symbol library
10. Numbers (rehearsal letters)
11. User symbol library
12. Special shapes
13. (“reserved”)
14. Barlines, braces, and brackets
15. Importing postscript programs
16. Text
17. Key signature
18. Time signature

Note items have 17 parameter fields, or 13 in addition to fields P1 through P4, which have similar meaning in all codes. Fields P5 through P17 are used for specifying values for parameters such as stem direction, type of accidental and displacement (P5), note head type (P6), rhythmic duration of a note (P7), stem length (P8), number of augmentation dots and flags (P9), etc. In the above example, the P5 value of “10.” means “Up stem/No accidental” (SCORE 1992b: 9). P6 of 1.000 (in the first three notes) means rhythmic duration of a quarter note and 3.000 (in the last note) means three times the duration of a quarter note (i.e., the duration of a dotted half note). The P8 value “.00” in the last note means “Normal stem length” (ibid: 15) and the P9 value “10.” means “Single dotted note”.

Rest records have 15 parameter fields. Clefs have 7 parameter fields. In the latter, for example, P5 is used to specify a clef type from nine different choices. In the example, the P5 value 0 indicates a treble clef.

The graphical approach of SCORE has advantages and disadvantages. For example, SCORE permits the moving or reshaping of most symbols independently of each other. Any individual note head may be resized or moved with great precision both horizontally and vertically. Changing the position or size of one symbol does not move or resize the other symbols in the score unless a specific formatting command is entered by the user. On the other hand, the level of

automation is low – even most features stated in the SCORE manual as "automatic" must be executed manually by entering a dedicated command.

In SCORE, a note is equivalent to a graphical note symbol including a note head, a stem, augmentation dots, and various modifiers including accidentals and articulations. As a representation, SCORE is explicit in respect to graphical aspects of music engraving. All symbols in SCORE are explicitly positioned on a two-dimensional coordinate system. Logical and performance data is represented either implicitly by graphical symbols or stored as parameters of graphical symbols.

Many editing operations, however, leave the score in a graphically distorted state. For instance, stems may, as a result of a transposition operation, become separated from their respective note heads. During an engraving process, the score must be occasionally fixed with manually typed commands that check the relationships between musical symbols and adjust the layout accordingly. Nevertheless, SCORE has a rich set of features and is suitable for professional-quality music engraving. It has thus been used by many music publishers (Smith 1997: 252). In particular, SCORE's record-based internal structure is thoroughly documented, which makes it a suitable subject for study.

SCORE is partly inconsistent. Many complex or rarely needed layout situations are encoded differently from simple or commonly used ones. This is partly due to the long development process from MSS through several versions of SCORE itself. SCORE's inconsistencies are also partly caused by the primitive data structure capabilities of the FORTRAN programming language.

3.9 Lime and Tilia

Tilia is a data structure system used in the Lime notation program (Haken & Blostein 1993). Unlike SCORE, Tilia allows records to contain arbitrarily typed data, including types other than just floating point numbers. Tilia stores graphical, logical, and performance information in its data structures. Tilia is, however, mainly logically-oriented: both detailed graphical information and performance data (i.e., MIDI data) is mostly generated automatically by the Lime program.

Tilia is based on linked lists. The elements of the lists are called nodes. The size of each node is 32 bytes (256 bits) divided into fields of various sizes. In each node, the first two fields, 32 bits each, are reserved for pointers to the next and previous node in the list. The remaining 24 bytes are used for storing the type (called "kind" in Tilia's terminology) of the node and a set of type-specific

parameters. A “note” node has a set of parameters different from, for example, a “text” node.

Musical data organization in Tilia is based on voices. In Tilia, each voice is stored in its own list. Each list starts with a “Voice Info” node containing a numeric identifier for the voice. The Voice Info node is followed by a “PRINT” node indicating a staff ID number. Tilia allows more than one voice to be displayed on the same staff.

Note nodes represent both notes and rests. A “rest” field determines if a note node represents a note or a rest symbol. Pitch is represented as both a playback pitch and a notated pitch. Playback pitch is coded with a combination of a MIDI key number and a fractional pitch value that even enable microtonal resolution if needed. Notated pitch is coded as a combination of a numeric value indicating a note name (C, D, E, etc.), an octave range, and a parameter for specifying the amount and type of accidentals attached to the note.

Lime is able to compute a playback pitch from a notated pitch. Detailed graphical information, such as coordinates for adjusting the placement of note symbols, are not stored in note nodes. Duration is also divided, by dedicated fields, into playback and notated duration. An “end chord” field is used to specify whether the next note in the list belong to the same chord (Haken & Blostein 1993: 46-50).

Nodes such as “smove”, “cmove”, “space”, and “zone” are used for specifying various graphical layout adjustments. Smove controls vertical placement of a staff. Cmove, space, and zone control horizontal placement of notes.

Tilia provides dedicated nodes for clefs, key signatures, barlines, beaming, etc. During a music formatting process, Lime inserts temporary “data” nodes to store parameter settings (ibid: 55-56). Tilia, as well, supports links between nodes belonging to different lists.

3.10 The Music preprocessor

Music by Eric Foxley (1988) is a text-based music notation language. Foxley’s *Music* was designed as a preprocessor for the UNIX troff typesetting program (Kernighan *et. al.* 1987) and follows the principles of other troff preprocessors such as eqn and pic. *Music* provides means for writing music notation with compact text expressions. Default settings are for commonly-used notation situations so that they do not have to be coded explicitly. This reduces the amount of text needed.

Below is version of the note example of figure 3-1 as a *Music* program:

```
.MS
timesig = 3 4;
key = c;
bars = 2.
c e g | c>. |
.ME
```

The program is embedded in a troff document where it is surrounded by ".MS" and ".ME" statements. A typical *Music* program consists of a *header* and a *score*. The header contains settings for variables for the piece of music such as a time signature, a key signature, and the amount of bars in the score. The header is terminated by a full stop character. The example score contains three quarter notes c, e, g, and a dotted half note c in the default octave starting at middle c. The octave range can be changed by changing the value of the "octave" variable. An octave range may be specified for individual notes by appending upward or downward arrow characters to the note name (e.g "c↑", "c↑↑" etc.) to transpose the note by one or more octaves up or down from the default octave. Note length is specified as the denominator value of the time signature and may be changed for individual notes by appending < (one value shorter) or > (one value longer) characters to the note name. To print a dotted note, a dot is appended to the note name. In the example, the default note length is quarter note.

In *Music*, note is semantically equivalent to a music notation symbol that consists of a note head and an optional stem. Note names are written explicitly for each note. Octave range and note length depend on modifications made to preceding notes, and on preceding time signature and variable settings.

Other text-based notation languages include DARMS (Selfridge-Field 1997a), GUIDO (Renz 2000), and CMN (Schottstaedt 1997). DARMS is an early music representation which has several dialects developed for specialized uses. GUIDO is a logically-oriented data input language for the GUIDO Notation Engine software implemented in C++ and Java. CMN (abbreviated from common music notation) is a notation language written in Common Lisp Object System (abbr. CLOS) (Steele 1990). CMN's representation is a logically-oriented notation language following the syntax of LISP. (Schottstaedt 1997.)

3.11 NIFF: Notation Interchange File Format

A committee was founded in the early 1990's by a group of researchers and music notation software developers to develop an interchange file format for

music notation data. The format was named Notation Interchange File Format (Grande 1997), abbreviated as NIFF or sometimes NIF. NIFF was intended to be used for data exchange between software from different manufacturers. One of the potential uses for NIFF was to enable the development of applications specialized for a limited task, such as recognition of printed or hand-written scores. Such applications could, by supporting NIFF as a data export format, omit music editing or printing functionality.

Although NIFF is a partly graphically-oriented representation, NIFF regards the logical component of music notation as the most important component to be preserved in data transmission. In NIFF-based data transmission, graphical positioning details can be left to be calculated automatically by the receiving program.

The NIFF file format is based on the Resource Interchange File Format (RIFF), a binary file format designed by Microsoft and IBM. Other applications of RIFF include Microsoft's WAVE audio file format and the AVI video file format. In RIFF, data is organized as "chunks". Each chunk consists of a four byte identifier, a 32-bit integer size field, and an arbitrary amount of data. The size field specifies the chunk size in bytes. A chunk may contain other chunks as its parts, thus allowing the construction of hierarchical structures. Another structure is a LIST of consecutive chunks (Microsoft 1991).

NIFF uses RIFF chunks for storing various structures related to music notation. Individual types of chunks are specified for constructs such as Score, Part, Voice, System, Staff, and Time-Slice. A Score may contain several instrument Parts, which may contain an arbitrary amount of Voices. Parts are printed on Staves, which are grouped into Systems. Time-Slice is used to group notes that appear on a specific point in time within the score (Grande 1997: 494-495).

NIFF specifies also distinct chunk types for various music notation symbols. These include rest, barline, key signature, time signature etc. Notes are comprised of separate stem and notehead chunks. A stem may be followed by one or several noteheads which form a chord. Also, a stem with no noteheads is allowed.

In NIFF, a Note's placement is expressed by its vertical position on staff expressed in staff steps. This approach is similar to SCORE. The pitch or name (and octave) of the note is therefore dependent on previous symbols. Note length ("logical duration") is defined with a numerator/denominator pair (i.e. 1/8 for an eight note).

Below is an example of a simplified NIFF representation converted to text pseudo-code in a manner similar to Cindy Grande's examples (Grande 1997). NIFF and generic RIFF constructs that are not crucial in demonstrating NIFF's fundamental syntactic and semantic principles were omitted from the example.

Each chunk is written on a separate line. On each line, a chunk type is followed by the chunk's parameters and values separated by commas.

```
Staff-header, number of staff lines=5
Clef, shape=G clef, staff step=2
Time-signature, top number=3, bottom number=4
Time-slice, type=event, start time=0/4
Stem
Head, shape=filled, staff step=-2, duration=1/4
Time-slice, type=event, start time=1/4
Stem
Head, shape=filled, staff step=0, duration=1/4
Time-slice, type=event, start-time=2/4
Stem
Head, shape=filled, staff step=2, duration=1/4
Barline, type=thin line
Time-slice, type=event, start time=4/4
Stem
Head, shape=filled, staff step=-2, duration=3/4
Augmentation-dot
Barline, type=thin line
```

In NIFF, each staff is stored in a RIFF LIST. A Staff Header chunk starts the LIST and is followed by other chunks as shown in the example. As the example shows, NIFF encodes both logical and graphical information. Representation of pitch is graphically-oriented, and duration is specified as a mixture of logical and graphical information.

3.12 Music notation markup languages

The Standard Music Description Language project (Sloan 1997), abbreviated as SMDL, was an ambitious standardization effort in the field of musical data representation. Based on the standard of SGML (Standard General Markup Language), SMDL is a text-based representation aimed at encoding virtually any kind of musical document.

As already described in the previous chapter, SMDL specifies several "domains" for categorizing musical information. The central or primary domain in SMDL is called *Cantus*. It describes a core of the musical work excluding

graphical symbols or improvisation. The other domains may use Cantus as a basis and include only the differences (such as timing differences) from Cantus.

Other domains in SMDL are called gestural, analytical, and visual. The gestural domain is intended for representing a performance of a musical work. The analytic domain is intended for representing an analysis of and commentary on a musical work. The visual domain is intended for encoding various graphical notations or displays of a work. These secondary domains can be regarded as derivatives or interpretations of Cantus. An SMDL document may include several different performances, analyses, or notations of the same Cantus (Sloan 1997: 470-472).

In SMDL, a note is represented by pitch and duration. Both of these so-called elements may be represented in several alternative ways. Pitch may be represented, for example, as a frequency, or as a named note within an octave. Duration may be represented, for example, in real time (e.g., in minutes or seconds) or in virtual time (e.g., in fragments of a beat). Other SMDL basic elements include articulation and ornamentation, dynamics, and timbre. SMDL allows notes to be organized into threads (voices) where each thread may have a different timbre (Sloan 1997: 472-473).

The Extensible Markup Language (XML) is a meta-language for creating markup-type documents (Eckstein 1999). Syntactically XML is similar to SGML. XML documents are conventionally text-based, although XML supports inclusion of binary data under certain conditions. XML allows the definition of a syntax for practically any type of content, including literature, hypertext, databases, graphics, multimedia, and music.

XML documents are organized as *elements* that may contain data or references to external documents. Each XML element is surrounded by a “tag” and “end tag” -pair. A group of tagged elements may be in turn surrounded by a collective tag pair. Nested tags form a hierarchical element structure with a single document-level tag on the top level of the hierarchy. Tags may also refer to other documents, making XML suitable for hypertext-like applications. Since XML documents are, by convention, text-based, they may be created and modified with a general-purpose text editor.

There are several published XML applications for representation of music notation including NIFFML (Castan 2001), MDL (Roland 2001), and MusicXML (Good 2001). NIFFML is an XML implementation of NIFF. NIFFML enables the writing and display of NIFF documents as text.

Both MDL and MusicXML were designed for data interchange between music applications. MusicXML is a logically-oriented representation. There, music may be organized in two, alternatively structural ways: “part-wise” or “time-wise”. In part-wise organization, notes are stored within “parts”. Time-

wise organization is based on "measures". "Note" tags enclose subtags; for example, pitch (expressed as "step" and "octave"), duration type (whole-, half-, quarter-, eighth-note, etc.), dots, stem direction, and lyrics. MusicXML is not designed for representation of detailed graphical layout (Good 2001). MDL is intended as an interchange format for "music notation, performance, analysis, and information retrieval applications" (Roland 2001: 126). For example, MDL supports representation of musical structures.

3.13 Object-based music representations

Object-based representations use objects as the principal organizational constructs. Objects may contain arbitrarily-typed data as well as operations for manipulating the data. MAX (Déchelle 2004) is an object-based graphical programming language used for building interactive musical performance applications. There are several versions of MAX. Some newer versions, including jMax (Déchelle 1999) and Max/MSP (Cycling74 2004), support processing of both MIDI events and audio signals. PatchWork (Laurson 1996) includes a graphical programming environment somewhat similar to MAX. PatchWork is intended primarily for music composition rather than for performance. PatchWork supports several forms of graphical and numerical representation of musical structures, including music notation.

Glen Krasner described ways for using the object-oriented Smalltalk-programming language for musical representation and data processing. Krasner presented a scheme for representing both musical data and actors involved in data processing as software objects. Among these objects were an "Orchestra", a "Player", a "Score", and an "Instrument". (Krasner 1991 [1980].)

MODE is an object-oriented, musical data processing system written in the Smalltalk programming language. MODE allows the processing and display of musical data in various forms, including graphical representations. Notes may be displayed as common music notation or as a piano-roll style, pitch-time representation. MODE also supports graphical wave-form display of digital audio signals, as well as the graphical display and editing of musical phrases (Pope 1991b). Object-based representations are discussed further in Chapter 6.1.5.

3.14 General-purpose graphical representations

General-purpose graphical representations may be used also for representation of music notation. In general-purpose representations, however, graphical information is encoded on a lower semantic level than that of dedicated musical rep-

resentations. In particular, logical and/or performance information is difficult, and sometimes impossible, to store and process. Also, for example, the alignment of note heads in relation to staves or stems is typically more difficult to arrange in a general-purpose representation than in a graphically-oriented musical representation.

Purely graphical representations are used in music publishing, however; for example, by some music engravers involved with contemporary music. One such application is to use a general-purpose graphics program to add custom symbols to a score initially produced with a notation program.

Graphical representations can be divided into two basic categories of representation: pixel-based and vector-based. A pixel-based graphics representation can be described as an iconic representation of a picture, while a vector-based one is more symbolic.

An example of graphical representations is the Adobe PostScript system (Adobe 1986). PostScript provides a text-based graphical programming language that can be used to control output devices such as printers. PostScript can also be used as a general-purpose programming language.

PostScript allows processing of both vector and pixel graphics. Vector graphics are constructed of "paths" that are formed by coordinate points and connecting lines or curves. Visual graphical shapes are created by either "filling" an area bounded by a path with a desired color or by "stroking" a line that follows a path. The width and color of stroked lines can be controlled, and the line may be solid or dashed. Fonts are available for optimized processing of text characters. Specialized fonts, such as the Adobe Sonata font, are available for displaying music notation symbols.

Adobe Illustrator is an interactive drawing program that is based on the PostScript imaging model. Adobe Illustrator may be used also for creating and processing musical scores. As general-purpose programs in general, Illustrator is not able to store any form of logical data or even general information about music notation. Placement of notation symbols, including spacing, must be done manually. Moreover, construction of many music notation symbols (such as slurs or braces) is difficult unless the symbols are first created in a music notation program and then imported to Illustrator. On the other hand, Illustrator and similar graphics programs offer a high degree of control of graphical details.

3.15 Comments on data representation systems

The differences in the focus or orientation between the above-described representations is demonstrated by their conception of constructs such as note and rest, or of parameters such as pitch and time. Most music representations include

some conception of "note", but with varying meanings. To performance-oriented representations, a note may mean a sound event or a key of a keyboard instrument. In graphically-oriented representations, "note" refers to a graphical symbol. In logically-oriented representations, the concept of note lies somewhere in between those two meanings. Graphically-oriented music notation representations contain a signifier for a rest, while performance-oriented representations may exclude it. In logically-oriented representation, the existence of a rest depends partly on whether the representation is geared for music notation or only for rendering a sonic performance.

For example, the Music V language is extremely performance-oriented, while SCORE's internal representation is profoundly graphics-oriented. In Music V, pitch is defined in terms of fundamental frequency. In MIDI, exact pitch is dependent on the MIDI instrument. In contrast to a frequency value, a key number is a step towards logical representation of pitch.

Most of the music notation languages and notation interchange representations described above are primarily based on the description of logical information. They leave graphical details to be generated automatically. Eric Foxley's *Music* is logically-oriented, but provides some control over graphical layout. NIFF allows both detailed graphical control and logical constructs to be included in the same document. In this sense, NIFF is a hybrid logical/graphical representation.

Many text-based music notation languages are intended to be written manually and thus are optimized for compact expression of notation situations. In such cases, the support of detailed graphical control is often limited. One reason for this is that to allow explicit control of more than one data component (such as both graphical and logical) would easily make the representation itself too complex to learn and remember.

For a musician, a logically-oriented notation language can be more compact, and is more intuitive to write manually than is a graphically-oriented one. Therefore, logically-oriented representation may prove more efficient and practical as data input representation. Yet, and especially in internal data representation as well as in data exchange, detailed graphical information needs to be preserved, especially if the manual labors of the music engraver are to be retained. The analysis model of music notation presented in this study is based on graphical information. There, logical information is considered secondary, optional, and dependent on graphical data. Semantically, however, the vocabulary of music theory and music engraving is preferred over the vocabulary of general graphics.

A common feature between iconic representations, such as digital audio signals and pixel graphic, is that they are easily interchangeable with their analog counterparts. A digital audio signal can be achieved with an automatic conver-

sion process of an analog signal. A reverse, equally simple conversion process can be used to convert the digital signal back to analog format. In many circumstances, the encoding and decoding process can be performed with insignificant deterioration of sound quality. This is similar to the way a picture can be converted to a pixel-based digital representation with a scanner device and then back to analog form with a printer. For symbolic representations, either the encoding or decoding process, or both, require a higher degree of interpretation of the signifieds and/or signifiers.

A notation program typically needs to handle more than one representation. For example, SCORE uses five representations, each for a specific purpose: (1) SCORE input language for data input, (2) internal database representation for data storage, (3) pixel graphics for displaying notation on a computer display, (4) the PostScript language for printing, and (5) MIDI, for playing music with a MIDI instrument. The input language is used as an intermediate representation to produce the internal database representation. The other three representations are created from the database representation by automatic interpretation processes. Music notation programs may also provide the means to translate MIDI or printed graphics into their internal representations. Conversion between differently-oriented representations is, therefore, the challenge that all notation programs must face.

Chapter 4
Object-oriented software engineering

This chapter describes the basic principles of object-oriented software technology. Included are a brief description of terminology as well as a historical overview of object-oriented programming languages and of some formal software engineering methods. The main features of Unified Markup Language (UML) are described with emphasis on the features needed for presenting the analysis model of music notation in Chapter 7. Three formal object-oriented methods are discussed: OOA by Coad and Yourdon, ROSE by Grady Booch, and OMT by James Rumbaugh et al. The similarities and differences of their basic principles, terminology, and methods of classification are described.

It is difficult to present a simple definition or description of object-oriented software engineering. Here, such methodology is described through the above-mentioned three example methods. Each method has a different approach to the subject and a different perspective on how object-oriented techniques differ from other techniques. In particular, each method has a different approach to the carrying out of an object-oriented software development process, and to each stage within a complete process. The methods have also been influenced by the principles and terminology of object-oriented programming languages, the first of which appeared before any of the above-mentioned methods were developed.

The UML notation used in this study is described here in order to make the study self-contained in this respect. UML is a rich language, only a part of which is needed for our purposes. Also, the language is used so widely that the ways of using it are likely to differ among users. Hence, this chapter also presents a description of UML as it is used in this study.

The first programming languages that can be described as object-oriented were developed in the late 1960's and early 1970's. It is difficult to define exactly what makes a programming language object-oriented, but some distinctive common features can be mentioned. Perhaps the most distinctive one is the ability to form modular constructs, called objects, by means of data and functionality. Another common feature is the ability to form objects that contain other objects as their parts. Yet another is the ability of one object to "inherit"

data and functionality from another object. Other features include the ability of objects to communicate with each other and to form various kinds of relationships.

During 1990's object-oriented programming and related software engineering methods became a central paradigm in the software industry. Object-oriented programming provides a formal method for creating software systems that simulate or mimic real-life systems. Some of the main benefits of the object-oriented paradigm are maintainability and reusability of the program code. At the same time, object-oriented methodology emphasizes the importance of systematic software design

Computational efficiency is not considered the primary aim of object-oriented programming. Object-oriented programming languages often produce computationally less efficient programs than do traditional function call based languages. Instead, object methodology has been viewed as a solution for handling increased complexity in software systems (Cox & Novobilski 1991: 3-29). Several formal methods have been developed for the production of object-oriented software. Unified Markup Language, or UML, is now widely used both as a production tool in the software industry and as a formal methodological tool in computer science.

4.1 Basic concepts and terminology

Object-oriented software engineering uses *objects* as basic units of software construction. An object is a combination of data and a set of *operations* that can manipulate the data. Data are organized as a set of attributes. An object may have various types of attributes; for example, numeric variables, text strings, truth values, or other objects. A certain set of values of an object's attributes is called the *state* of the object. Operations are constructed from executable program code. Operations form the *behavior* of an object. Collectively, attributes and operations are called the *properties* of an object. Objects communicate by invoking each other's operations.

In a purely object-oriented system, all the system's data are stored within objects. An object may allow or deny other objects to have access to all or part of its attributes (or operations). One common practice is for an object's data to be accessed only through its operations. This kind of data protection or data-hiding principle is called *encapsulation*.

Class is the definition of an object. Classes are typically created by programmers using an object-oriented programming language. In a class-based object system, an object is an *instance* of a class. Instances are created and destroyed while the program is executed. Typically, a class may have more than one

instance. Although class is a typical feature of object-oriented programming languages, some languages do not use classes but are still object-oriented; for example SELF (Ungar & Smith 1991). In SELF, objects are created by cloning other objects called *prototypes*. This text concentrates on describing class-based, object technology. Here, classes are used merely as conceptual instruments for defining abstractions and for structural organization. It is not assumed that a class-based programming language will be used for implementing the object structures presented in this study.

A class is identified by a *class name*. A common convention is to write a class name with an initial capital letter followed by lower case letters; for example, "Car", "Window" or "Note". In more complex class-names, several words may be concatenated with capital letters used as word separators (e.g., "SportsUtilityVehicle" or "KeySignature"). Other naming practices exist but this one is used exclusively in this study. The same convention is also used in the UML manuals.

Instances are also identified by their names. To distinguish instance-names from class-names, instances are typically designated with an initial lower case letter followed by mixed-case letters or numbers. An object may also include the class name. For example, "myCar", "car1", and "car2" might be used as names for instances of the class "Car".

Classes or objects may relate to each other in various ways. A relationship between two classes A and B means that an instance of class A has a relationship with an instance of class B. There are three basic types of relationships: *association*, *aggregation*, and *inheritance*. *Aggregation* is a relationship where a class (or set of classes) is part of another class, called an *aggregate*. The part-object may be physically stored within the aggregate. Many programming languages also allow aggregation *by reference*, where the part is physically stored elsewhere, but a reference to the part is stored within the aggregate. Languages that use *reference semantics* allow, for example, the moving of an object from one aggregate to another by the moving of its reference, i.e., without moving of the actual data of the object within computer memory. Also, reference semantics makes it possible for an object to be logically part of more than one aggregate.

Inheritance is a relationship between classes wherein a class shares a part of its properties with a similar but more specialized class. The more specialized class is said to *inherit* properties from the more general class. Inheritance is also called a generalization / specialization relationship, depending on the direction of the relationship between the general and specialized class, also called a *superclass* and *subclass*, respectively. A superclass is a "generalization" of its subclasses, and a subclass is a "specialization" of its superclass. *Association* is a generic relationship, other than aggregation or inheritance, between two or more

classes or objects. An association between two classes indicates that the classes are somehow related.

In a class-based, object-oriented system every object is an instance of some class. Yet, not all classes necessarily have instances. A class that is designed not to have instances is called an *abstract class*. Abstract classes may have (and usually do have) subclasses that have instances.

A central term used in this study is the *state* of an object. A state is a combination of an object's attribute values, and it may be either *stable* or *unstable*. An object is in an unstable state if it has not fully completed an operation and is not ready to be used again without causing a possible error. Ideally, upon the completion of every operation, an object should return to a stable state. Encapsulation, the principle that an object's attributes cannot be accessed directly from outside the object, makes it possible to achieve stability in a controlled way. This is an important factor with regard to the reliability of the system.

Objects communicate by executing or *invoking* each other's operations. In many programming languages, an operation is invoked by the sending of a *message* to an object. A message is a request for an object to perform an operation. A *protocol* is an agreement among or arrangement of the types or kinds of messages that are sent between a group of objects. The operations of a class define a protocol that can be used to control the respective instances. Subclasses may "override" the operations of their superclass to implement their own specialized behavior. In this way, different classes may have a different behavior for the same message. This principle is called *polymorphism*.

Runtime (or *run-time*) is a state during which an object system is operational. Runtime begins when the execution of an object-oriented program is started. Generally, instances are created during runtime. Compile time (or compile-time) is a term that describes the phase when programming language code is converted to executable machine code. During compile time, the language syntax is checked, and in some languages, types of objects are checked as well. The number of features checked varies among programming languages. In some programming languages, object types are resolved in runtime rather than during compile time.

In some object-oriented languages, objects have knowledge of their class at runtime. This feature is sometimes called *runtime type identification* or *runtime type information* (RTTI). In other languages, this capability is not provided by the language and thus – if considered necessary – must be implemented to each class by the programmer. The capability of objects to hold information of their class might be useful for other objects as well. For example, an object might use the information of the classes of its parts in order to determine how to treat the classes. Stroustrup (1994: 315-316) has described the advantages and disadvan-

tages of using runtime type identification. According to him, polymorphism offers a more "object-oriented" solution to most problems where the use of runtime type information is considered necessary. Still, and it seems somewhat reluctantly, Stroustrup agreed to add support for RTTI in the C++ programming language (1994: 306-307).

In object-oriented methodology, the terms (object-oriented) analysis and (object) modeling are used often as synonyms. There is, however, a distinction between the two. In general definitions of the term analysis, it means both the breaking down of a thing into smaller parts (for individual study) and their critical examination. Object modeling, in general, means a process of forming an object model. The model might be built intuitively, but some form of analysis is needed to create a model that resembles or simulates a target system.

It can be said that analysis is a higher-level process than modeling is. For example, a model may be analyzed (examined critically). In turn, object-modeling techniques, UML in particular, offer systematic and (unofficially) standardized notation conventions that are widely known in computing science and in the software industry. Therefore, the result of an object-oriented analysis can be conveniently presented as a UML model.

4.2 Object-oriented programming languages

Simula (Dahl 1966) was the first programming language that can be regarded as object-oriented. It was the first to suggest the metaphor that a computer program should be a combination of physical objects and their behavior. Simula was designed for simulation of physical phenomena.

The Smalltalk programming language (Goldberg & Robson 1989) was developed during the 1970's at the Palo Alto Research Center of Xerox. The Smalltalk system consisted of, not only the programming language itself, but also an integrated, network-distributed software development environment. Smalltalk inherently supported the development of programs having an iconic and graphical user-interface. The ideas of the Xerox user interface were used in the design of first the Apple Macintosh system and later the Microsoft Windows environment – although both of them omitted the Smalltalk development environment.

Smalltalk has been regarded generally as inefficient for creating mainstream application programs (Cox & Novobilski 1991: 38-39), but it has played an important role in many research and development projects and in prototyping (e.g., Pope 1991a; Pope 1991b; Krasner 1991). Moreover, the Smalltalk system has influenced the design of many modern object-oriented languages and environments for software development.

Smalltalk is a purely object-oriented language in which all constructs are classes or objects. The syntax of its programming language does not even include control structures. Instead, the latter are implemented as operations of (Boolean) objects created by test operations. Smalltalk includes the concepts of class, encapsulation, and inter-object communication by messages.

C++ (Stroustrup1992), one of the most widely used object-oriented programming languages, is an extension of the C programming language (Kernighan & Ritchie 1977). C++ retains the static typing principles of C while adding support for classes. C++ has undergone several revisions, the design process of which has been aimed at retaining compatibility with C while adding support for Smalltalk-like programming techniques (within the scope of strictly static typing).

Smalltalk and C++ represent two distinct schools in the design of programming-language. Many central concepts of C++ can be traced to the mathematically-oriented FORTRAN programming language. A typical FORTRAN implementation is a compiler that translates a completed program or a program subroutine and yields a file that contains a machine-code language interpretation of the program for a specific computer platform. Like FORTRAN, C++ is statically typed; i.e., the object types are determined at “compile time”, which is when programming language code is translated into machine code.

Smalltalk, on the other hand, is a highly dynamic language. The object types are determined at runtime, i.e., during the time the program is being executed. A typical Smalltalk implementation is an interpreter (although many implementations also include a compiler). In this respect, the roots of Smalltalk can be traced to the linguistically-oriented LISP language (Steele 1990).

Both C++ and Smalltalk have adapted the concept of class. Smalltalk implementations include collections of general-purpose classes for data storage and for input/output and user interface development. A class library has been proposed also for the forthcoming ANSI standard specification of C++.

In prototype-based programming languages, Smalltalk’s dynamic principle has been taken further. The SELF programming language does not include the concept of classes but implements inheritance by making copies (“clones”) of prototype objects at runtime (Ungar & Smith 1991). Also, several “hybrid” programming languages have been developed. They include features from both Smalltalk- and/or SELF-like dynamic languages and static languages. These include, for example, Java, Dylan, NewtonScript, Python and Delphi.

Even the most commonly-used object-oriented programming languages differ in basic capabilities, implementation of these capabilities, and even basic terminology. Still, they have enough common denominators to make possible the development of software engineering methods that are independent of program-

ming language. The central common denominators are the concepts of class and inheritation, as well as the capability to construct hierarchical, "part-of" object structures. Described below are some related software engineering methods.

4.3 Object-oriented software engineering methods

Several formal object-oriented software engineering methods were developed during the 1980's and 1990's. One purpose of these methods was to provide system designers with tools for describing and specifying a software system. Some of these methods were presented as replacements of pre-established processes of software engineering. Some came about as evolutionary steps from earlier methods of software engineering.

A common principle is to divide an object-oriented software development process into three stages: *object-oriented analysis*, *object-oriented design*, and *object-oriented programming*. The role of the stages varies among the individual methods. Especially, the exact distinction between analysis and design is often vague. The term *object-oriented programming* is commonly used to mean the process of realizing an object-based software system by means of a (usually object-oriented) programming language. The term *implementation* is also used as a synonym for programming.

Object-oriented design is the process of making a specification according to which the programming task is executed. Object-oriented analysis is a process that precedes the design stage. The main purpose of the analysis task is to analyze a "problem domain" and describe it as a system of objects. The analysis and design processes might also be called decomposition and composition, or analysis and synthesis, respectively (Pope 1991a).

According to James Rumbaugh *et al.*, "a software engineering methodology is a process for the organized production of software using a collection of predefined techniques and notational conventions" (1991: 144). In object-oriented methodology, various techniques are described for the identification and classification of objects. Use of a graphical notation convention for defining class and object structures is also a typical feature of the methodology.

In the following sections, three early object-oriented software engineering methods are discussed. Each method takes a different view on the importance and role of the three stages. The methods differ also in the use of basic terminology. Further differences can be found in their relationships with earlier (non-object-oriented) software engineering methodology. The methods discussed are the OOA method by Peter Coad and Edward Yourdon, the ROSE method by Grady Booch, and OMT by James Rumbaugh et al.

There are many other object-oriented methods that are not described here (e.g., Slaer & Mellor 1988; Jacobson 1992). According to Booch et al. (1999: xviii), more than 50 methods had been introduced by 1994. The Coad & Yourdon OOA method is discussed here because it is an early method that presents analysis, design, and programming as distinct stages in the software engineering process. The Booch/ROSE method is discussed partly because of the author's references to the philosophical and theoretical background of object-oriented techniques. OMT is discussed partly because it is, along with the Booch/ROSE method, a direct predecessor of the Unified Modeling Language (UML). Let us recall that UML, a notation language rather than a method, is used as the notation technique of this study.

4.3.1 The Coad & Yourdon OOA method

Some of the first object-oriented methods covering analysis, design and programming as distinct stages were presented in a series of books: *Object-Oriented Analysis* (Coad & Yourdon 1991a), *Object-Oriented Design* (Coad & Yourdon 1991b), and *Object-Oriented Programming* (Coad & Nicola 1991). Coad and Yourdon describe Object-Oriented Analysis as a method for finding classes and objects ("Class-&-Objects"), for identifying structures, and for defining attributes of the problem domain. This method is commonly referred to as either "Coad & Yourdon" or *OOA*. The complete, three-stage process is also called *OOADP*.

Coad and Yourdon define the term "Object-Oriented" with the equation (Coad & Yourdon 1991a: 30):

Object-Oriented = Classes and Objects
+ Inheritance
+ Communication with messages

For Coad and Yourdon, all systems that exclude any of the three factors in the equation are not object-oriented. In the equation, Communication with messages is a "means for managing complexity" (ibid.: 12-18, 30), which may be achieved even if the programming language does not include a concept called messages (as the Smalltalk language does). C++, for example, uses function calls for a similar purpose and can thus be regarded as an object-oriented language. Coad and Yourdon present a graphical notation system for describing classes and objects, inheritance structures, "part-of" structures, attributes, services (i.e., operations), and connections (i.e., associations).

Coad and Yourdon define analysis in this way (ibid.: 18-19):

– – the study of a problem domain, leading to a specification of externally observable behavior; a complete, consistent, and feasible statement of what is needed; a coverage of both functional and qualified operational characteristics (e.g., reliability, available, performance).

Further, Coad and Yourdon state that analysis focuses on "*what* the system must do to satisfy the client, not *how* the system must be implemented." In contradiction to their definition, however, Coad and Yourdon do not give a detailed description or any example of what a "complete, consistent, and feasible" specification should be.

Coad and Yourdon present their method as a replacement for, or rival to, the entity-relationship and data flow diagram methods that dominated systematic software engineering (especially database system design) throughout the 1980's. Since they were among the pioneers of object-oriented methodology, Coad and Yourdon also considered it necessary to point out the advantages of object-oriented methodology in order to justify their approach. The whole OOADP method is straightforward and practical-minded. The third book of the series gives detailed descriptions of several exemplary software projects, including their source code listings in both Smalltalk and C++.

Coad and Yourdon present a graphical notation system for defining classes and their relationships. Separate notation for state transition diagrams is also described, but the main focus of the method is on class and object diagrams. Classes are drawn with round-cornered rectangles that enclose the name of the class. Classes and instances are drawn in the same diagram, which distinguishes it from the methods discussed below. Aggregation and inheritance relationships, called structures, are typically drawn with tree-like hierarchies flowing from top to bottom. When the analysis requires a large amount of structures, they are divided into Subjects.

Coad & Yourdon's OOA process consists of five "major activities": identifying classes and objects, identifying structures, identifying subjects, identifying attributes, and identifying services. The persons that carry out the process are called *analysts*.

The class and object identification activity starts with investigation of the problem domain. There, the analyst should study the field of inquiry by using various techniques such as first-hand observation, listening to problem domain experts, checking previous OOA results, checking other systems, and reading the literature that describes the problem domain. The analyst should account for all nouns encountered in the investigation process and weigh them systematically (Coad & Yourdon 1991a: 58-60).

For weighing potential classes and objects, Coad and Yourdon present a practical, two-step process. First, in order to find candidates for classes and

objects, the analyst should look within the target system for potential objects, such as "structures, other systems, devices, things or events remembered, roles played, operational procedures, sites, and organizational units". Second, the candidates are challenged with a list of criteria called "needed remembrance, needed behavior, (usually) multiple attributes, (usually) more than one object in a class, always-applicable attributes, always-applicable services, domain-based requirements, and not merely derived results" (Coad & Yourdon 1991: 60-78).

If the investigation reveals any *structures*, they should be challenged as potential generalization-specialization relationships or as whole-part relationships. References to *other systems* can reveal points of interaction among those systems and call for specific classes and objects. *Roles played* refers to how human beings act within the system. The analyst needs to consider whether specific classes and objects should be used to represent the various roles. If the system should hold any *operational procedures* over time, they may call for specialized classes and objects. *Sites*, which are physical locations involved with the system, are potentially classes and/or objects. *Organizational units* are also potential classes and objects.

Needed remembrance refers to inquiring whether a system needs to remember anything about the object, whether the object can be described, and what its potential attributes are. *Needed behavior* refers to questioning whether an object should have "services" (i.e., operations). There may be objects that have services but not attributes, i.e., behavior without remembrance. *(Usually) multiple attributes* suggests that objects should usually have more than one attribute each. Otherwise, a simple value could be more practical than a full-fledged object. *(Usually) more than one object* in a class suggests that classes that have only one instance should be challenged. In particular, if there is another class with similar attributes and services, a common class or a generalization-specialization structure should be considered. *Always applicable attributes* and *always applicable services* are criteria for questioning whether a set of attributes or services applies to all objects in a class. If the class has attributes or services that are irrelevant, then some objects should be considered as generalization-specialization structures.

Domain-based requirements refers to the importance of separating analysis decisions from design decisions. The analyst should avoid making decisions and assumptions based on implementation requirements or limitations such as computer hardware. The analysis process should concentrate on the problem domain and leave design decisions to software designers or to the software design stage. For example, task and data management are considered as design decisions. The *not merely derived* criterion suggests the exclusion of attributes and objects that

can be easily derived (e.g., calculated) from other attributes or objects. This criterion helps to avoid redundancy, i.e., the unnecessary duplication of data.

The structure-identification activity involves defining generalization-specialization ("Gen-Spec") and whole-part structures for the identified classes and objects. Objects with similar behaviors are indications of potential generalization-specialization structures. There, new classes may be defined for collecting common attributes or services of the previously identified classes

The subject-identification activity focuses on dividing sets of structures into entities called subjects. Identification of attributes concentrates on the definition and organization of both attributes and "instance connections". (Instance connections are called *associations* in many other object-oriented methods, and also in this study.) Identification of services involves definition of a set of services that the classes and objects provide for the system.

While Coad and Yourdon concentrate largely on the use of class and object diagrams, they also present a notation for describing services called a "service chart". A service chart demonstrates the changes in the state of an object when a service is executed. Service charts have symbols for displaying basic elements of computer algorithms such as states, conditions (i.e., if-then clauses), loops, and transitions between these. Coad's & Yourdon's service charts resemble the traditional flow charts commonly used for visualizing algorithms (ibid.: 145).

The OOA process specifies the problem domain. The specification is used as a basis of the design stage (OOD stage). There, the OOA model of the problem domain is expanded with components involved in a particular implementation. These components include human interaction, task management and data management (ibid.: 178-179). The resulting OOD specification is then given to programmers in order to accomplish the programming (i.e., OOP) stage.

4.3.2 The Booch/ROSE method

Grady Booch first presented his software engineering method in the book *Object-oriented Design with Applications*. A revised version was presented in *Object-oriented Analysis and Design with Applications* (Booch 1994). Booch gives a detailed description of object-oriented software development, from both theoretical and practical points-of-view. Booch's method is commonly referred to as either the ROSE method or the Booch method.

The Booch method concentrates on two tasks: (1) finding and classifying objects and their relationships using both graphical diagrams and a formal text description; and (2) defining the dynamic aspects of the object-system by means of state-transition diagrams and additional text descriptions.

Booch defines the "object-oriented analysis" as follows (1994: 39):

> Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

According to Booch, the emphasis in object-oriented analysis is on the "building of real-world models". Object-oriented design, in turn, is defined as:

> – – a method of design encompassing the process of object-oriented decomposition and notation for depicting both logical and physical as well as static and dynamic models of the system under design

The emphasis is on "the proper and effective structuring of a complex system". (Ibid.)

Booch does not give explicit definitions of the basic terms "analysis" and "design". Although the definitions of both terms can be roughly deduced from Booch's text, the distinction between OOA and OOD is not made very clear. Booch himself admits that the boundaries between the two are "fuzzy". He continues, however, by adding the following:

> In analysis, we seek to model the world by *discovering* the classes and objects that form the vocabulary of the problem domain, and in design, we *invent* the abstractions and mechanisms that provide the behavior that this model requires. (Ibid.: 155.)

Booch gives a list of techniques that can be used in object-oriented analysis: "classical approaches", behavior analysis, domain analysis, use-case analysis, CRC (Class/ Responsibilities/Collaborators) cards, informal English description, and structured analysis. Classical approaches are derived from classical categorization. Behavioral analysis focuses on dynamic behavior as the source of classes and objects. Objects with similar behavior are grouped under the same class. This technique is related to conceptual clustering. Domain analysis can be used as an additional method for finding and defining abstractions. There, objects and classes are organized into domains. Domain analysis is often done by examining existing software systems and by consulting domain experts, i.e., persons that are closely familiar with the domain (Booch 1994: 155-158). Booch's object identification and classification techniques are explained in more detail in section 4.7, below.

Use-case analysis, first described by Ivar Jacobson, is a practice, where a system under examination is tested through example situations in use (Booch 1994: 155-161). CRC cards are regular index cards used in building simulations of scenarios within a system. Every class in a scenario is given a CRC card, on which are written the class name, its responsibilities and collaborators (other classes). New responsibilities and collaborators may be added on the CRC card as the development team evaluates the role of the class within the scenario (ibid.: 159).

Informal English descriptions offer an alternative approach to analysis. In this case, the developer will write a description of the problem and underline each noun and verb in the text. Nouns are considered as potential objects and verbs as their potential operations. This approach is similar to the principle proposed by Coad & Yourdon (see section 4.3.1), such that an existing text may be used as the source document.

Structured analysis is a traditional software engineering method that was used widely in the software industry, especially before object-oriented techniques became popular. Although Booch discourages the use of structured analysis, he describes how it could be used as a basis for object-oriented analysis, should pragmatic reasons demand this. Booch, however, warns against using structured analysis instead of object-oriented analysis as the front end to an object-oriented design. In his criticism of structured analysis, for example the use of data-flow diagrams, Booch sees fundamental differences between structured and object-oriented analysis (ibid.: 160-161).

Perhaps more than most authors of object-oriented software engineering textbooks, Booch addresses the historical and philosophical background of object-oriented software engineering. In particular, he points out the heritage of Plato's and Aristotle's techniques of classification and categorization (Booch 1994: 151, 168). Booch does not insist on using any particular method for identifying objects and their properties. Instead, he presents several alternative or complementary methods, some of which are also used in fields other than software engineering.

Booch, however, does describe a systematic high-level method for software design. He emphasizes that all design projects are unique and that problems can result from developmental processes that are either too strict or too ad hoc. Booch presents his "rational" development process as a suggestion that does not have to be followed without question. Booch divides object-oriented software development projects onto two layers: the macro-development process and the micro-development process. The macro-developmental process presents the major activities involved in the task:

1. Establish core requirements (conceptualization)
2. Develop a model of the desired behavior (analysis)
3. Create an architecture (design)
4. Evolve the implementation (evolution)
5. Manage postdelivery evolution (maintenance)

To describe these briefly (Booch 1994: 248-264): In conceptualization, requirements of the system are defined. Analysis produces a model of the problem domain. Design creates a software architecture. Evolution involves translation of

the design documents into program code. Maintenance involves making adjustments for the system, which are implemented in a new evolution.

The micro development process consists of the following four activities

1. Identify the classes and objects at a given level of abstraction.
2. Identify the semantics of these classes and objects
3. Identify the relationships among these classes
4. Specify the interface and then the implementation of these classes and objects

To summarize Booch (1994: 234-248): The micro-process is cyclic. This means that once step 4 is accomplished, the process returns to step 1. This micro-process cycle will continue throughout the macro-process. The micro-process includes implementation – which has begun already, at very early stages of the macro-process, even before the analysis or design stages are completed.

Booch creates his own graphical notation system for describing classes, objects, and their relationships. Unlike in OOA, classes and objects have dedicated diagrams. Booch uses an amoeba-like shape as a symbol for classes and objects. While classes and objects themselves are more complex to draw than are Coad & Yourdon's round-cornered rectangles, Booch’s notation allows more freedom in the graphical placement of the objects. Furthermore, Booch’s notation is richer and offers more detailed expression than does OOA.

Booch also presents several types of diagrams to illustrate the behavior of objects and the higher-level constructs called “modules”. These include state-transition diagrams, interaction diagrams, module diagrams, and process diagrams (ibid.: 199-208, 217-228). Furthermore, Booch describes a system for writing text specifications (ibid.: 196-199). There, elements like classes or operations are assigned a list of topics to be given a value or explanation in text, such as “Name:”, “Description:”, “Attributes:”, “Operations:”, etc. The approach is similar to the “manual page” system of the UNIX operating system (Kernighan & Pike 1984: 308-312). Booch’s emphasis is, however, on object and class diagrams, especially the latter.

4.3.3 The Object Modeling Technique

The Object Modeling Technique, abbreviated as OMT, was developed by James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen (Rumbaugh *et al.* 1991). Its notation is more systematic than that of the two above methods, and it offers object-oriented substitutes for traditional software engineering techniques, e.g., for entity-relationship graphs and dataflow diagrams. By building on the tradition of pre-object-oriented techniques,

OMT takes an opposite direction to the more radical departure presented by Coad and Yourdon.

OMT uses the term *modeling* as the central activity of the engineering process. The OMT process is divided into three stages: analysis, system design, and object design. The analysis stage involves description of the problem, without consideration of the implementation details, such as data structures. The system-design stage concentrates on high-level modularization of the system. It is at this stage where the implementation issues, such as optimization, are considered. Object design focuses on detailed formation/construction of the individual objects. Implementation and testing are also mentioned as parts of a software "life cycle" but are not presented as parts of the OMT method (Rumbaugh *et al.* 1991: 5, 144-147).

The OMT analysis stage involves construction of three types of models: object models, dynamic models, and functional models. Combined, they form an "analysis model". An *object model* describes and defines the classes and objects in the system. *Dynamic models* are similar to the state-transition schemes, as presented by Booch and others. *Functional modeling* describes the flow of data through the system. The object modeling of OMT can be regarded as a modern equivalent of entity-relationship modeling, where functional models resemble traditional data-flow diagrams. Rumbaugh also describes the relationships between OMT and other software engineering methods (ibid.: 266-274).

The OMT analysis process starts by gathering requests from users, developers, and managers. The requests are compiled into a "problem statement". The problem statement is used as a reference document for building the object, dynamic, and functional models.

In the object-modeling process, Rumbaugh advises starting with a written description of the problem and to consider nouns in the text as candidate classes. Then, unnecessary and incorrect classes are discarded according to a given list of criteria. A class is discarded if it belongs to the following categories: redundant classes, irrelevant classes, vague classes, attributes, operations, roles, and implementation constructs. Redundant classes are classes that hold in common the same information with some other class. Irrelevant classes are those that have nothing to do with the problem. Vague classes are those which have ill-defined boundaries. Names that describe attributes or operations of objects are not considered as correct classes. Roles are not considered as good names for classes. Implementation constructs, such as computer hardware objects, data structures or algorithms, should be excluded from the analysis model, although they can prove relevant at the design stages (ibid.: 148-156).

In the task of analysis, the modeling of associations is a three stage process. First, candidate associations are identified; second, irrelevant associations are removed; third, semantics of associations are specified (ibid.: 156-162). A similar procedure is used for identifying attributes. Rumbaugh's analysis procedures resemble those of Coad and Yourdon. In both methods, potential items are picked and then tested against a given set of criteria.

Rumbaugh sees dynamic modeling as insignificant for static systems such as databases, but quite important for interactive systems (ibid.: 169). Thus, dynamic models may be seen as an optional part of OMT. Dynamic modeling involves the modeling of user interaction, tracing events caused by interaction with users or other external devices, building state diagrams for objects involved with interaction, and building event-flows between objects (ibid.: 169-179).

Functional models "show how values are computed, without regard for sequencing, decisions, or object structure" (ibid.: 179). Functional modeling involves construction of data-flow diagrams and function descriptions. A specialized notation is created for data-flow diagrams. Functions may be described with natural language, mathematical equations, or pseudo-code (ibid.: 179-186).

Rumbaugh suggests an iterative work process for the analysis. On each iteration, the analysis model is refined. Also, an iteration may reveal problems in the analysis model that require major restructuring of the model. The aim is to produce a "cleaner, more coherent design" by iterating each modeling stage (ibid.: 186).

Rumbaugh's book devotes much attention to describing and using the OMT graphical notation. Object identification and classification principles are discussed on a pragmatic rather than theoretical level. Rumbaugh also provides practical advice on how to use object-oriented techniques with traditional programming languages such as FORTRAN, C and Ada (ibid.: 340-363).

4.3.4 The Unified Modeling Language

The Unified Modeling Language (UML) was designed by Grady Booch, Ivar Jacobson, and James Rumbaugh. UML is a combination of earlier methods created by its designers: ROSE by Booch, OOSE by Jacobson, and OMT by Rumbaugh et al. Upon its introduction in the late 1990's, UML was quickly accepted by the software industry. It seems to have remained the most widely used design language and formal, object-oriented analytic method.

UML is described as "a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system" (Booch *et al.* 1999: xv). Compared to graphical notation systems of OOA, Booch/ROSE, and OMT, UML includes more features, including several different types of dia-

grams and a wider range of inter-object relationships. UML adopts its basic syntax from OMT, especially in class and object diagrams, but adds more types of relationships. Parts of both Booch's and Jacobson's notations have also been adapted to UML.

All three of the software engineering methods described above included three basic subjects: explanation of the principles of object-oriented software engineering, advice on how a development process is carried out, and description of a graphic notation for classes and objects, their relationships, and their behavior. UML, in turn, is separated both from explanation of basic object-oriented concepts and from development methodology. The authors have, however, published a book titled *The Unified Development Process* (Jacobson et. al. 1999). It is a methodological guidebook in which UML is used as notation language. On the other hand, the documentation of UML itself does not require the use of any particular method for producing UML language documents.

4.4 UML principles and terminology

UML offers nine different types of diagrams divided into two categories: *structural* and *behavioral* (Booch *et al.* 1999: 93-98). They are intended for visualizing static and dynamic aspects of a system, respectively. There are four types of structural diagrams:

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram

Class diagrams are used to visualize classes and their associations, including inheritance and aggregation structures. Object diagrams are used for displaying example situations with a set of objects and links that illustrate various types of associations. Component and deployment diagrams are used to visualize structures of a level higher than class diagrams. There, one or more classes may be presented as a component, and one or more components may form a deployment (ibid.: 98).

UML contains the following behavioral diagrams:

5. Use case diagram
6. Sequence diagram
7. Collaboration diagram
8. Statechart diagram
9. Activity diagram

Use-cases illustrate various activities that the system will perform. Activities demonstrate "what" a system does instead of "how" is does it. Activities are initiated by actors, which may be objects or users. A sequence diagram shows a time-ordered flow of messages between objects. A collaboration diagram also illustrates the flow of messages between objects, but from an alternative perspective. Stage-chart diagrams illustrate the transition of an object between various states when it is carrying out a specific task (ibid.: 233-256, 331-339).

Activity diagrams display flows of activities within the system. There, activities are higher-level entities than the operations of single objects. Instead, activities typically consist of collaboration between a group of objects. Activity diagrams consist primarily of named activities and transitions from one activity to another, possibly via condition nodes of the "if-then-else" type (ibid. 257-273).

UML also allows the user to construct diagrams that mix features from the above-listed diagram types. In UML terminology, *thing* is used as a collective name for every item or unit that can be modeled. E.g., an object, a class, a component, or a property can be called a thing. Below, some central UML elements and expressions are described with examples.

4.5 UML class and object diagrams

UML class diagrams display classes, their properties, and relationships between classes. The principal relationships in class diagrams are discussed below. Object diagrams display instances of classes, and can be regarded as "snapshots" of the object-system (or a part of it) in a chosen situation. Object diagrams display connections between object instances, not between their respective classes. Therefore, inheritance relationships are not displayed. Booch, Jacobson, and Rumbaugh state that object and instance are "largely synonymous". They believe that an association may be also an instance, although it is not an object (Booch et al. 1999: 185). Here, object diagrams are described more briefly than are class diagrams, because they have minor importance as compared with class diagrams in this study.

Other structural diagrams and/or dynamic diagrams in general are not described here. Apart from one sample interaction, presented in Chapter 9, they were not considered necessary for this study.

Central features of UML diagrams are described with simple examples. Each example shows a UML *statement*. In practice, typical UML diagrams contain more than two or three classes or objects. On the other hand, an entire class structure or object system is usually broken into several small diagrams for the sake of clarity, or because of practical limitations imposed by available display area or computer screen resolution.

4.5.1 Class and its properties

In UML diagrams, a class is shown in a rectangular box containing the name of the class and its properties. Properties are separated from the class name with a horizontal line. Attributes and operations are typically separated from each other by a vertical line.

Figure 4-1 shows an example diagram containing a single class. The class is named CMNSymbol. It has four attributes (“origin”, “size”, “dimensions”, and “value”) and two operations (“draw” and “play”). In addition to name, attributes, and operations, UML also allows a fourth category, called responsibilities. However, a class can be displayed by using only the class name.

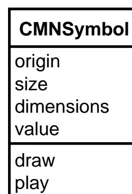


Figure 4-1: Class

4.5.2 Association and aggregation

An association between two classes is shown with a line connecting the classes as in figure 4-2. Associations may have properties called *adornments* such as *name*, *role* or *multiplicity*. In UML terminology, aggregation is described as an adornment of association (Booch *et. al.* 1999: 143). Figure 4-2 states that the class A is associated with the class B. On the instance level, the figure states that an A object (i.e., an instance of the class A) is associated with an instance of class B. Since no adornments are presented in the diagram, an A object may be associated with an undefined amount of B's (instances of B) and vice versa.

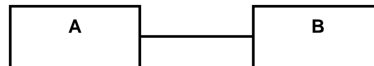


Figure 4-2: Association

Figure 4-3 shows an example of an aggregation. The latter is illustrated by an open diamond shape at the "whole end" of a whole-part relationship. In the example, B is part of A. The roles of A and B are as follows: A is the aggregate, and B is the part. The diagram states that an instance of class A may contain an unspecified amount of instances of class B, while each instance of B may be part of an unspecified amount of instances of A.

The *multiplicity* adornment defines the amount of instances of a specific class allowed at each end of an association. Figure 4-4 shows an aggregation with a multiplicity adornment. The association in the figure can be read as "a 1-to-4 aggregation exists between A and B." Figure 4-4 determines that A has exactly four instances of class B as its parts. Respectively, B is (always, if additional class diagrams are not given) part of one (and only one) instance of class A. Multiplicity may specified by giving an exact value, a list of discrete values (e.g. "0,1,2"), a range (e.g. "0-5"), or an asterisk ("*") denoting an arbitrary amount.

A *role* adornment states that an object refers to its associate object by that name. Figure 4-5 shows an association of a class named Person with itself. A

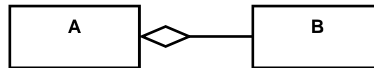


Figure 4-3: Aggregation

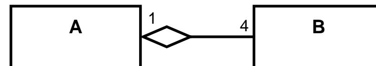


Figure 4-4: Aggregation with a multiplicity adornment

role name is shown at both end of the association. The association states that a Person (i.e. an instance of the class Person) is associated with another Person and refers to that other person as “parent”. The other person, in turn refers to the first Person as “child”.

An association may be given a *name* as an adornment. A name may be used, for example, if there is a need to refer to a particular association among other associations.

A special case of a one-to-many aggregation is called *composition*. In this case, a part may be part of one object only; moreover, the part can not exist without the whole. Figure 4-6 shows an example of a composition between classes A and B. There, an instance of a class B is part of an instance of class A. The existence of instances of B is dependent on the existence of A’s instance(s). If the instance of A is deleted, then the B’s respective instance is deleted with it (Booch *et al.* 1999: 147, 459-460).

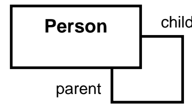


Figure 4-5: Association with role adornments

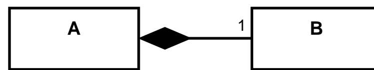


Figure 4-6: Composition

4.5.3 Inheritance

Inheritance is illustrated in UML with an arrow symbol. A large, open arrow head is at the generalization end of the relationship. Figure 4-2 shows an example inheritance structure with a superclass A and two of its subclasses: B and C. Both classes B and C inherit the properties of class A. An instance of class B will have the properties of both A and B. In turn, an instance of C will have the properties of both A and C.

Some programming languages support a feature called *multiple inheritance*. There, a class may have more than one direct superclass. Figure 4-8 shows class C, which inherits classes A and B. A and B could, in turn, have a common superclass. Languages that support multiple inheritance should also include a way to handle potential conflicts that arise when parallel superclasses share the same properties. There the programmer should be able to control which of the conflicting properties are inherited from the individual superclasses.

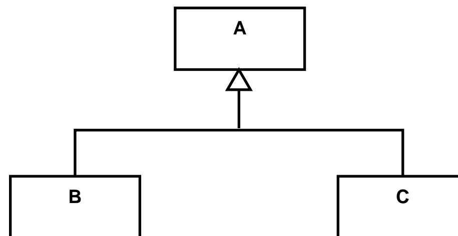


Figure 4-7: Inheritance

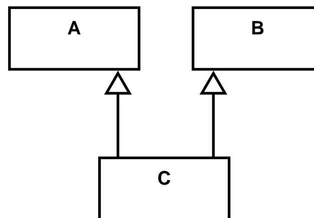


Figure 4-8: Multiple inheritance

4.5.4 Other relationships and features

UML specifies a number of other relationships, including dependency and association class. *Association class* is a class that gives additional properties to an association. An association class is used when an association is considered to be so complex that it requires the use of properties that are logically part of the association and not part of the associated classes (Booch et al. 1999: 147-148).

Dependency is used for defining a connection between two things, where a change in one thing will reflect on or in the other (Booch et al. 1999: 63). This

relationship is shown with a dashed arrow pointing out the direction of the dependency.

4.5.5 Object diagrams

Object diagrams are used for showing instance-level relationships. An object diagram serves as a demonstration tool showing a “snapshot” of a hypothetical or real situation of the object system at runtime.

Object-diagrams may include several instances of the same class. As opposed to class diagrams, association and aggregation relationships are both replaced with *links* (drawn with solid lines) connecting objects. Inheritance relationships are typically not shown, since many programming languages implement inheritance as a compile time construct rather than a runtime feature. Even with prototype-based languages, such as SELF, inheritance structures are not often relevant when the runtime behavior of the system is demonstrated.

In Figure 4-9, an example object diagram is shown. The diagram is an instance-level equivalent of the class diagram of Figure 4-5. The objects are displayed as boxes. Objects can be distinguished from class symbols by the colon symbol that separates the object identifier and the class name, and by the horizontal line under the identifier/name string. The example shows that the object “jill” of class “Person” is a “child” of Person “john” and “john” is a “parent” of “jill”. As noted above, instance names are commonly, although not exclusively, denoted with an initial lower case letter, while class names have an initial upper case letter.

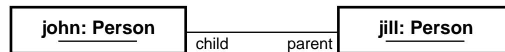


Figure 4-9: Object diagram

4.6 Text representation of object systems

Some object-oriented programming languages and programming environments, such as Java and Objective-C, provide a way of storing a state of the object system in a file. Often, the file formats are binary and meant to be edited within those particular object environments. General-purpose, text-based formats are also available. The technique for storing the state of the object system is called object "serialization", "persistence", or "activation/passivation". Its use and implementation is demonstrated by Coad and Nicola as programming examples in Smalltalk and C++ (Coad & Nicola 1991: 317-385). A similar system for Objective-C is described by Cox (Cox & Novobilsky 1991: 144-146).

Coad and Nicola's solution is syntactically close to XML. In fact, XML is used as the data storage file format of many object-oriented programs. XML documents are text-based and, typically, literally mnemonic. Therefore, XML "tags" can easily be mapped to class names and attributes of an object-oriented system. This allows for XML or a similar text representation to serve as an alternative or a complementary technique for demonstrating the structure of object systems.

A serialized object system typically contains only the object structure and the attribute values. Behavior, i.e. program code, is typically stored in the actual computer program. Although some programming languages support serialization of program code, the inclusion of such code would make the representation dependent on that particular programming language. Thus, a serialized, object-system representation does not substitute class diagrams but may be used as replacements or alternatives for object diagrams.

4.7 Object identification and classification techniques

A common problem in applying any object-oriented software engineering method is the way in which object and classes are identified and defined. The textbooks for engineering methods often approach the problem thorough examples. However, object identification and classification techniques that are systematic and generally applicable have received almost no discussion in the literature on software engineering. Grady Booch has addressed this situation in terms of Western philosophy and human cognition.

Booch describes an object to be either "a tangible and/or visible thing", "something that can be apprehended intellectually", or "something toward which thought or action is directed" (1994: 82-83). Furthermore, "an object has state, exhibits some well-defined behavior, and has unique identity" (ibid.). In a software object, its state is presented by the object's attributes and its behavior is

implemented in the object's operations. Identity is a property that separates an object from all other objects (ibid.: 83-97).

Once the objects have been found, they must be classified. There are several techniques for doing this. Booch lists three principal classification techniques: classical categorization, conceptual clustering, and prototype theory (1994: 150-155). Classical categorization is based on principles first described by Plato in his essay *The Statesman* (in Booch 1994: 168). Aristotle applied this technique in his essay *Categories*, where he described the differences between classes and objects (ibid.). Classical categorization was adopted by later philosophers such as Aquinas, Descartes, and Locke (ibid.: 151). In this technique, objects are considered to be in the same class if they have the same set of properties.

Classical categorization is used widely in modern Western science. However, many natural categories, such as "chair" or "bird", are difficult or impossible to define in terms of any set properties (ibid.: 153). Conceptual clustering and prototype theory are more modern techniques that attempt to address the shortcomings of classical categorization. In conceptual clustering, classes are defined as a set of concepts, and objects are tested against the concepts. A concept is a higher-level entity than is a single property. A concept may have vague boundaries, such that decisions may have to be made on the basis of whether an object belongs more to one class than to another (ibid: 153-154).

Prototype theory is derived from Ludwig Wittgenstein's concept of "family resemblances." In prototype theory, objects are tested against a proto- or stereotypical object. An object is considered to belong to the class represented by the prototype if it bears a sufficient amount of "family resemblance" to the prototype. Booch suggests using all three classification techniques, starting with classical categorization. If the latter doesn't produce a satisfactory result, then the other two techniques are used, in the order that they were introduced above.

4.8 Summary and discussion

All of the object-oriented software engineering methods described above (although UML cannot be called a method) include some kind of graphical notation system. All these methods offer a way to illustrate classes and objects as well as the basic relationships among them, including aggregation, inheritation, and association. Some kind of behavioral notation is also provided, but with much more variation in the amount of features and the emphasis place on them.

OOA, in particular, offers a very restricted way of expressing dynamic aspects of an object system. One reason for Coad and Yourdon to lay such emphasis on static aspects might be that the authors view the analysis stage as basically for constructing static conceptions of the classes and objects of the tar-

get system. Dynamic aspects, such as descriptions of algorithms, belong more (or mostly) to the design and programming stages.

Coad and Yourdon use their notation as a tool for their analysis method. Booch, in turn, describes several alternative methods, especially for the analysis stage. Although Booch introduces his own notation, he permits the borrowing of features from other kinds of notation (1994: 177). Rumbaugh concentrates more on notation and its use instead of methodology. Still, Rumbaugh and his co-authors do provide concise methodological advice for various stages of the development process.

UML presents a complete departure from methodology. UML has enough features to be used in probably most if not all object-oriented development projects that require graphic notation of the object system. It may also be used throughout a development process included in both analysis and design. Moreover, a user may select only those diagrams or expressions that are needed for a particular task. UML could also be used as a direct replacement of Coad and Yourdon's as well as Booch's notation.

The methods differ in how they make the distinction between analysis and design. Coad and Yourdon make huge efforts in separating the two stages, whereas Rumbaugh emphasizes the integration of analysis and design into a single and systematic development process. An extreme example of separating analysis and design is described by Slaer and Mellor, where, unlike with Rumbaugh, only an analysis method is described, and where it is remarked that the analysis might differ from the results of the design process (Slaer & Mellor 1988; Rumbaugh 1991: 273). Rumbaugh, in turn, states that "there is no absolute line between the various design phases" (1991: 187).

The methods also differ among themselves with regard to how, in the analysis, they identify various kinds of relationships between objects. Coad & Yourdon emphasize, in the analysis stage, the importance of finding inheritation and aggregation structures (i.e., "Gen-Spec" and "Part-of"). Rumbaugh, in turn, suggest that generalization and aggregations should be primarily modeled in the design stage. Rumbaugh sees aggregation merely as "an association with extra connotations" (1991: 156).

According to Rumbaugh, there isn't "any such thing as a perfect analysis" (1991: 187). He states, moreover, that

> All abstractions are incomplete and inaccurate. – – The purpose of abstraction is to limit the universe so we can do things. In building models, therefore, you must not search for absolute truth but for adequacy for some purpose. There is no single 'correct' model of a situation, only adequate and inadequate ones. (Rumbaugh 1991: 16.)

Hence, I suggest that decisions on what to focus on in the analysis, and where to draw the line between analysis and design, should be made only after the problem domain has been examined. The exact distinction between analysis and design can be made after both the problem domain and the solution domain have been defined. Analysis should operate on the terminology of the problem domain, while design would deal with terminology of the solution domain.

Chapter 5
Refining the methodology with linear logic

The object-oriented software engineering methods presented in the previous chapter provide advice and examples for identifying classes and their relationships. However, the detailed construction process of an analysis or design model is generally explained in a relatively abstract way. Moreover, the methods place very few restrictions on what kinds of relationships or structures are allowed. Although analysis and design examples are provided, they illustrate relatively simple problems that do not approach the complexity of music notation.

As described by Booch (1994: 148-150) a common method for performing an analysis and design process is by trial and error. There, a primary model is created *ad hoc* and then tested, after which it is either refined or discarded. In dealing in this way with a system as complex as music notation, the amount of models to be tested can easily become impracticably large – especially if the time available for performing the analysis is limited. Here, linear logic is presented as a basis for a disciplinary rule set that can be used to systematize further the object-modeling task. One advantage of the rule set is that it encourages the analyst to be consistent in the usage of class relationships, "part-of" structures in particular.

Linear logic was introduced by Jean-Yves Girard (1987) as an extension or alternative to classical logic. When applied to object-oriented analysis, linear logic leads to the building of coherent aggregation structures. When strict criteria for defining aggregation relationships are at the analyst's disposal, other structures become easier to identify. This can both improve consistency in the model and speed up the analysis task. In this chapter, the principles of linear logic are described. Also discussed are the application of linear logic to object-oriented analysis and its implications for UML.

To judge from the amount of publications, research in linear logic was most intensive in the late 1980's and early 1990's. Since then, interest seems to have declined, but there still is active research in the topic, both on theoretical and practical levels. This chapter concentrates on describing the fundamental concepts of linear logic and its application to object-oriented methodology.

Advanced theoretical or syntactical concepts of linear logic, as well as its application in logical calculation and logic programming, are beyond the scope of this study.

5.1 Linear logic

Linear logic is a logic based on limited resources. Girard demonstrates the fundamental difference between classical and linear logic with the following proposition (1995):

if A and $A \Rightarrow B$, then B

According to the proposition above, in classical logic both A and B would hold, whereas in linear logic A would be "consumed" when the proposition is evaluated. That is to say, A would be rendered false. Girard gives an analogy of A being the sum of $1, B being a pack of cigarettes, and the implication arrow meaning "get". In that case, the sum of $1 would be consumed in getting the pack of cigarettes. Girard compares classical logic's interpretation of this case to a situation where the owner of A would have an unlimited amount of money, which is rarely the case in real life.

Girard does not present linear logic as a replacement for classical logic. Instead, he proposes it as an extension to it: a "logic behind logic" (Girard 1987: 2). Compared to classical logic, linear logic reflects a different notion of time. The truth values of factors are expected to change as a result of linear expressions. As a distinction from classical logic, Girard (1995) has presented a specialized syntax for clauses using linear logic.

Girard compares linear logic to intuitionist logic. Like intuition-based logic, linear logic is constructive. Moreover, it can be translated from intuitionist logic in a consistent way. Girard regards computer science as a central application area for linear logic. He holds, for example, that linear logic can enhance optimal performance by enabling parallel processing on the logical level (Girard 1987: 2-8).

Linear logic reflects physical, real-world objects. Therefore, it can be applied to the study of such natural phenomena as the behavior of physical material or energy. The basic concepts of linear logic yield interesting implications when applied to object modeling. The idea of preserving objects as if they were physical material is easily correlated to computing hardware, but seems to contradict many conventions of software design and programming. In particular, linear logic challenges the "reference semantics" used in many conventional and object-oriented programming languages. When applied to computer science, lin-

ear logic challenges both the established von Neumann-style mathematical metaphor of programming languages, as well as common resource management conventions (e.g., memory management).

5.2 Computational applications

Applications of linear logic in computing science have been studied extensively. This includes the development of linear-logic programming languages, memory management, and management of shared computing resources, such as input/output hardware. Lygon (Winikoff & Harland 1996) is an example of a linear-logic programming language, and it can be regarded as a linear-logic equivalent of Prolog (Sterling & Shapiro 1986). Another linear programming language is LO, which includes object-oriented-like features, such as "organizational" inheritance (Andreoli & Pareshi 1990).

Many computer applications of linear logic focus on logic programming, including development of new logic-programming languages. Some of the applications are theoretical and fully adhere to the principles of linear logic. Others are more liberal and pragmatic because they include some linear functionality (Wadler 1991). Newer theoretical contributions include Girard's Light Linear Logic (1998) and Asperti and Roversi's Intuitionistic Light Affine Logic (2002).

Henry Baker has applied principles of linear logic to the implementation of linear programming languages and techniques. He has, for example, experimented with application of linear logic to the LISP, FORTH and PostScript programming languages. He has also described how linear logic relates to object-oriented programming languages (Baker 1994c). Baker has both described the implementations of purely linear logical language and has proposed linear variables as extensions to existing programming languages (ibid.).

Baker argues that linear logic solves many problems caused by mathematical thinking based on classical logic, which is the kind of mathematical thought that forms the basis of many conventional programming languages, including FORTRAN and C. In particular, Baker illustrates the benefits of linear programming as applied to the management of shared computer resources, such as the central processing unit (CPU) or a video display. He also states that linear logic adapts more naturally to the modeling of physical objects than do the mathematically oriented, conventional programming techniques.

According to Baker, linear variables are "use-once" resources. They are "consumed" by functions that take them as parameters. Thus, variables whose values are needed more than once must be explicitly copied before their evalua-

tion. Baker demonstrates this principle by programming examples in LISP (1994a).

On his view (Baker 1994a: 35-36), linear objects correspond to real-world objects: "Linear objects have 'true' identity." In Baker's "linear style of programming", objects may be easily moved but not easily copied or destroyed. On the other hand, since linear functions consume their arguments, a function must explicitly return the argument if it is intended to be used again.

The central physical resources in computing are memory, processing units, input/output devices, and peripheral devices. Since they all are physical objects, each of them can be used by only one party at a time. Thus, they can be seen as true linear objects. They cannot be copied (without requiring external material or energy). They can, however, be passed from one user to another. Moreover, in Baker's linear style of programming, virtual resources, such as variables, are treated as if they were physical objects. This requires a special discipline for handling the resources, which is ideally provided by the programming language.

According to Baker, linear logic allows only one "path" to an object (1994a: 36). In other words, there can be only one direct reference to a linear object. Further, the object may be accessed by only one other object at a time – the object that holds the reference. A linear object cannot be copied, unless it explicitly provides a method for copying itself. When an object is copied, all its parts are copied with it. (This procedure is called a "deep copy" whereas in a "shallow copy" only references to the parts are copied.) Because there is always only one access path to an object, objects may be part of only one aggregate at a time.

The spatial behavior of a linear programming language may be demonstrated with a simple example. If we assume that **a**, **b** are variables, the statement

c = **a** + **b**

would yield **c** (as a sum of **a** and **b**), while **a** and **b** would be consumed. Another interpretation could be that **a** and **b** are merged, and **c** would indicate the merger of variables. In a computer implementation, the memory allocated for **a** and **b** could be reused by **c**. Thus, memory space requirements could (ideally) remain constant.

5.3 Considerations

Linear logic questions the principle of "reference semantics" used widely in object-oriented programming languages. Reference semantics means that objects are not physically stored as part of each other, even in an aggregation

relationship. Rather they are accessed by a reference, which is typically the memory address of the object.

In many high-level languages, such as Smalltalk and SELF, reference semantics is considered a benefit: an object may be efficiently moved from place to place and referred to by several objects at once. Thus, an object may logically be part of several other objects at once. This violates the main principle of linear programming, which states that there may be only one physical as well as logical instance of an object. Languages that use reference semantics allow multiple logical instances of one physical object.

The high esteem for reference semantics, which is apparent in the software engineering methods presented in the previous chapter, can be seen as a reason for the acceptance of their inability to provide systematic principles for evaluating the resulting object models. They all fail, for example, to strictly and unambiguously distinguish aggregation from association. This is perhaps not surprising, since in commonly-used, object-oriented programming languages both association and aggregation are typically implemented in the same way: by referencing.

Linear logic solves this problem simply and straightforwardly by allowing only one logical instance of an object. This implies that an object may be an immediate part of only one object at a time. To be used as a part of another object, either it first must be removed from its previous owner, or it must be explicitly copied. After copying, there would exist two independent objects. Association, on the other hand, is equivalent to the name of an object – not to the object itself.

Furthermore, when a container object has total ownership of its parts, it can also block outside objects from access to those parts. This yields true data encapsulation. In reference semantics, an object that (even temporarily) allows access to its parts loses total ownership of them. Therefore, there are strong grounds on which to state that linear logic can guarantee true encapsulation, while reference semantics cannot.

Linear logic has been criticized as being computationally inefficient (Baker 1994b), although both Girard and Baker have regarded it as a means of improving efficiency (Girard 1987: 2; Baker 1994a). Assumed inefficiency may be one reason why linear logic hasn't yet gained wide acceptance among programmers or in the development of mainstream programming languages. Another reason may be that programmers are simply not motivated enough to learn a discipline that differs from conventional algorithm design and implementation. A third reason might be that linear logic may prove difficult to understand by persons accustomed to classical logic, one of the cornerstones of digital computing.

Nevertheless, linear logic has an established position among logics and among the theoretical tools of computer science.

5.4 Application in object-oriented analysis

The implications of linear logic on object-oriented analysis and object modeling in general can be divided into two areas: 1) implications on structure, and 2) implications on dynamic behavior. Both implications can be derived from Baker's statement that "there is only one 'path' to an object at a time". This automatically implies that an object can be part of only one other object at a time. On the other hand, since every object must exist somewhere within the object system, every object must be part of some other thing, either an object or a larger construct such as a whole system (if it is something other than a large object) or some undefined construct. In a pure object system all data is stored within objects; hence we can simply assume that all objects within an object system are part of some larger object. We may also assume that the system itself is an object that is part of some larger, undefined "supersystem". This structural implication leads to a clear distinction between aggregation and other types of association.

The main dynamic implication on object modeling is derived from the linear-logic principle that clauses or functions consume their arguments. The effect of linear logic on dynamic modeling depends on the degree of linearity used in the process of developing software. If a purely linear programming language is used in the implementation, requirements imposed by linearity must also be taken into account in dynamic modeling. Because (as suggested by Coad and Yourdon) implementation issues should, however, be ignored at the analysis stage, linear dynamic behavior should be modeled from the design stage onward. This also leads to the question of whether detailed dynamic modeling should be performed at all in an object-oriented analysis. The aggregation structure may, however, be modeled according to linear principles without assumptions being made about the implementation programming language. In this case, it is up to the designers and implementers to decide to what degree linear logic is applied beyond structural issues.

Linear logic provides a systematic means for controlling complexity. It leads to the formation of a coherent aggregation hierarchy. Moreover, it provides a method for systematically conducting and evaluating an analysis. There may be other, even better, methods for systemization of the analysis process. For this study, however, none were readily at hand. Linear logic was adopted here mainly because it restricts the amount of aggregation relationships and forces the building of a simple, "part-of" structure, thus making it possible to handle a

system as complex as music notation. Furthermore, the distinction between aggregation and association is clarified. Without this systematic and strict additional method, all decisions concerning aggregation would have to be argued independently, which would make the analysis process extremely difficult and slow.

The main criticism against linear logic stems from its computational inefficiency. Even so, computing efficiency is not the primary aim of object methodology in general; and since the present study is primarily theoretical, practical issues such as performance are considered secondary here.

5.5 Formal rules for a linear object system

The main principles of a linear object system are defined by the following rules:

1. There may be only one physical instance of an object
2. Every object must have a unique identifier
3. Every object must at all times be part of some other object within the system
4. If an aggregate is destroyed, all its parts are destroyed with it
5. An aggregate controls the access to its parts

As described above, the main implication of linear programming is that there may be only one physical instance of an object. Rule 1 restricts aggregation, such that an object may only be the immediate aggregate of one object only. Aggregations may nevertheless form a hierarchical structure. To become the immediate part of another object, it must first be removed from its present aggregate. As stated by Baker (1994a), to be acquired by the next user, the object must be returned to its original location. The only alternative to the removal of an object would be to construct an identical but independent copy of the object.

The second rule states that every object has a unique identifier by which its owner refers to it. From here on, this identifier shall be called an *object name*. An object name does not imply the physical existence of an object. In linear object terminology, a name is *unbound* to the object. Baker (1994a), in contrast, uses the word *name* exclusively to mean a *bound name*, i.e., the actual object – which can be referenced only once (Baker 1994a).

Derived from Rule 1, Rule 3 states that every object must have an *owner*. It thus follows that a system will form a single aggregation hierarchy. One object shall be the root of the aggregation hierarchy. As described above, even the root object may be conceived as part of a larger, but undefined system.

According to Rule 4, aggregation controls the existence of the object. If any parts of an aggregate are to be preserved, they must be explicitly removed from

the aggregate and placed into another object prior to the destruction of the aggregate. As a result, there will be no objects without an owner (Rule 3).

Rule 5 means that an aggregate may either block or permit access to its parts. Either all references to the aggregate's parts must be passed through the aggregate, or the parts must be removed from the aggregate and inserted as parts of the accessing object. The object's owner has control over its contents, but can give access to those contents to only one object at a time. During the access operation, however, the aggregate shall hand ownership over to the accessing object. The original owner can access the object only after it has been returned by the accessing object.

Within the scope of this study, *inheritance* is not affected by linear logic. Conventional, class-based inheritation is not in contradiction with Girard's or Baker's principles. Common inheritation structures – both single and multiple inheritance – can be used in a linear object system. For example, in a static type of class-based object system, e.g., in C++, inheritance relationships are constructed at the programming stage – prior to the execution of the program and the creation of the object instances. This behavior is also visible in the notation conventions of UML object diagrams. UML gives no predefined means of showing inheritation hierarchies in object diagrams. Inheritation is shown in class diagrams only. In dynamic programming languages, where inheritance structures may be changed during run-time, linearity can impose restrictions on the use of some of the language's capabilities. These considerations are, however, ruled out here.

The implications of linear logic on dynamic modeling can be summarized as follows:

1. An aggregate cannot allow direct access to its parts.
2. Upon a request for data or resources, an aggregate will either hand over total ownership to the data/resource or will produce a complete copy of the requested data.
3. Operations consume their arguments.
4. An operation must explicitly return any argument that needs to be preserved.

5.6 Implications for UML notation

UML provides sufficiently explicit notation to model linear-logic-based class relationships. In particular, UML's composition feature is equivalent to a linear aggregation relationship. Regular UML aggregations would, however, either not

be allowed or should be interpreted as compositions. Other types of associations would be allowed, provided that the rules listed above are followed.

In linear class diagrams, association classes must also follow the formal rules just presented in section 5.5. This means that an instance of an association class must be part of another object in the system. Furthermore, the instance of the association class may only exist when the respective association exists. A suitable aggregate for the instance of the association class would be one (and only one) of the objects associated with each other. The choice of this aggregation relationship can, however, be left to the implementation stage. Therefore, UML notation for association classes is permissible.

In structural diagrams, linear logic, at the least, affects state diagrams. Because operations consume their arguments, an object's state may change considerably during operations. The explicit copying requirements described above should also be presented in the UML state diagrams. In general, however, neither UML syntax nor UML semantics would be affected.

5.7 Toward a systematic analysis process

One interesting side-effect of linear logic is that it opens a door to the systemization of object-oriented analysis. Since there can be only one aggregation hierarchy with a single root object, this root object can be used as the starting point of an incremental and iterative decomposition process. There, the target system (i.e., the root object) is divided into parts, each of which is divided into smaller parts. This process continues until a desired level of abstraction for the system is reached or until primitive (i.e., indivisible) attributes are encountered.

After the aggregation hierarchy has been constructed, objects can be classified. The search for similarities in these classes can lead to the definition of superclasses. Superclasses themselves can be investigated for similarities that might justify the definition of higher-level superclasses. Finally, through use of the inheritation structure, the aggregation structure may be reduced and simplified.

If the initial aggregation structure is detailed at the level of individual attributes and operations, classical categorization may be sufficient for constructing the inheritation hierarchy. Otherwise, conceptual clustering or prototype theory could be used as well. We shall call this method *Top-down Aggregation, Bottom-up Inheritation* (TABI).

Steven Travis Pope has discussed the difficulties of systemizing object-oriented analysis and design processes. According to Pope, the systematic, structural, top-down decomposition techniques used in many non-object-oriented engineering methods cannot be applied in an object-oriented environment

because "there is no top" (Pope 1991a: 34). By this Pope means that object-oriented programming languages generally allow multiple, separate aggregation structures to exist in the same object system. In a linear aggregation structure, the root object forms a "top". Therefore, a top-down decomposition process can be conducted.

TABI could be used either as the main principle for organizing the analysis process, or it might serve as a means of testing an existing structural model at some phase of an iterative modeling process. As pointed out by Booch (1994: 229-234), however, formal methodology can have negative consequences when applied too strictly. Therefore, TABI should be viewed as an option rather than a mandatory process. Furthermore, when a finished model is examined, the question of how (through what process) the model was constructed should be irrelevant. We thus leave further discussion of TABI for future study.

Chapter 6
Analysis principles for music notation

Here, the methodology presented in the previous two chapters is applied to analysis of music notation. Basic principles of analysis are discussed, and the goals and scope of the analytic model presented in the next chapter are described. We revisit some current models of music representation, discussed earlier, but this time from an object-oriented point of view.

The first of the analysis principles discussed in this chapter concerns the choice of information to be used as the primary target of the analysis (i.e., choosing the orientation of the object representation). The second principle involves how to categorize the signifiers of an object-oriented representation. The third principle deals with the scope of the analysis. Also in this chapter, general requirements for our analysis model are presented, and a set of preliminary examples are given so as to clarify our method further.

6.1 Existing systems

Questions concerning the choice of orientation for analyzing music notation can be approached through examination of existing object-oriented or object-like representations. As will be shown below, the structural architecture of the representation reflects the primary orientation or motivation of each one.

First we describe two systems that are not designed specifically for musical or notational purposes: Sound Processing Kit and Adobe Illustrator. They both have relatively simple representational structures. Sound Processing Kit is an object-oriented software toolkit for audio signal processing; Adobe Illustrator is a vector-based, general-purpose drawing program. In addition to these two systems, we describe a group of musical representations and briefly explain the ways each one manifests various object-oriented features; aggregation, inheritation, and associations, in particular.

6.1.1 Sound Processing Kit

Sound Processing Kit (Lassfolk 1995, 1999) is an object-oriented software system for audio signal processing. Sound Processing Kit (SPKit) was designed for use by programmers for implementing audio-signal processing programs, and by educators for teaching digital-signal processing techniques. SPKit contains a class-based object system that provides objects for individual tasks of signal processing. These tasks range from simple arithmetic operations, to signal routing operations, to complex audio effects such as reverberation.

SPKit utilizes both inheritation and aggregation in order to conserve on programming efforts. The SPKit inheritation structure is built on a single superclass, the SPKitProcessor. It provides basic attributes and operations for connecting signal-processing objects and for transmitting audio signals between objects. All other classes in SPKit are subclasses of SPKitProcessor. Inheritation is used in other classes as well. For example, a set of Butterworth filters (Dodge & Jerse 1985: 189-193) is built on a superclass named SPKitButterworthFilter, which implements the common properties of second-order, Infinite Impulse Response (IIR) filters (see Oppenheim & Schaefer 1975: 195-269). The four subclasses are specializations of the four basic Butterworth filter-variants: low-pass, high-pass, band-pass, and band-reject.

Aggregation is used for constructing complex audio effects from relatively more primitive objects. For example, the class SPKitSchroederReverb implements a classic modular reverberation algorithm invented by Manfred Schroeder (Dodge & Jerse 1985: 229-237). Schroeder designed several artificial digital reverberators based on two modules: a comb filter and an allpass network (a.k.a., allpass filter). Both of them consist of a delay line and a feedback loop. This results in a “circular delay”, in which a signal is continuously recycled and produces a bouncing, echoing sound. Combining several of these units creates a diffused reverb effect, in which individual echoes are difficult to hear. A signal-flow diagram of one type of Schroeder reverberator is shown in Figure 6-1. The

figure displays a digital audio signal flowing from left to right through four parallel comb filters and two, serially-connected, allpass networks.

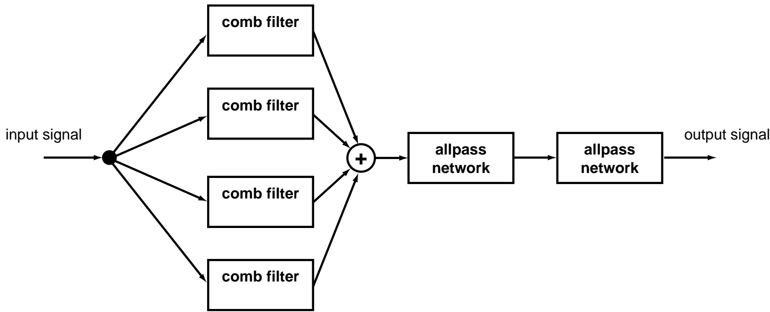


Figure 6-1: Signal flow diagram of a Schroeder reverberator

A UML class diagram of the equivalent SPKit Schroeder reverberator is shown in Figure 6-2. There, the class SPKitSchroederReverb represents the complete reverberator. SPKitSchroederReverb contains four SPKitComb objects and two SPKitAllpassNetwork objects. SPKitComb and SPKitAllpassNetwork contain one SPKitFBDelay object each. SPKitFBDelay is a class that defines a delay line, which can be included in a signal feedback loop. SPKitSchroederReverb has been implemented by Janne Halmkrona.

One advantage of SPKit is that the sound-processing modules initialize themselves automatically when connected to other SPKit objects. During the initialization process, the objects negotiate with each other to determine basic signal-processing parameters, such as signal-sampling rate or channel count, and to identify which objects they are connected to. Because new classes can inherit most of this behavior from existing SPKit classes, the system is easy to extend.

SPKit was not intended specifically for end-users, but for programmers and for studying the implementation of signal-processing algorithms. Therefore, its advantages might be recognized mostly by people with experience in computer programming. SPKit was a successful experiment in object-oriented design. In

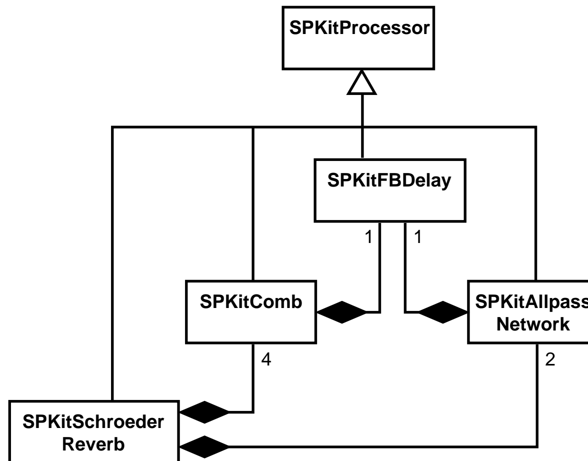


Figure 6-2: UML class diagram of the SPKit Schroeder reverberator

educational use, SPKit provides a concrete tool for teaching the vocabulary of object-oriented design and the application of signal processing theory.

SPKit is an example of a system in which representation and processing algorithms are tightly connected to each other. The representation, i.e., the SPKit class system, is based on the assumption that objects are connected with each other in a specific way and order. Furthermore, it is assumed that audio-signal processing parameters and the audio signal itself are transmitted from object to object by using predefined transmission protocols. No interference is needed from outside objects once a sound processing task has been started.

SPKit was a successful approach to signal processing mainly because a PCM audio signal is itself a rather simple representation – especially as compared with common music notation or its computer representations. A similar object system would not work equally well in a music notation system. For example, optimal spacing would be very difficult or even impossible to achieve by notation symbols negotiating and adjusting their positions by themselves, without the supervision of a higher-level object – or a human being. Moreover, if the representation of a music notation system were tied to the use of some predefined spacing-algorithm, new results in this difficult field of research would be hard or impossible to apply.

6.1.2 Adobe Illustrator

Illustrator is a drawing program developed and marketed by Adobe Systems, Inc. Illustrator is a vector-based program; i.e., it uses vectors as its fundamental graphical element. In addition to vectors, Illustrator supports text fonts (which, in fact, are also comprised of vectors) and pixel-based graphics that can be imported from other programs.

Illustrator is heavily based on the PostScript graphics language, also designed by Adobe. Like PostScript, Illustrator organizes vectors into “paths”. A path is a combination of two-dimensional graphical coordinates and commands that dictate what kind of graphical shape will be used to connect a group of coordinates. Among the available shapes are straight lines and Bezier curves. The path may be “struck” to form a line or “filled” to form a solid area with a specified color.

Illustrator also allows the formation of groups of any kind of graphical objects it supports. Groups can also hold other groups so as to form hierarchical aggregation structures. Virtually any number of objects within a single document may be grouped. Conversely, the objects can be “ungrouped” for individual manipulation. Grouping allows the manipulation of several objects as a single unit. Hence, with a single command a user can apply a graphics-processing function to all objects in a group; such functions include moving the object to another position, rotation, changing the object’s size or color, and so on.

In the data representation of Illustrator, every object – such as a path or a group – is referred to as a certain “Art”. A group object, for instance, is named GroupArt, and a path object is called PathArt. Other object types available are CompoundPathArt, TextArt, TextPathArt, TextRunArt, and PlacedArt (Abobe 1993). Compound paths are combinations of paths that can be used to create graphic objects that contain “holes” (i.e. blank areas). The various text-related objects are used to display text which is comprised of predefined characters from a specified font. PlacedArt is an object for pixel graphics imported from other programs.

Illustrator serves as a good example of a representation that is both simple and flexible. Moreover, Illustrator’s representation has been proven to work in professional illustration tasks. Yet, even though Illustrator’s representation and graphic capabilities are sufficient to create music notation, the program lacks any kind of musical or music-engraving intelligence. Every object created must be created and placed manually, which makes music engraving work difficult. Still, Illustrator and similar applications can be used to modify the graphic output of an external music notation program. Illustrator’s representation can also

be used as a model by which to design the low-level graphic kernel of a music notation program.

Although Illustrator's representation is not genuinely object-oriented, it includes a simple and shallow inheritation structure. All objects can be considered special instances of a generic Art object. With GroupArt objects, the user is allowed to construct hierarchical aggregation structures of practically unlimited depth. Generic associations, on the other hand, are not available.

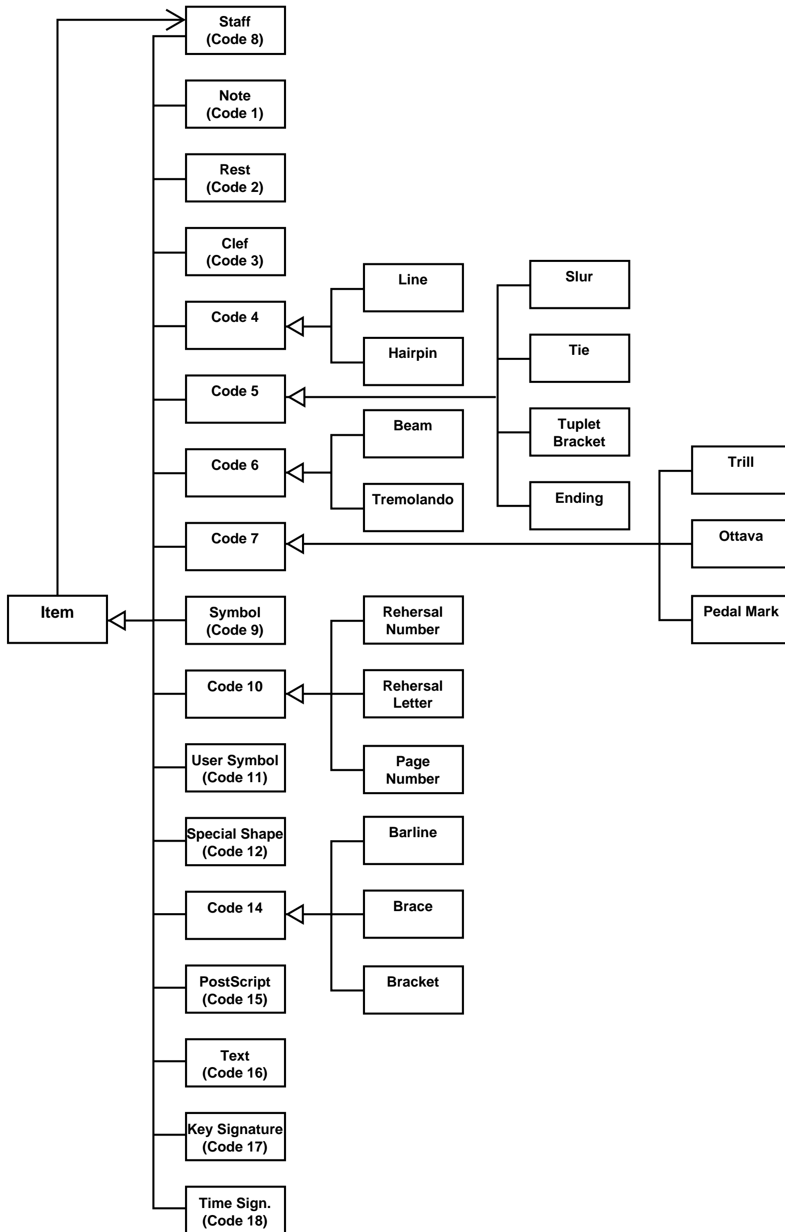
6.1.3 SCORE

Although the SCORE parameter list, described in Chapter 3.8, is not a genuinely object-oriented representation, it can be examined from an object-oriented perspective. For example, SCORE includes a classification scheme for organizing musical symbols. Figure 6-3 shows an UML class diagram of SCORE's data representation.

SCORE's equivalent of a class is an *Item*. Each Item has a code number that determines the type of the Item. SCORE version 3 has 17 code numbers, which represent different types of symbols. The class diagram in Figure 6-3 shows an inheritance structure with a single superclass, Item, on the left. In the UML diagram, subclasses are named either by the type of symbol they represent or by their code number. The latter naming convention applies to Items that represent two or more different kinds of symbol. If two or more different symbols share the same code number, they are modeled in the diagram as subclasses of that code number.

Every Item is associated with a Staff, but not vice versa. This uni-directional relationship is shown with an arrow head pointing out the direction of the association. This feature is called "association navigation" in UML terminology (Booch *et. al.* 1999: 143-144). Each Staff has a unique numerical identifier. Staves within a single system are typically numbered from the bottom-most to the top-most staff, from 1 to the maximum of 32, respectively. Other Items have a Staff number attribute that indicates which Staff the Item is attached to. The Staff, however, does not know which or how many Items are attached to it. Moreover, SCORE allows a Staff to be removed without the attributes of the Items attached to it being affected. Thus, SCORE allows Items to exist that are attached to a nonexistent Staff.

Not shown in the UML diagram, each SCORE Item has a horizontal and a vertical coordinate. The coordinates determine the Item's position on a two-dimensional coordinate system. For all Items except Staves, the coordinates determine the Item's position on a Staff. For Staves, the coordinates determine


Item
Staff
(Code 8)
Note
(Code 1)
Rest
(Code 2)
Clef
(Code 3)
Code 4
Code 5
Code 6
Code 7
Symbol
(Code 9)
Code 10
User Symbol
(Code 11)
Special Shape
(Code 12)
Code 14
PostScript
(Code 15)
Text
(Code 16)
Key Signature
(Code 17)
Time Sign.
(Code 18)
Line
Hairpin
Beam
Tremolando
Rehersal
Number
Rehersal
Letter
Page
Number
Barline
Brace
Bracket
Slur
Tie
Tuplet
Bracket
Ending
Trill
Ottava
Pedal Mark

the position on a page or within a system. Also, all Items have a size attribute, as well as additional attributes that are specific to the type of Item.

Many notation symbols closely associated with notes are implemented in SCORE as parameters of the Note Item. These symbols include accidentals, articulations (e.g., staccato, tenuto, sforzando, and the like), augmentation dots, stems (including the stem's direction, origin, length, and thickness), and ledger lines. This can be regarded a logical "part-of" structure although the parts are implemented as parameters or groups of parameters rather than as distinct Items.

One deficiency in the data structure of SCORE is the limited availability of associations that bind related symbols together. For example, Beams are associated with a Staff, but they have no direct or explicit association with the note stems to which they are visually connected. For example, if a Note is moved during a manual editing process, the Beams are left hanging in their original position. If the note is moved far from the Beam, the Beam has to be manually moved or resized; then, with a dedicated editing command, it must be graphically connected to the nearest stem. Furthermore, stem lengths may also have to be adjusted through the use of another editing command. SCORE associates stems with Beam Items implicitly by interpreting their coordinates (SCORE 1992b: 85-87).

It can be argued that SCORE would benefit from the use of explicit associations and from other structural features provided by object-oriented programming languages. For instance, the FORTRAN programming language could, in principle, be used to implement associations or association-like referencing to objects. This has been demonstrated by Rumbaugh *et al.* (1991: 340-365). The main reason for lack of associations within SCORE, however, probably lies in the design of its data structures – not in the implementation programming language itself.

SCORE uses a simple form of inheritance by reserving the first few parameters of each Item for common attributes. Further, some Items have "subclasses" implemented by the grouping of similar symbols under a common Item. In this respect, SCORE approaches an object-oriented architecture. SCORE's simple inheritance structure is mainly founded on graphic similarities between objects. For example, Beam and Tremolando (also a beam) share a common Item number, as do Slur and Tie. Also in this respect, the SCORE parameter list can be regarded as a graphically-oriented representation.

6.1.4 Tilia and MusicXML

Tilia and MusicXML utilize more modern representational techniques than those used by SCORE. In particular, they support associations (called, e.g.,

"links" or "references") between objects belonging to different aggregation structures. Also, they show a tendency toward logically-oriented representation, in contrast to SCORE's graphically-oriented approach.

In Tilia (Haken & Blostein 1994), a *voice* is the basic aggregational unit. Each voice is stored as a separate list of nodes. Note and rest nodes, for example, are parts of a voice. A voice may visually move from one staff to another. On the other hand, more than one voice may be printed on the same staff. A "PRINT" node in a voice determines the staff on which subsequent nodes are printed (e.g., nodes such as a note or rest symbols). The aggregation hierarchy of a Tilia document is limited to two levels: voice and node. The relationship between a staff and the nodes within a voice can be regarded as an association.

Tilia allows the creation of links (i.e., associations) between nodes in different voices. This feature is used during the automatic formatting process in the Lime program. Tilia includes a primitive form of inheritation, somewhat similar to SCORE: the first four fields have a common meaning in all node types. Other fields are specific to each node type.

MusicXML (Good 2001) allows two types of alternate aggregation hierarchies, called "part-wise" and "time-wise". In time-wise representation, "note" elements are parts of a "part" element, which is, in turn, part of a "measure" element. In part-wise documents, the note elements are part of a measure element, which is part of a part element. The top-level aggregation element is either "score-partwise" or "score-timewise", referring to either one of the aggregation hierarchy types.

A MusicXML note stores its properties as either XML attributes or as subelements, such as "pitch", "duration" (a numeric value in beats), "type" (a text string representing the durational value of a note, e.g., "eight" or "16th"), "stem", "beam", and "lyric". Besides note elements, measures or parts may contain elements such as "key" (key signature), "time" (time signature), and "clef".

MusicXML allows one to define identifiers for elements such as parts, measures, and beams. This in turn allows the building of associations between objects. On the other hand, MusicXML does not offer any form of inheritance.

6.1.5 Object-oriented representations of music notation

Despite the dominant position of object-oriented methodology in software engineering at the beginning of the twenty-first century, relatively few purely object-oriented representations or analyses of music notation have been published. Among these are the MOODS project, Glendon Diener's TTree/glyph structure, Andrew Eales' Music Notation Toolkit, and Mika Kuuskankare's and Mikael Laurson's ENP system. While being purely object-oriented, the MODE system

(see Chapter 3.13) is a polymorphic representation, wherein music notation is only one form among many. Moreover, MODE is not designed for preparing musical scores for printing or publishing purposes.

The MOODS system aims to achieve the cooperative manipulation of music notation, and includes an XML representation for transmission of musical manuscripts (Bellini & Nesi 2001, Bellini *et al.* 2002). An overview of the system is presented on the MOODS project WWW home page (MOODS 2004).

In his Ph.D. dissertation, Glendon Diener presented a graphically-oriented representation of music notation based on a hierarchical aggregation structure. Diener used the representation to implement his Nutation notation program for NeXT workstations. Diener's representation is centered around a concept called "glyph". A glyph is a graphic symbol or a combination of several symbols treated as one. A glyph can be as simple as a single note or as complex as a staff and all the notes and other symbols on it (Diener 1990: 17-19).

Glyphs are stored in a hierarchical data structure called a TTree, which resembles the "list" data structure of the Lisp programming language (Steele 1990). A TTree may grow in two directions, such that each leaf may be a root of another TTree (Diener 1990: 32-35). Diener also explains that glyphs can be implemented as objects, which would benefit from general, object-oriented features such as inheritation (ibid.: 22-25). Furthermore, the Nutation program includes polymorphic features for displaying musical data in various forms that differ from and go beyond common music notation. These forms include a piano-roll notation and Okinawan music notation (ibid.: 83-85, 99-105). In both these forms of notation, the same basic glyph/TTree structure was used. Diener mentions that Nutation also includes an object-oriented class system (including an inheritation structure), but does not describe it in detail in his dissertation (Ibid.: 22-25).

Diener's dissertation can be regarded as a case study in using a particular data structure for storing and manipulating notation symbols. His glyph-based representation is consistently graphically-oriented, and the TTree system forms a hierarchical and user-extendable aggregation structure. Diener's text puts less emphasis on inheritation and association relationships.

Andrew Eales (2000) published an UML description of his Music Notation Toolkit, a software system written in C++ for the Microsoft Windows environment. Eales presented an UML diagram of the "problem domain" and a model of the "problem domain with extensions". The first of these could be called an *analysis model*, and the second a *design model*, respectively. Both models include a deep aggregation hierarchy, with classes called "Score", "Page", "System", and "Staff" forming the highest four levels. At the bottom of the aggregation hierarchy are the classes "Chord", "Note", and "Rest". In the problem-

domain model they are parts of a "Voice". Voices are part of a "Bar", which is, in turn, part of a Staff. Few inheritation relationships are presented in the model of the problem domain. In the extended model, the class "MusicSymbol" is presented as a common superclass for most objects. "LineSymbol", "MusicEvent", and "AbstractBar" are presented as additional, intermediate superclasses.

Eales' UML diagrams show his object system to be a combination of graphic, logical, and performance objects. Some comparisons between our own model and Eales' are presented in Chapter 9.

The Expressive Notation Package (ENP) by Mika Kuuskankare and Mikael Laurson (2003) is an object-oriented music notation program intended for compositional and music-analytic use. ENP is written in Common LISP and in the OpenGL graphics programming language. ENP includes an object-based representation system with a hierarchical aggregation and inheritation structure. ENP also supports persistent object storage as described in Chapter 4.6.

6.2 Basic criteria for a new object representation

One way to improve consistency in a music notation representation is to give priority to either logical, graphical, or performance information. In principle, this should make the representation consistently logically-oriented, graphically-oriented, or performance-oriented. In an object-oriented representation, priority can be achieved by modeling only one of these types as object and modeling the other types as properties or relationships of those objects. Giving one of the basic types of information absolute priority over the other types can also help to make that representation explicitly the best-suited or most "idiomatic" one for a chosen application. One characteristic of an explicit, "best fit" representation is that one-to-one mapping exists between signifiers and signifieds. Here, the central question is, What signifieds are the representation's signifiers intended to represent?

For notational purposes, performance information is the most difficult criterion on which to ground one of the three basic types, because straightforward, one-to-one mapping is, in many situations, hard or impossible to achieve between notation symbols and performance signifiers, such as MIDI events – let alone samples of audio signals in a digitally recorded, acoustical performance. If a performance-oriented representation is intended to support representation of music notation, then signifiers not directly tied to performance would very likely be needed in addition to or instead of the notation symbols. This would result in a hybrid performance/notation representation, where consistency and explicitness would be hard to achieve. The proposed notation extensions to MIDI, mentioned above, exemplify such a situation.

Many existing computer representations are at least partly logically-oriented. In both NIFF and MusicXML, logical information is given priority in data transmission between programs. Preservation of graphic layout is considered less important and can be generated automatically. Yet, construction or reconstruction of many (if not all) forms of logical information from a graphically-oriented representation (e.g., from the SCORE parameter list) is a relatively simple task as compared, for instance, to the difficulties involved with automatic spacing or page layout (as described in Chapter 2.7). For instance, the note name and octave range may be quite easily computed given the type of note symbol, its position on the staff, and its environment. Therefore, the preservation and explicit expression of detailed graphics should be a central concern, especially if the representation is intended to preserve the full expressiveness of music notation itself. Logical information can (and perhaps should), however, be given priority, especially if notational expressiveness is considered secondary to the preservation of some form of "logic" beyond notation symbols. In fact, the names of both MusicXML and SMDL (Standard "Music" Description Language) reflect this kind of approach: their names suggest that they are representations of "music" rather than (only or primarily) "music notation". This leads to a rather difficult, but nevertheless interesting philosophical question: "Beyond music notation, what is music?"

If a logically-oriented approach is to be chosen as the basis for an analysis model, a central problem would be how to find and define the logical signifiers and, especially, what their respective signifieds would be. If the logical type of information in music notation somehow reflects human logic or cognition, then some kind of conceptual or cognitive analysis should be performed in order to find objective criteria for defining these logical signifieds and to discover suitable principles for their categorization. The fact that there already are logically-oriented representations cannot be used as the only evidence of the existence of such signifieds, or if they do exist, what they would be. The principles of how and from where the signifieds in the existing logically-oriented representations are derived, could, however, form a relevant subject for study.

In analyzing an assumed logic beyond music notation, the analyst should consider, for example, what a "logical note" is and how it relates to or differs from a visual note. Even if such logical signifieds could be found, it is hard to imagine a logically-oriented representation completely free of purely graphical signifieds – especially if the aim of the representation is to preserve, in detail, all information of a printed score. Very likely, such a logically-oriented representation would eventually result in a hybrid, logical/graphic design.

In a graphically-oriented representation the signifieds are visual symbols. Thus, rigorous one-to-one mapping between signifiers and signifieds may be

easily achieved. This is a fundamentally different situation from a logically-oriented representation, where the signified can be regarded as something that the signifiers (i.e., visual symbols) represent – rather than what they "are". Therefore, a graphically-oriented computer representation can be regarded as more iconic, and a logically-oriented representation as more symbolic, in their relations to music notation.

It is obvious that logically- and graphically-oriented approaches each have their own advantages and disadvantages. A logically-oriented representation may be optimal and explicit for many notation-related uses in music production or for algorithmic music analysis. MusicXML, for example, is intended specifically for these areas of application (Good 2001). For other areas, a graphically-oriented representation is likely to be more optimal. These last areas include music engraving (or other tasks involved with the processing and publication of printed music), notation of Western art music (especially that composed after World War II), manual music analysis, or other uses where the main concern is precise and detailed graphic expression. Further, as a more iconic representation, a graphically-oriented system stands on a firmer conceptual basis than does a logically-oriented one, because there is undeniable visual evidence of its signifieds.

A central problem in graphically-oriented representation is how to relate the visual signifiers with the other types of information. In SCORE, most logical and performance data are discarded completely. Adobe Illustrator provides an even simpler representation, but operates on a non-musical semantic level. In the present study, an object-oriented approach is proposed as a solution, such that logical information is represented as relationships between graphic symbols.

The next chapter presents a new, object-oriented analysis model, which is based on the following two assumptions:

1. Music notation represents music by interrelated graphical symbols.
2. Music notation does not exist without the presence of at least one identifiable graphic symbol.

It thus follows that a software simulation of music notation can be realized by a system of objects, all of which have a visual appearance. Furthermore, a document of computer-simulated music notation having no visual objects should contain no objects at all. Therefore, all logical data should be stored as attributes of visual objects or represented as relationships between visual objects. The result is a consistently graphics-oriented representation, where the need and role of logical information is acknowledged. In this representation, one-to-one mapping of signifiers and signifieds is easy to achieve. The analysis/interpretation

process used in forming the representation is relatively simple and straightforward.

In a purely graphics-oriented representation, purely logical constructs such as “voice” and “part” should be ruled out as organizational objects, because they have no explicit visual appearance. Although a voice can be read and “extracted” from a polyphonic texture, “voice” itself has no unique and distinctive visual shape. The same applies also to a part, although a part may sometimes have an explicit symbol that indicates its existence, as when an instrument's name is printed to the left of a staff. In general, any element that can be regarded as purely logical should be questioned and, if found to be so, should be ruled out as a potential object. Hence such an element also cannot serve as a container for other objects. Logical aspects can, however, be modeled as attributes of or associations between purely graphic objects. For example, if a note is regarded as a graphic element, and thus a valid object, then a voice (or part) can be modeled as an association between notes.

Although the performance-oriented approach was ruled out as the basis of the analysis model, the relationship between performance information and graphic objects must still be considered. Firstly, as acknowledged in the SMDL design principles (Sloan 1997: 470-471), there can be several different performances of the same score. Secondly, several scores of the same piece of music (e.g., which differ from each other in graphic layout) can yield a similar performance. A performance of a musical score can be considered as a unique (and often non-reconstructible) interpretation process. Once a performance is created, it can (and maybe even should) be modeled as a separate object-system of a completely different kind, such as a PCM audio signal or MIDI data. Moreover, a printed score can provide sufficient information for a human performer to create a musical performance. Similarly, a graphically-oriented computer representation can be designed to hold sufficient information for rendering an algorithmic performance. Therefore, there is no absolute need for storing purely performance-related data in a graphically-oriented representation. Moreover, a performance of a score is not a mandatory part or property of the score itself. (The inclusion of performance data could enhance the usability of a score, but any particular performance should not manifest itself as the only possible interpretation of the score.) Hence, exclusively performance-related aspects can be ruled out of the analysis model.

In SMDL, the analytic domain is mentioned as one form of musical document. Also, Diener mentions analysis as one of the main uses of music notation. In many cases, analysis of a musical score involves addition of analytic symbols to the score itself. In some cases, musical analysis is comparable to performance, because it may yield a different, albeit often visual, representation of the

score. Some music notation programs, such as Nightingale, support Schenkerian analysis (Byrd 1994), which is one example of such an interpretation process. Although many analysis methods make use of music notation, they often use special rules that do not fall within the scope of common-practice music notation. Therefore, Schenkerian and other analytical notation practices have been ruled out of the present analysis model.

In a practical computer application, non-visual objects may be needed; for example, to represent a computer file system, data input and output devices, or for optimization of computing performance. Consideration of the need for such objects is an issue for the design and implementation of a computer program – not the purpose of an analysis aiming at true simulation of a "real life" system.

6.3 Categorization principles

Given that an object-oriented representation is based on modeling visual symbols, the principles for categorizing these symbols should also be considered. In the case of music notation, at least three potential principles can be listed:

1. Shape or appearance
2. Evolutionary origin
3. Function

If the analysis were based on shape or appearance, then classification of objects could be done according to their visual similarity. This approach is used in many general-purpose graphics programs. Examples of names of objects used in such programs are "line", "circle", "ellipse", "rectangle", and "polygon". When applied to music notation, this classification principle would not require much expertise in the problem-domain from the analyst, but would probably yield a model lacking fundamental logical information carried by music notation. Also, graphic terminology could be hard for a musically-educated user to understand. Therefore, it can be argued that exclusive use of general-purpose graphical terminology would render the representation unfit for explicitly musical use.

Shape or appearance may be a usable categorization principle, if the problem domain is quite abstract or is foreign to the analyst. This approach could also be usable when the function or origin of the system is unknown or considered irrelevant. For example, in the analysis of electronic music, different kinds of sounds or sound events can be classified without reference being made to their origin, i.e., the device or synthesis algorithm that produced them. This principle is applied, for example, in Spectromorphology, a method developed by Dennis Smalley (1986) for the analysis of electro-acoustic music.

Evolutionary origin may be a fruitful classification principle, if objects can be regarded as evolutionary offsprings of a single “ancestor” or group of ancestors. For example, the evolution theory of natural science is based on the assumption that all life forms have a common ancestor and thus can be classified according to their assumed evolutionary origin.

Considering the fact that music notation is a continuously evolving system, evolutionary origin could be seen as a valid line of questioning. However, to build a consistent and realistic model would require extensive research on the evolution process of music notation. Furthermore, it can be doubted intuitively that all notation objects could, for example, be traced to a single evolutionary ancestor. After all, music notation is man’s creation – not an independent organism. Therefore, an evolutionary approach can be regarded as theoretically interesting, but impractical.

The functional approach categorizes objects according to what they *do*, e.g., their effects on musical structure. This task requires knowledge of music theory, and decisions on classification are made according it. Function is a relevant classification principle, when, for example, a system is static (non-evolutionary) or when the function of objects is known but the evolutionary origin of objects is unknown or considered irrelevant.

However, since music notation is a visual system, function and appearance cannot be completely separated. In fact, in music notation the function of a symbol may be, in many cases, derived from its shape. More precisely, the function of a notation symbol is derived from two factors: shape and context. Therefore, both function and shape should be taken into account in the classification process. Evolution may be taken into account by making the object structure expandable.

We may derive the function of some music notation symbols through shape alone. For example, a treble clef is complex and unique enough to be distinguishable. On the other hand, the function of simple shapes, such as dots or lines, may be interpreted only within a context. For example, both a note stem and a barline are vertical lines, each possibly having the same length and thickness. Slurs and ties are more complex examples of music notations that have a similar shape but different function. Nevertheless, such symbols may still share enough relevant properties (such as a complex drawing algorithm) to be grouped under a common class or superclass.

To conclude the discussion on classification principles: function can be regarded as the safest primary criterion for the analysis of music notation, as long as one's goal is the computer implementation of music-production tasks. The other two approaches may serve as additional means of identifying and classifying objects. For example, beams and tremolandi, slurs and ties – each of

these have a different musical function, though each pair is closely similar in graphic shape.

As pointed out by Raymond Leppard, the use and correct interpretation of notation symbols have varied from composer to composer, and style to style, throughout the history of Western art music (1988: 27-34). Therefore, misconceptions may arise if symbols in historic manuscripts are categorized solely by their currently-established function. Moreover, since music notation has evolved much over its long existence, it can be expected to continue to evolve in the future. Therefore, to make the model extendable and sufficiently flexible, evolutionary aspects should also be taken into consideration in the classification process.

6.4 Scope of the analysis

In Chapter 2.4, I listed Glen Diener's three basic uses for music-notation programs: compositional, analytical, and archival. Ideally, an analysis model should be applicable to any of these uses. To achieve this goal, in my discussion of musical objects I have omitted all references to how music notation is input, output, edited, or stored. In other words, the purpose has been not to analyze any specific kind of computer application. The aim, moreover, has been not to model even a manual (non-computer-based) process of entering or processing music notation. Rather, the target of analysis is the core of general-purpose and abstract music-notation.

Thus, the analysis aims at providing a description of key abstractions of the common Western music-notation system. In principle, this description should be applicable in the design of any type of computer program that includes capabilities for music notation.

Following this choice, I have not included terms or objects such as files or input/output devices. The analysis contains only objects that can be found in a printed musical manuscript and generalizations of those objects. Still, even a printed score is an application of music notation, which involves application-specific objects such as "paper", "ink", or "glue". These kinds of objects, too, were excluded from the analysis.

Retaining Booch's description of the scope of an object-oriented analysis (1994: 155), my analysis does not include a formal description of the behavior of the object system. As a result, the analysis model does not include UML diagrams for state, use case, interaction, or activity. The purpose of the analysis was not to define a set of rules for describing or implementing the algorithms needed in a practical computer application. These include, for example, the ability of a notation program to check for the correct amount of beats in a measure (e.g., see

Byrd 1994). Also beyond the scope of my analysis are the rules involved in engraving (e.g., see Ross 1970).

The analysis model is based on the assumption, dictated by linear logic, that a musical score may be regarded as a single, hierarchical symbol structure. On a high level of abstraction, a whole score may be regarded as a single symbol. Respectively, each individual note in a score can be regarded as a symbol that may be further divided into smaller, more primitive symbols. In a computer application, symbols may be further decomposed into the primitives of graphics, such as pixels or vectors.

On a low level of abstraction, a single symbol might not have any musical meaning or function. Even on a higher level of abstraction, an object may need additional symbols to be fully functional; in other words, it may require an "environment" or context.

6.5 Application of linear logic

It is not assumed that a linear programming language is used for implementing our object model. The model does respect the rules of linearity, but the treatment of the objects as linear is not assumed. Therefore, associations may be implemented with pointers to memory locations, or in any other way that is supported by the implementation programming language. The distinction between association and aggregation should, however, be respected. An aggregation – or "composition", used in my model instead of regular aggregation – always indicates that the existence of parts is, without exception, dependent on the existence of the "whole", i.e., the aggregate. Respectively, an association may exist, at least in principle, without the existence of the object, called the "associate", referred to by the association. These requirements should be acknowledged in the processes of design and implementation.

6.6 General requirements

To improve and further systematize the analysis model, some general requirements are also presented here. The requirements are mostly derived from the general principles of object-oriented methodology and from systematic software engineering practices in general. Some of the requirements are also derived from the discussion above. The requirements can be summarized as follows:

1. Consistency and simplicity
2. Avoidance of redundancy
3. Independence of notation application area
4. Independence of computer hardware and software runtime environment
5. Independence of implementation programming language

Consistency is achieved by the inclusion of only those objects which have a visual appearance. Nonvisual data or operations are modeled as properties of some visual object or as associations of visual objects. Simplicity is achieved by keeping the amount of objects, relationships and properties small while including enough information to make the model understandable.

Redundancy is minimized by the non-inclusion of objects, properties, relationships, or adornments for a purpose that is already handled explicitly by some other feature in the model. In particular, the model does not contain elements whose main purpose is to enhance computing efficiency. The purpose of the first two requirements is to keep the model compact, coherent and true to the nature of music notation as a visual communication system.

Requirement 3 refers to the uses listed by Diener (1990: 7). The model should be applicable to the design of any type of program that can process music notation. This requirement also helps to keep the model compact by ruling out most references to any detailed, application-specific means of entering, processing or displaying notation data. Requirement 3 does not guarantee, however, that the model is optimal for any specific type of application.

Requirement 4 means that the model should make no assumptions as to what kind of computer, operating system or other hardware/software environment the notation application runs on. This does not, however, guarantee that the model can be implemented on all software or hardware environments.

Requirement 5 means that special features of a particular programming language are not used in the analysis. The analysis should be implementable by the majority of commonly used, object-oriented programming languages. For example, multiple inheritance is not used in the analysis, because it is not supported by all commonly used, object-oriented programming languages.

In addition to the above requirements, the following principle was used to define the vocabulary of the model: *only nouns that are part of the general vocabulary of common music notation can be used as names of concrete classes* (i.e., classes that may have instances). Invented names may be used to denote abstract classes (i.e., generalizations that cannot have instances), if a semantically appropriate noun does not exist in the music notation terminology. The purpose of this principle is to keep the analysis within the scope of the problem

domain. Moreover, if invented names were allowed for concrete classes, the model could become cryptic.

Optimization of storage space and optimization of computing performance were not considered important as modeling requirements. As a result, no assumptions are made about how many or about what type of data elements are used for storing the objects or their properties. Optimizations of performance and storage space are issues that belong to the solution domain. There, knowledge of the specific application area, hardware capabilities, and implementation programming language are needed for decision-making.

As are symbolic representations in general, my object model is an interpretation of the problem domain. Although the aim here is to present an explicit object-representation, some practical requirements have guided my interpretation. One important practical requirement is that an object model must be realizable (implementable) in practice. Moreover, the model should be implementable using reasonable computational resources. On the other hand, the model should be flexible, not demanding the use of any specific algorithm for music-processing tasks, such as for spacing or for generation of a musical performance. These requirements have led to the redefinition or abandonment of some problem-domain terminology.

6.7 Preliminary examples

To illustrate and clarify some central aspects of the model presented in the next chapter, some simple examples are given here. The examples are presented also to show the kinds of things considered when the model was under construction. The examples address the construction of both aggregation and inheritation structures.

My model is intended to be explicit in respect to music notation as a graphic system. It represents each graphic symbol with a dedicated object. As a general principle, everything that can be seen in a printed score has an object representative in the model. Conversely, the level of interpretation with respect to logical information is kept low, as demonstrated in the few examples below.

As described above, music notation is regarded as a hierarchical and graphic aggregation structure, which consists of complex symbols that are constructed from simpler ones. One hierarchical level, the whole score, may be regarded as a single symbol. For example, the score of a symphony or a piano sonata may be regarded as a single, unique symbol. A multi-page score may be subdivided into pages, pages into systems, systems into staves, and so forth.

The difference between graphical, logical, and performance information can be again demonstrated with a pair of examples shown in Figure 6-4. Figure 6-4

a) shows a situation that could have different interpretations depending on the weight of different types of information (as discussed in Chapters 2.6 and 6.3). A central question is, Should the example be interpreted as a single note or as two notes? When the notation example is played, a single sound event is produced – on the assumption that traditional interpretation practice of Western classical music is followed. Yet, visually, we can detect two note symbols that are connected by a tie symbol. An even more ambiguous situation is shown in Figure 6-4 b). There, the tie is divided into two parts because of a page turn or line break. The question here is, Should this example be regarded as containing one or two ties?

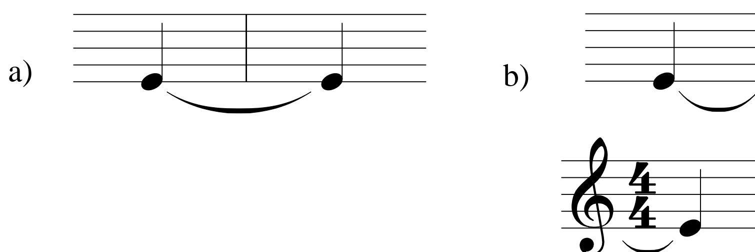


Figure 6-4: Examples of tied notes

It can be argued that Figure 6-4 b) shows logically only one tie, which continues from one page or line to another. Visually, however, there are two tie symbols. The former interpretation can be called more logically-oriented and the latter interpretation more graphically-oriented. The latter, graphically-oriented interpretation is used in my analysis model. Similarly, Example 6-4 a) is modeled as two note-objects instead of one.

The same analysis principle applies to slurs. On some occasions, however, slurs are broken into two or more segments; for example, to provide space for another symbol. Dashed slurs are also sometimes used. In the analysis model, a dashed or a broken slur or tie is regarded as a single symbol. The distinction from the example of Figure 6-4 b) is that both tie symbols shown in the figure can be individually identified and thus named as ties. Also, other notation symbols that can be broken into two or more segments are modeled as a single object; e.g., tuplet brackets broken into two part to provide space for a tuplet number (see Geroy & Lusk 1996: 289, 341-346). On a lower, more generic graphic level, such segmented symbols can be treated as constructs of separate objects. This is regarded here, however, as an issue for software design or implementation.

The analysis model contains a multi-level hierarchical aggregation structure. There, the "part-of" relationships reflect the assumed importance of notation symbols. For example, a note is considered to be more important than, for example, accidentals or articulation symbols. As a general principle, symbols whose existence or function is considered to be dependent on some more important object are modeled as parts of that more important object. Thus, accidentals and articulations are modeled as parts of a note.

To summarize, the analysis model presented in the next chapter uses graphic information as its basis. The level of abstraction is higher, or more domain-specific, than are individual pixels on a computer screen or printed paper, and higher than line or curve segments, which are the basic elements of vector-based graphics programs. The level of abstraction is even higher than "paths" or areas, which are the basic levels of abstraction in many drawing programs, such as Adobe Illustrator. The chosen level of abstraction is intended to be understandable to a music engraver and/or a musician.

In the analysis model, logical information is embedded in the graphic objects as properties or relationships between objects. Performance information is represented mainly implicitly. An interpretation process that simulates a human musical performance is required in order to retrieve explicit performance information from an implementation of the model.

Chapter 7
The analysis model

This chapter presents an analytic model of music notation as a set of UML class diagrams. The model is not a complete formal description of the common Western-music notation system. The amount of detail is suppressed to keep the central aspects of classification decisions understandable. Too much detail could easily blur the overall view of the problem domain.

This model presents neither a complete collection nor full description of objects that can be found in common music notation. Rather, the analysis model shows the main structure of an object system that should provide a usable basis on which to build a computer application. The model should be expandable, that is, adaptable to new objects, if such are needed for a specific application. Also, the amount of properties in the objects themselves is small. The main reason for this is readability. Properties are presented only if they are fundamental to understanding the function of the object in question, or if they help to clarify a decision made in the analysis. Thus, the properties presented in the analysis model should be considered neither necessary nor sufficient for implementing a working program.

All classes in the model may not be needed in every computer application. On the other hand, an application will likely need additional classes and properties to handle such tasks as data input, data output, and storage. Furthermore, embellishments or adornments, such as multiplicity definitions, should be considered as suggestions rather than as mandatory requirements. Practical implementation issues, such as storage space requirements or optimization, may require the use of more or less strict adornments, depending on what is needed.

7.1 General definitions

Presentation of the analysis model begins with the root class of the inheritation hierarchy (called CMNSymbol). This is followed by a roughly top-down decomposition process, starting from the top-level aggregate class (called Score) and proceeding down to elementary notation symbols (notes, rests, barlines, slurs,

beams, etc.). Inheritation and association structures are explained in my discussions of the aggregation hierarchy.

The following general definitions are given for interpreting the class diagrams:

1. The immediate superclass of every class is CMNSymbol, unless defined otherwise.
2. If the amount is not shown in an aggregation/composition relationship, then the amount is one-to-any.
3. If amount adornments are not shown in an association, then the amount is arbitrary ("any-to-any").

It is further assumed that all class names presented in the model reside in a unique name space. For example, the class Note refers only to the common music notation object called "note" and to nothing else called a note, that is to say, not to any similarly named object outside the domain of common music notation.

Generally, the difference between an aggregation and an attribute is as follows: if a thing is a class, then it is shown as an aggregation structure; if a thing is an atomic value, such as an integer or floating point number, then it is modeled as an attribute. However, the distinction between a class and an atomic value is not always clear or relevant. In some cases, attributes are also used as a "short hand" substitute for aggregation. A short verbal description is given of each class.

As in UML in general, names of abstract classes – i.e., classes that cannot have instances – are written in italics. In the analysis model, abstract classes are used sparingly. The main purpose for declaring a class abstract is to show that the class name is invented rather than part of the established vocabulary of the problem domain. Such classes are considered as not representing any identifiable music notation symbols. Rather, abstract classes represent some common features of a group of identifiable symbols.

Neither the diagrams nor the explanatory text of the analysis model contain illustrations of any music notation symbols. Instead, class names are chosen so that the respective notation symbols can be looked up in Gerou & Lusk (1996), Heussentamm (1987), and similar guidebooks.

7.2 The CMNSymbol class

Figure 7-3 shows the class CMNSymbol. It is the superclass of all other classes in the model. Thus, CMNSymbol holds the properties common to all classes in the model. In Figure 7-3, two operations are shown: Draw and Play. Draw

causes CMNSymbol to render a visual representation of itself. In turn, Play is used for rendering a sonic representation of the object. Because all objects within the problem domain are considered to be visual, Draw will always produce some form of graphic output. The Play operation can produce a sound event (e.g., if the object is a note), or it can affect the sonic performance in some other way, for example, by causing a pause, transposition of subsequent notes, change of tempo, and so on. Play is the only purely performance-oriented property in the model, and it is presented only as an example of a way to derive performance information from the graphically-oriented object system.

CMNSymbol
origin size dimensions value
draw play

Figure 7-1: The CMNSymbol class

Each CMNSymbol is assumed to manage a two-dimensional coordinate system, henceforth called the “internal coordinate system”. Each CMNSymbol is also assumed to be positioned in an “external” coordinate system, which is managed by its aggregate object – typically, another CMNSymbol. The origin attribute determines the position of the object’s internal graphic origin within the object’s external coordinate system. The size attribute determines the object's size (i.e., a scaling factor) relative to the size and dimensions of its aggregate object. The dimensions attribute determines the dimensions of the object within the external coordinate system.

The positions of all parts of a CMNSymbol are manipulated relative to the origin of CNMSymbol’s internal coordinate system. As a consequence, when a CMNSymbol is moved to a new location, all its parts move with it, and retain their relational position within their aggregate CMNSymbol. The same applies when a CMNSymbol is scaled in size: all its parts are affected by the same scaling factor. However, specialized subclasses of CMNSymbol may treat position and size information more intelligently than just as mechanical factors.

CMNSymbol is also defined to hold a "value" of unspecified type. The purpose of the value attribute will vary between various subclasses. For example, in a time-signature class, the value would hold the value of the time signature; e.g., a fractional number such as 2/4, 3/4, etc.; or if needed, a more complex data type. In a notehead class the value could hold the type of notehead (e.g., open, closed, square, round), and so on with other types of value.

The techniques and data structures that a CMNSymbol uses to store and manipulate its parts are undefined. The analysis model shows aggregation hierarchies, but does not specify whether an aggregate uses a single data structure to store all its parts, or if it uses distinct attributes for each part. Some exceptions to this principle are made, however, in order to improve readability. Detailed specification of the internal structure of CMNSymbol is considered a design issue. An example design-model of these properties is described in Chapter 9.2.

7.3 The top-level aggregation structure

A printed musical score is modeled as a hierarchical structure where the score itself is considered as one complex symbol. The score, in turn, consists of pages, which may contain other notation symbols, but might also contain conventional text or graphics. The analysis model follows this kind of hierarchical organization, but is limited to music notation symbols only. Conventional text and/or graphics are ruled out unless they have a distinct musical or notational function; e.g., song lyrics or names of instruments.

Figure 7-2 presents the aggregation hierarchy starting from the top-level object, Score (on the left) to Staff (on the right). There, a Score contains an arbitrary number of Pages. A Page, in turn, contains an arbitrary number of Systems or Staves. There is an important restriction, not shown in the UML diagram but defined in the linear formal rules in Chapter 5.5. This restriction is that a Staff instance may either be part of a System or of a Staff, but not part of both at the same time. A System contains an arbitrary amount of Staves. All four of these classes – Score, Page, System, Stave – are subclasses of CMNSymbol. Here, a page of music notation is considered to be read from top to bottom and left to right. Special notational layouts, such as circular staves, are ruled out of this analysis.

The aggregation between Score and Page is defined as "ordered". Although Score is a subclass of CMNSymbol and thus holds a coordinate system to organize its parts, Pages are not considered to be stored within a single, two-dimensional coordinate system. Instead, Pages can be regarded as a kind of third dimension within a document. Pages are defined as being ordered within a Score because (1) the reordering of pages results in a different and usually incorrect

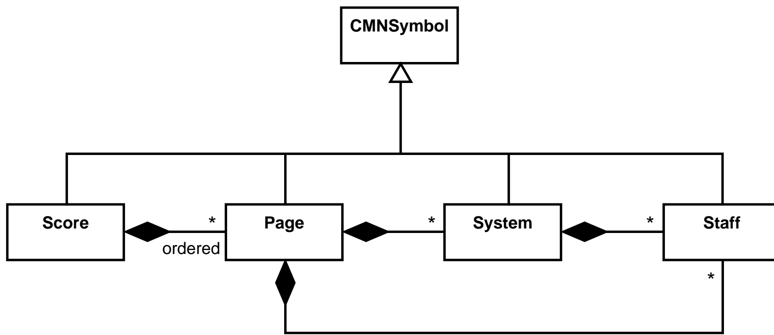


Figure 7-2: Top-level aggregation structure

interpretation of a score, and (2) because no other mandatory property defines their placement in relation to each other. Nonetheless, an implementation of a Score object might be able draw two or more Pages on a single sheet of paper (or in the same window in a computer display). To achieve this, Score would need to manage a two-dimensional, internal coordinate system like that of other CMNSymbols.

Ordering of other aggregations in Figure 7-2 is undefined, because the objects of parts are located in the two-dimensional coordinate system of their aggregate object. Since the graphical position of each System is known by its aggregate (i.e., Page), the Systems can be regarded as ordered "implicitly" by their graphical locations within the Page's internal coordinate system. The same principle of implicit ordering applies also to Staves within a Symbol, with the functional distinction being that staves within the same system are read or played in parallel, instead of successively from top to bottom.

CMNSymbol is the root of the inheritation hierarchy, and Score is the root of the aggregation hierarchy. This means that all objects are either direct or indirect parts of a Score object. In the rest of this chapter, inheritation hierarchies are simplified by the omission of the CMNSymbol from the class diagrams.

In music notation, staves can be grouped to form systems. This organization is defined visually by drawing a “systemic barline” that connects the staves. Other connecting symbols, too, such as vertical brackets and braces, are used to form groups and subgroups within a system. The analysis model contains a SystemicBarline, a subclass of a generic Barline class (described below). SystemicBarline is not defined as a direct part of System, but instead as a part of Staff. This is so because the superclass Barline is already defined as being part of Staff. By this definition, Staff can contain any types of Barlines, including SystemicBarlines. The specialized relationship between SystemicBarline and System is defined by an association. The association indicates that, when a System is present, there is always a SystemicBarline associated with it. The other connecting symbols are modeled with StaffConnectors. StaffConnector is a superclass for vertical brackets or braces (classes Bracket and Brace), which represent various, additional forms of grouping within a system.

Figure 7-3 shows a model of the System class and its main contents. Along with Staves, System contains an arbitrary amount of StaffConnectors, which are either Brace or Bracket objects. A StaffConnector object is associated with one or two Staff objects. As described above, a System is associated with a SystemicBarline.

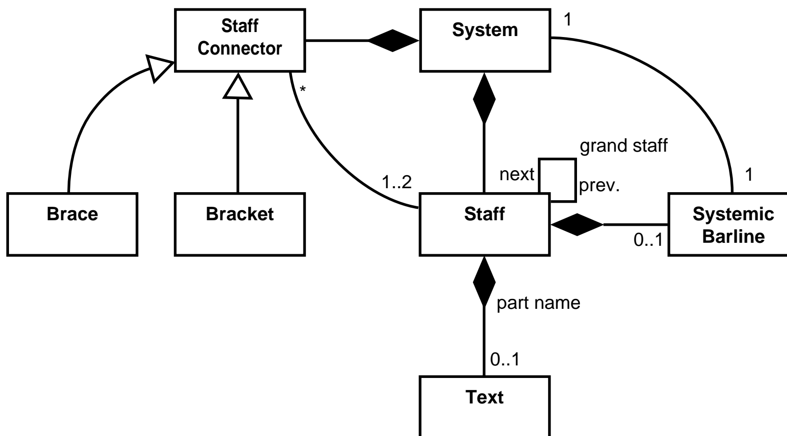


Figure 7-3: System-level structure

Brace and Bracket are symbols that form logical groups and subgroups within a System. The groups are not, however, direct equivalents of instrumental sections, of parts, or of voices. As stated in the previous chapter, music notation does not contain an unambiguous symbol for representing part, section, or voice. The model permits a part to be written on more than one staff (such as a piano part), or one staff may include several parts. Nevertheless, parts, sections, and voices are not represented as classes or even as associations in the analysis model.

7.4 Staff

Staff is a central aggregate in the model. Staff's parts include staff lines as well as many other symbols, including notes, rests, clefs, barlines, slurs, dynamic markings, etc. These symbols are divided into two abstract classes: CoreSymbol and Attachment. CoreSymbols are symbols that have a dominant role in forming a line of music, both compositionally and in forming the graphic layout. Examples of CoreSymbols are notes, rests, clefs, and barlines. Attachments are symbols that affect the ways in which CoreSymbols are interpreted; in particular, notes and rests.

As shown in Figure 7-4, Staff consists of an arbitrary amount of StaffLine objects. Staff also contains an arbitrary amount of CoreSymbols, Attachments, and LedgerLines. Through the use of association, staves can be grouped so as to form a grand staff. The "grand staff" association has the roles "next" and "prev." (short for "previous"), by which two instances of Staff can refer to each other.

CoreSymbol is a generalization that represents symbols that can be considered elementary or "primary" in music notation and that thus form a "core" of music notation. "Attachment" is a generalization for classes; it represents symbols that add information to and are, in some form, "attached" to CoreSymbols. Generally, Attachments hold little or no musical information without the existence of at least one CoreSymbol. On the other hand, legitimate and readable scores, at least of simple musical pieces, can be written with only Staves and CoreSymbols.

Subclasses of CoreSymbol share some functionality both logically and graphically. The main reason for the classification is derived from the graphic placement of the objects. They are generally placed on staff lines or on ledger lines. Also, they all have an explicit musical function in themselves, as long as they are placed on a staff. Attachments, on the other hand, are generally placed either above or below a staff, and gain explicit musical meaning only when they are attached to (associated with) a CoreSymbol.

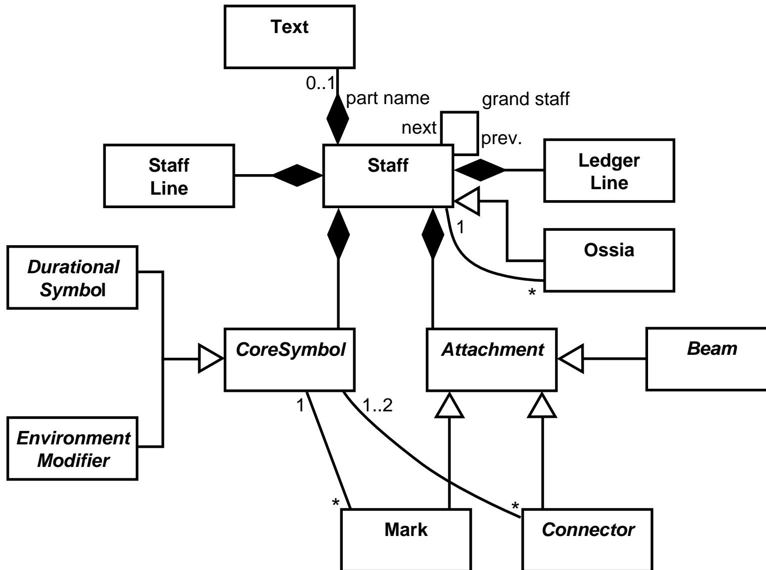


Figure 7-4: Staff-level structure

CoreSymbol has the subclasses DurationalSymbol and EnvironmentModifier for representing two different kinds of symbols classified by their musical function. The next sections are dedicated to describing DurationSymbols and EnvironmentModifiers. Attachment has the superclasses, Mark and Connector, which are described in section 7.8.

Ossia (Gerou & Lusk 1996: 228) is defined as a subclass of Staff. The association of an arbitrary number of Ossia objects with one instance of Staff is described further on.

7.5 DurationalSymbol

DurationalSymbol is a CMNSymbol that instructs a performer either to produce a sound event or to hold a pause of a specified duration. With its primary subclasses Note and Rest, DurationalSymbol forms the inheritance and aggregation structure shown in Figure 7-5. DurationalSymbol is a generalization that

includes the common properties of Note and Rest, as well as properties that affect the duration and spacing of symbols on a staff. DurationalSymbol contains an arbitrary amount of AugmentationDot objects. DurationalSymbol is also associated with Pause, which is a subclass of Mark. Pause is described in more detail in section 7.8.2.

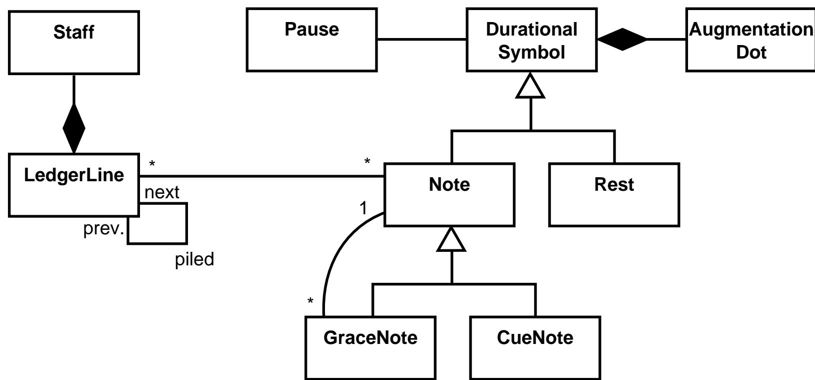


Figure 7-5: DurationalSymbol class structure

Notes and rests have common features both musically and in engraving practice. In particular, they determine the rhythmic structure of music. In engraving practice, similar rules are used to reserve space horizontally on a staff for both notes and rests. There are also differences; for example, in the spacing of whole rests versus whole notes (see Ross 1970: 77-78). Still, the amount of similarities justify the grouping of notes and rests under a common superclass.

A Rest is constructed of a rest symbol whose shape determines the basic duration of the rest. Additional symbols that modify the durational properties of Rest (e.g., Dots) are inherited from DurationalSymbol. Note is a more complex symbol. Its structure is described in detail in the next section.

7.6 Note

A detailed diagram of the Note class and its parts is shown in Figure 7-6. Note is a highly complex class that relates with several other classes. Some of Note's

properties are inherited from DurationalSymbol; for example, the association with AugmentationDot. The latter two classes and the respective relationships were shown in figure 7-5

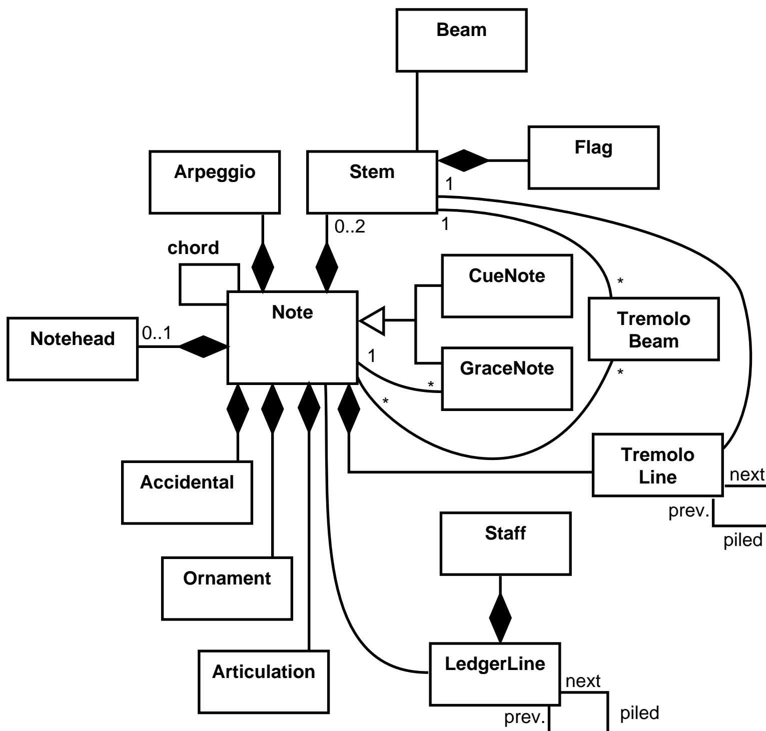


Figure 7-6: Note class structure

Notehead is a central, but not mandatory part of a Note object. Notehead is modeled as a dedicated class. The value attribute of Notehead, inherited from CMNSymbol, is assumed to hold the type of note head (e.g., open half-note, open whole-note, closed, etc.). In some situations the Notehead may be absent, as when rhythms are denoted only with stems, flags and beams. This is indicated by the 0..1 adornment in the aggregation between Note and Notehead.

Note contains from zero to two Stem objects. Music notation sometimes uses Notes with two stems pointing in opposite directions, as in two-voice parts where both voices share a common note head.

A Stem may contain an arbitrary amount of Flags. Stem objects of different Notes may be beamed. Beam, which is part of Staff, is described in more detail in sections 7.9. Note objects may also contain Accidental, Articulation and Arpeggio objects.

A Note may be related with other Notes through a “chord” association. Here, Chord indicates a group of notes that are stacked one above the other, are of equal note value, and, if the note value requires the use of a stem, share the same stem.

Note has two subclasses: CueNote and GraceNote. They differ from plain Notes by their interpretation and their visual appearance: smaller size and, in the case of grace notes, spacing conventions. GraceNote is also associated with one Note.

Note may be associated with LedgerLine objects. LedgerLines may be stacked, indicated by a respective association. LedgerLine is part of Staff.

Note has no attribute for explicitly representing pitch or duration (i.e., a time value). The reason for this, is that pitch is implicitly defined by the note’s vertical position on the staff and by the preceding clef, key signature, accidentals and barlines. A Note’s value is formed as a combination of a note head, an optional stem, beaming, flags, and dots.

As described by Gerou and Lusk (1996: 26-44), Articulation indicates “how a note or chord is played”. The authors describe five “main” articulations (staccatissimo, staccato, tenuto, accent, and marcato), three articulations of force (sforzando, forzando/forzato, and sforzato), and simile. According to the model, more than one Articulation may be attached to a single Note. Ornaments are graphic symbols that can be regarded as macro statements, where a written, ornamented note is played as more than one note.

TremoloBeam may be associated either with a Stem or directly with a Note. The former association refers to situations in which stems are connected with tremolo beams. The latter association applies when no stems exist. A Note may contain TremoloLine objects, which may be associated with a Stem. TremoloLines, which are part of the same Note, are also associated with each other, as defined by the “stacked” association.

7.7 Environment modifiers

The EnvironmentModifier class is a subclass of CoreSymbol. It represents various symbols that apply changes to the environments of each other or of other

notation symbols, of Notes and Rests in particular. EnvironmentModifiers may affect either the rhythmic or harmonic structure of a Staff, a group of Staves or a whole System. Similarly to other CoreSymbols, EnvironmentModifiers play a dominant role in spacing.

A class diagram of EnvironmentModifier and its subclasses is shown in Figure 7-7. The subclasses of EnvironmentModifier are Clef, KeySignature, TimeSignature, and Barline. Like other CoreSymbols, EnvironmentModifier is part of Staff. An EnvironmentModifier may affect more than one system. For example, a Barline may be drawn across several staves. This behavior is modeled using the association with the "extends to"-role between EnvironmentModifier and Staff. Each instance of EnvironmentModifier is always part of one and only one Staff, while it can "extend to" an arbitrary amount of other Staves.

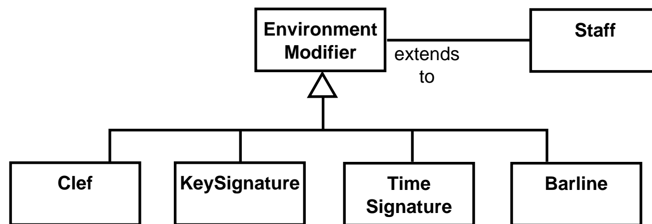


Figure 7-7: Environment modifiers

Barlines are further divided into the set of subclasses shown in Figure 7-8. The subclasses of Barline are SingleBarline, DoubleBarline, and FinalBarline. SingleBarline is a single-line symbol that marks off measures; it is the most common type of barline. DoubleBarline is a two-line barline with both lines having the same thickness. FinalBarline represents the end of a passage of music, and in some cases, the beginning of a new one. The passage(s) may be repeated. A FinalBarline symbol consists of two (or sometimes three) vertical lines, typically one thinner and one thicker line. FinalBarline may contain an arbitrary amount of RepeatDots (typically two for each Staff per FinalBarline). Two FinalBarlines may be adjoint (marked by an association). Two adjoint FinalBarlines may share a common thick line (see Gerou & Lusk 1996: 246). Both SingleBarline and DoubleBarline may be dashed.

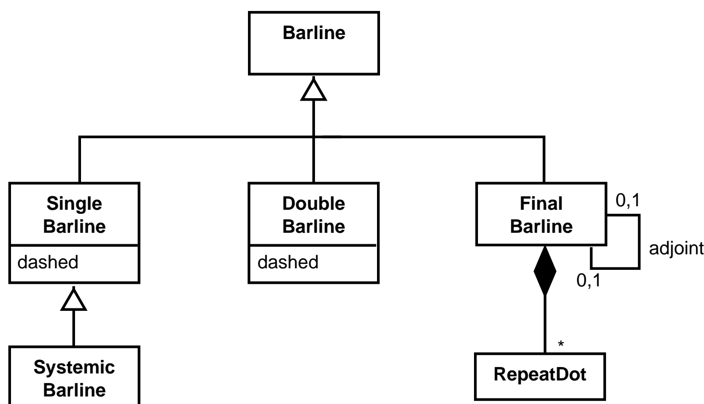


Figure 7-8: Types of barlines

Ross describes two special kinds of barlines: systemic barlines and connecting barlines (1970: 151-152). Gerou and Lusk also mention the systemic barline as a symbol that groups staves into systems (1996: 56, 308). Systemic barlines are modeled with a respective class, SystemicBarline, defined as a subclass of SingleBarline. The analysis model does not contain a class for connecting barlines. Instead, the diagram in Figure 7-7 shows that any Barline, through its superclass, EnvironmentModifier, may be associated with an arbitrary number of Staves, including the Staff which the Barline is part of. SystemicBarline is modeled as a dedicated class to indicate its specialized function of connecting staves to form systems.

7.8 Attachments

As presented in Figure 7-4, the Attachment class contains the subclasses Connector and Mark. Connector is a superclass for symbols which, either tightly or loosely, connect two or more other symbols. Mark represents an instantaneous event, although logically it may signify the beginning of a gradual progression.

The difference between Connector and Mark can be described as follows: Connector marks a musical structure that has an explicit beginning and end. In contrast, Mark is either an instantaneous event or beginning of a process which

has either an implicit end or in which another symbol states the end. Examples of Connectors are slurs and ties. Examples of Marks are dynamic expressions (***f***, ***mf***, ***ppp***, etc.). They can be regarded as symbols that set a condition, which stands until a new condition of the same type is encountered.

Attachments are not aggregates of the symbols they affect. Thus, Attachments do not contain notes, rests, or other, equally high-level symbols. The relationships between Attachments and Notes, for example, are modeled as associations.

7.8.1 Connector

A classification of Connector symbols is shown in Figure 7-9. A Connector symbol may be logically continued, for example, to the next line or page. The continuation is modeled as an association between two Connector instances. It is assumed that a Connector is associated with the CoreSymbols that start and end some passage, indicated by the Connector.

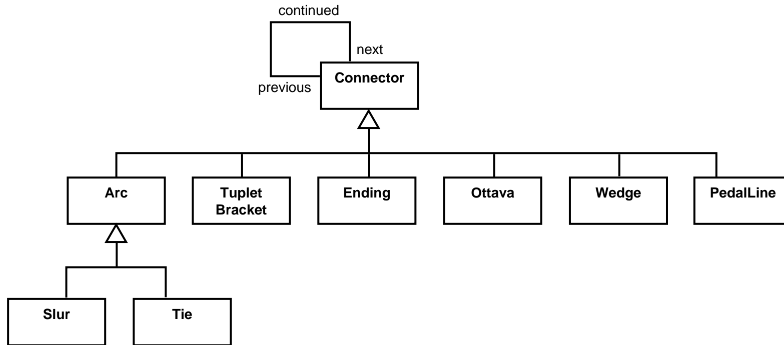


Figure 7-9: Connectors

The subclasses of Connector are Arc, TupletBracket, Ending, Ottava, Wedge, and PedalLine. Slur and Tie share a common visual shape, but they differ in graphical placement and function. Thus, they are modeled as subclasses of Arc, which represents their common properties. Tuplet represents both tuplet brackets and numbers. Ending represents ending brackets and numbers. Ottava is assumed to include both a text expression (e.g., "8va") and a horizontal line, although this is not explicitly shown in the UML diagram. Wedge represents graphic crescendo and diminuendo symbols (see Gerou & Lusk 1996: 134). In SCORE terminology, wedges are called "hairpins". PedalLine represents only pedal symbols that contain a line indicating how long the pedal is held. An instantaneous pedal mark is a subclass of Mark (see below), and is not considered to be a Connector.

7.8.2 Marks

The class diagram of the Mark class and its subclasses is shown in Figure 7-10. The subclasses of Mark are DynamicMark, PedalMark, Pause, TextFrame, and Tempo. DynamicMark objects include common text abbreviations such as ***f***, ***ff***, ***fff***, ***p***, ***pp***, ***fp***, etc. Text-based markings may also represent gradual changes to dynamics belong to this class; e.g. ***dim.*** or ***cresc.*** etc. Graphic dynamic symbols, or Wedges ("hairpins"), are modeled as subclasses of Connector.

PedalMark is a class for both text (***ped.*** etc.) and graphic symbols. Pause represents fermata signs and pause signs. As described above, pedal lines are represented by the PedalLine class, which is a subclass of Connector. Tempo represents symbols that set or modify speed. These include metronome marks, text tempo expressions (e.g., ***Allegro***), and tempo modifiers, such as ***accel.***, ***rall.***, and so on.

TextFrame is divided into the subclasses Lyric and RehearsalMark. Lyric represents a fragment of a song lyric, typically a single word or syllable, including a trailing hyphen. RehearsalMark represents both rehearsal numbers and rehearsal letters. TextFrame objects may be associated with each other by flows. This follows the principle used in many text processing and page layout programs, such as FrameMaker (Branagan & Serra: 1994: 156-157).

Mark could be extended with more subclasses. Also, some of Mark's subclasses could be divided into more specialized subclasses. For example, guitar-chord frames (Gerou & Lusk 1996: 103-106) or special percussion symbols are potential candidates for subclasses of Mark.

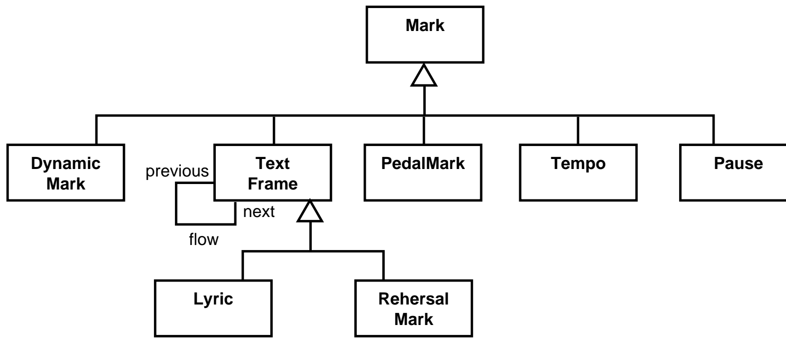


Figure 7-10: Marks

7.9 Beams

Beams can be divided into several subclasses. Also, a complex set of associations is defined. The class diagram is presented in Figure 7-11. The superclass Beam is divided into the subclasses RegularBeam and TremoloBeam. RegularBeam represents beams that substitute flags for indicating various note durations. TremoloBeams represent beams that are used to form tremolandi with two or more consecutive notes or chords. RegularBeams are always connected to stems, whereas TremoloBeams may be used also with notes that do not have stems; e.g., whole notes (see Gerou & Lusk 1996: 334).

Beam is part of Staff. If a Beam spans across several Staves, for example in a piano score, only one of the Staff instances may contain the Beam as its part.

RegularBeam is divided into subclasses PrimaryBeam and SecondaryBeam (see, e.g., Gerou & Lusk 1996: 62-89). SecondaryBeam may also be a FractionalBeam (ibid.: 333-334). SecondaryBeams are “aligned” with a PrimaryBeam (marked in Figure 7-11 with an association). This way a PrimaryBeam knows which SecondaryBeams belong to the same beam group and can, for example, control their shape and placement. Both PrimaryBeams and SecondaryBeams

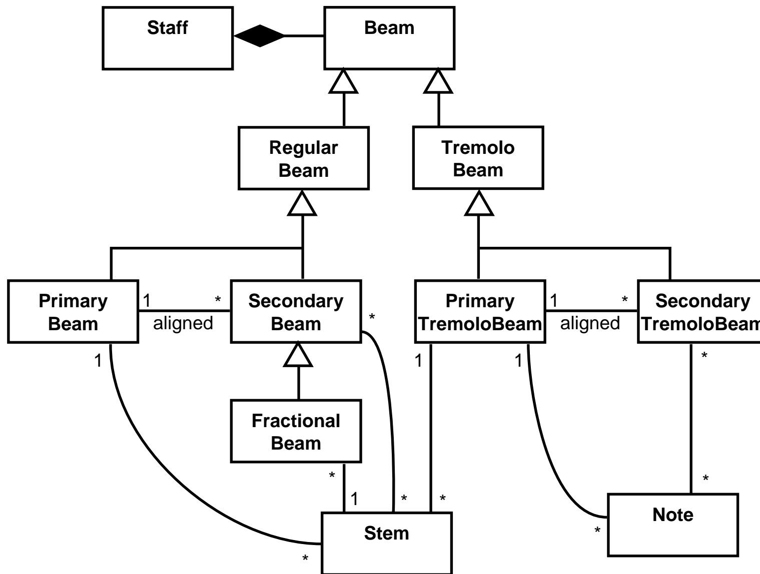


Figure 7-11: Class diagram of Beam and its subclasses

are associated with an arbitrary amount of Stems, while a FractionalBeam is associated with only one Stem.

TremoloBeams are divided into PrimaryTremoloBeams and SecondaryTremoloBeams. When TremoloBeams are used with notes that have stems, a PrimaryFractionalBeam is connected visually to the stems; respectively, SecondaryTremoloBeams are placed in-between the stems but are kept separate from them. When used with whole-note symbols, both types of tremolo beams are kept separate from other symbols, and they also have the same length. The connection between a Stem and a PrimaryBeam is modeled as an association between the two classes. SecondaryTremoloBeams are not connected to Stems, and therefore not associated with a Stem. An association is, however, defined between both types of TremoloBeams and Note, for situations in which a Note has no Stem.

Chapter 8
Object diagram examples

This chapter contains examples of selected features of the analysis model. The examples are presented as music notation and their respective object diagrams. The order of the examples follows approximately that of the previous chapter. The examples are deliberately simplified in order to save space and enhance readability.

8.1 Systems and Staves

The following note example shows a system with empty staves. Figure 8-1 a) presents a note example with a system containing two staves plus a two-stave grand staff encoded by a brace. The whole system is enclosed by a bracket. Figure 8-1 contains a respective UML object diagram. There, Stave objects are named as st1 through st4, from topmost staff (st1) to the lowest one (st4). The System sys1 is part of Page pg1, which is part of Score sc1. The System instance sys1 is the direct aggregate of all other objects, except the Barline object b1. The Brace object br1 is associated with Staves st3 and st4. St3 and st4 are also associated with each other through a “grand staff” association.

The Bracket object bkt1 is associated with the highest and lowest staff (st1 and st4). The SystemicBarline object is part of the Staff st4 and is associated with all other Staves as well as the System object sys1.

8.2 CoreSymbols

Figure 8-2 a) is a simple note example. The respective object diagram is shown in Figure 8-2 b). The note example is the same as in Chapter 3. There, Note objects are named n1 through n4 for respective notes progressing from left to right in the note example. Score and Page objects are omitted from the diagram, although they must exist in a complete object system, according to the analysis model.

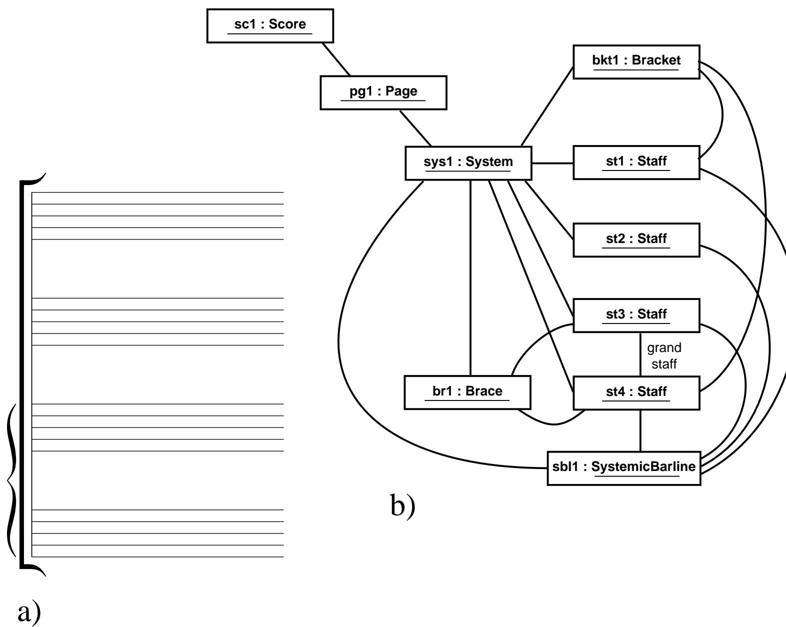
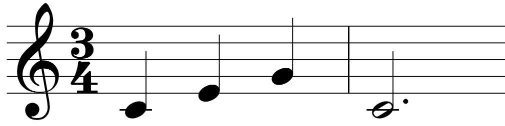


Figure 8-1: Example of a system with staves and grand staff

The value attributes of the Notehead objects are not shown in the diagram. If the value attribute held a value indicating the type of Notehead, the value could be, for example, “closed” for objects h1 through h3 and “open” for h4.

It is assumed that the position and dimensions of the Note objects are stored in the respective attributes inherited from CMNSymbol. It is also assumed that these parameters are scaled in relation to the Staff object’s coordinate system.

The object structure shown in Figure 8-2 b) is partly similar to the example of the SCORE Parameter List shown in Chapter 3.8. The main differences are that the object diagram has a deeper aggregation hierarchy. On the other hand, the object diagram includes less parametric detail. This demonstrates that the analysis model is not ready to be used in a computer program. Yet, by the addi-



a)

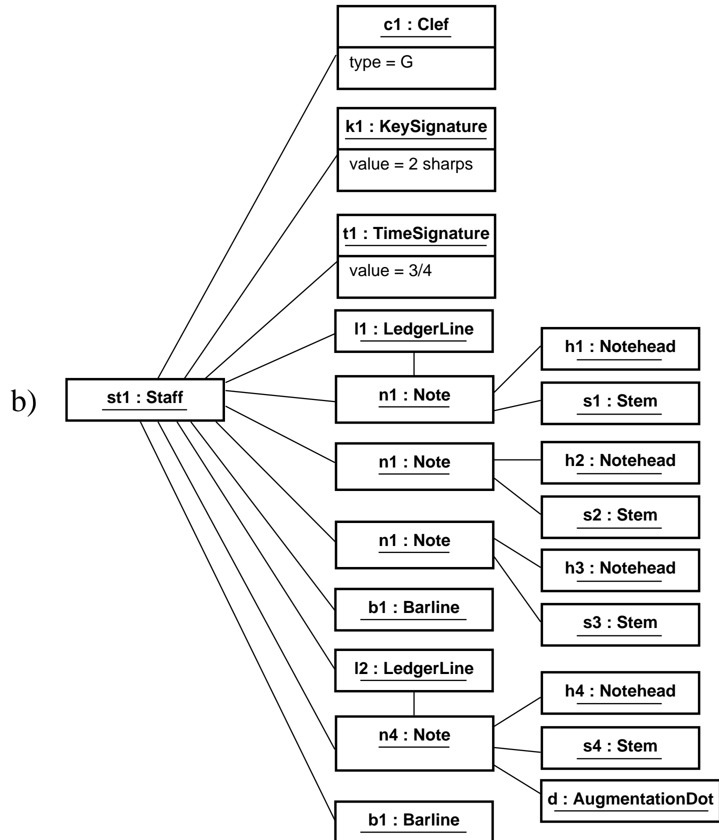


Figure 8-2: A two-bar note example and equivalent object diagram

tion of relatively few parameters, such as graphical coordinates for each symbol and note head types, a simple SCORE-like application could be implemented.

Obviously, a conventional, general-purpose notation program would require many more parameters for each individual class.

8.3 Chords and Stems

Figure 8-3 shows an example of a chord made up of three half notes, along with its respective object diagram. There, the three note objects (n1, n2, and n3) are connected by a “chord” association. The stem is part of the Note n3. Notehead objects have been omitted from the diagram.

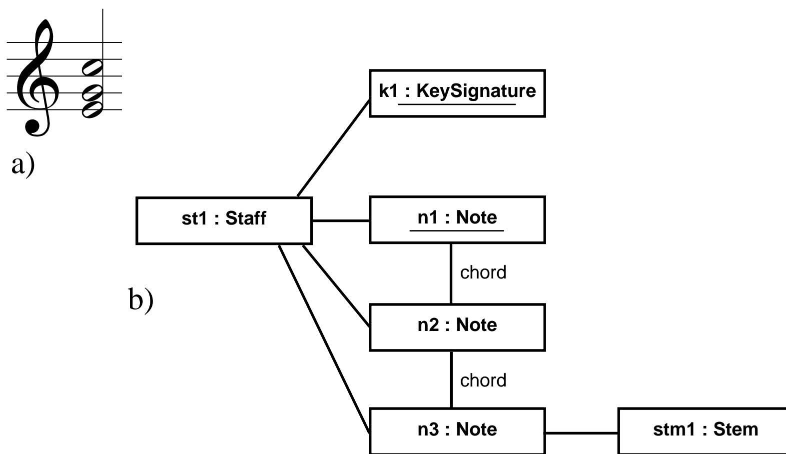


Figure 8-3: Sample chord

8.4 Beams

Figure 8-4 presents an example of two beamed notes. A staff is omitted from the note example as well as from the respective object model. The example contains a dotted 16th note and a 32nd note. The stems of the notes are connected by a primary and a secondary beam. In addition, the stem of the 32nd note is associated with a fractional beam. Though not shown in the diagram, Note and Beam objects are parts of a Staff.

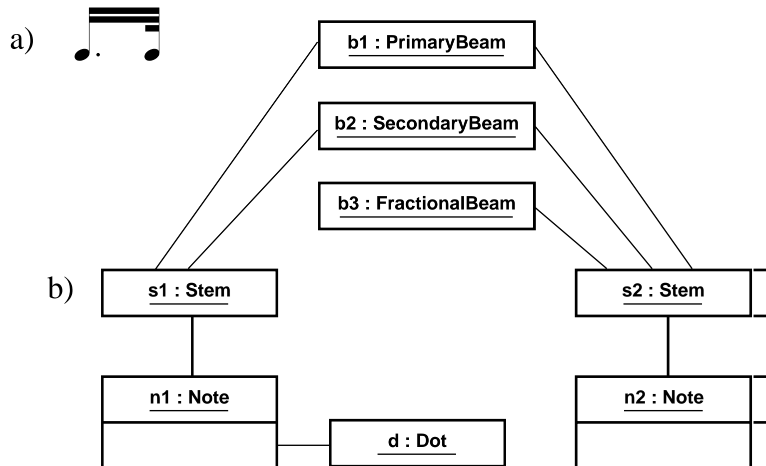


Figure 8-4: Example of two beamed notes

An interesting question about implementation arises if an application program is to produce the notation of Figure 8-4 a); i.e., Notes or other objects without a visible Staff. There would be at least three alternative ways to implement it. (1) The object system could implement a way to "hide" either certain types of symbols or any symbol at all. Thus, any or all of a Staff's lines could be hidden. (2) A staff with zero staff lines could be used, which is already allowed by the analysis model. Such a "lineless" staff would be a coordinate system invisible in itself but implicitly visible through the symbols (other than staff lines) that it contains. (3) The situation could be regarded as a subset of the aggregation structure of the analysis model. There, all aggregates in the hierarchy would be reduced from the object system down to (and maybe including) the Staff class.

8.5 TremoloBeams

Figure 8-5 a) shows a note example with two half note symbols connected by a group of tremolo beams. A respective object model is presented in Figure 8-5 b).

As indicated in the class diagrams in Figures 7-6 and 7-11, TremoloBeam symbols are related to Notes and Stems in a complex way. This makes TremoloBeams difficult to implement.

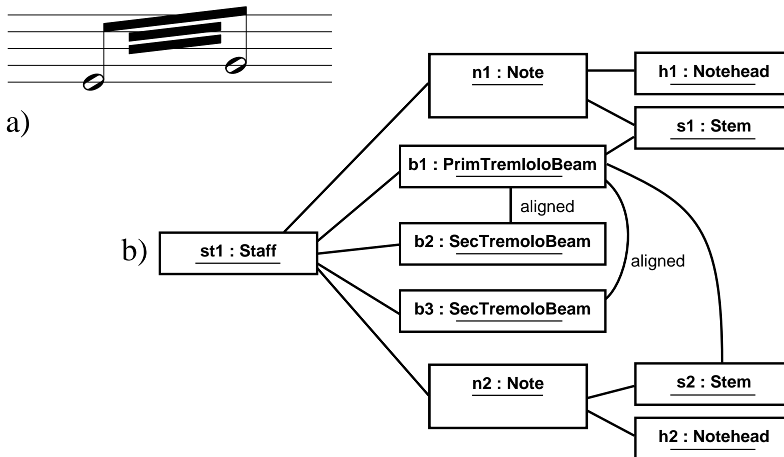


Figure 8-5: Examples of tremolo beams

In the object diagram of Figure 8-5 b), the Staff object st1 contains the Note objects n1 and n2 as well as the TremoloBeam object b1, b2, and b3. The class names – PrimaryTremoloBeam and SecondaryTremoloBeam – are abbreviated in the model. The PrimaryTremoloBeam object is associated, or “aligned”, with both SecondaryTremoloBeam objects. The PrimaryTremoloBeam object is also associated with the Stem objects of both Notes. The object model also contains Notehead objects as parts of each Note.

8.6 Ties and Slurs

Figure 8-6 shows the same note example as in Figure 6-4 a) together with a respective object diagram. The diagram shows a Staff object containing two Notes, a Barline, and a Tie which is associated with both Notes.

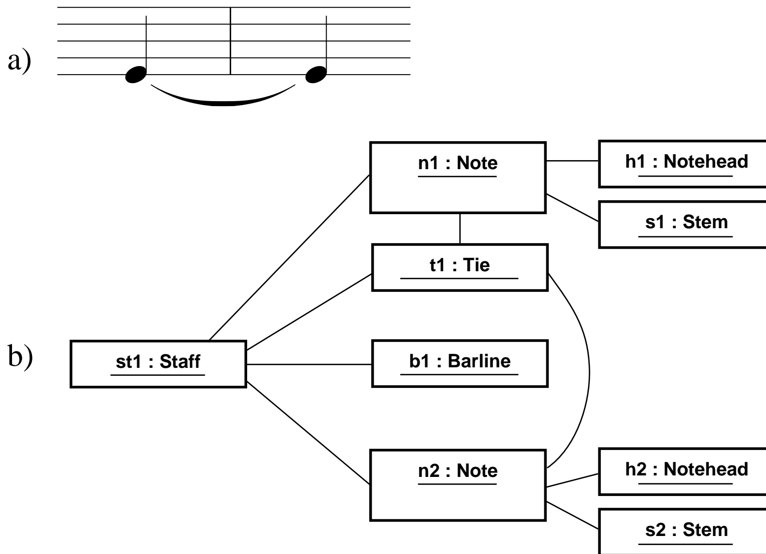


Figure 8-6: Examples of tied notes

Figure 8-7 shows an object diagram of the note example of Figure 6-4 b). It shows two Staff objects, both containing a Note object and a Tie. The Tie objects are connected by an association. The value attribute of the Clef object c1 indi-

cates the type of clef ("G"). Respectively, the value attribute of the TimeSignature object t1 indicates the time signature 4/4.

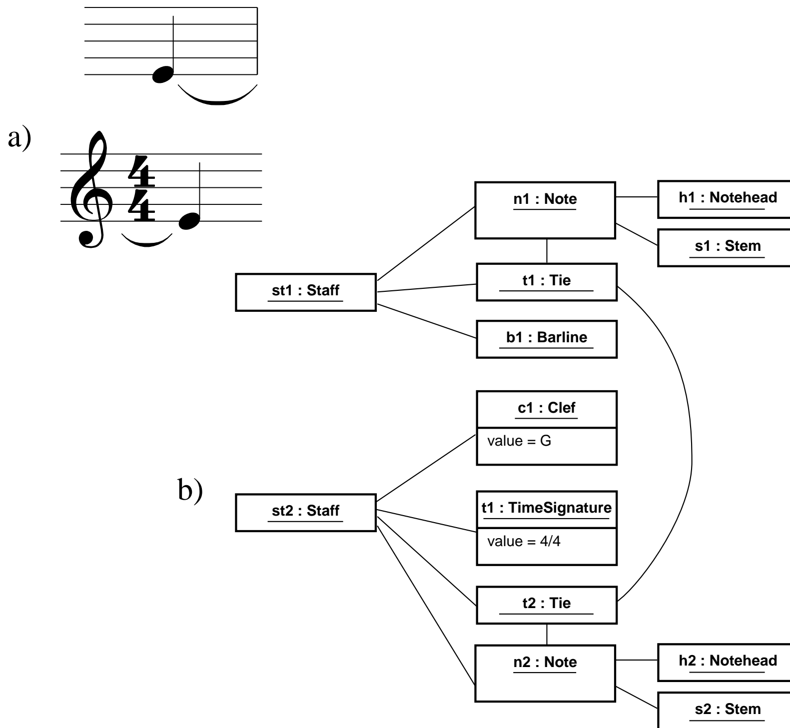


Figure 8-7: Examples of tied notes

8.7 An XML example

As an alternative object-level representation to UML diagrams, Listing 8-1 below presents an example of an object system encoded in XML format. The XML listing represents the note example of Figure 8-2 a), and is structurally equivalent to the object diagram in Figure 8-2 b). The XML listing is, however, more detailed in its use of object attributes. In the XML representation, each aggregation relationship is stored as an XML element, with start and end tags named according to class names from the object model.

The XML example presented below is simplified from a full-featured XML document. In particular, it does not contain a formal definition of the representation in the form of a Document Type Definition (DTD) (Eckstein 1999: 9-10) or a "DOCTYPE" reference to an external DTD. The XML example is, however, syntactically correct and "well-formed" according to general XML requirements (Eckstein 1999: 14-15).

The listing shows a hierarchical aggregation structure with the Score object sc1 on the top level, System sys1 on the next level, and Staff st1 on the next level. The Staff object, in turn, contains other objects, including StaffLines, Notes, Barlines, and LedgerLines. All objects contain attributes named "origin" and "size". They are represented with respectively named XML elements. Both origin and size contain two coordinate values, horizontal and vertical. Origin represents the position of the object within its aggregate's internal coordinate system. Size, in turn, represents the width and height of the object relative both to the size of the object's aggregate and to an assumed "default" size.

Some of the objects contain a "value" attribute. For example, in Notehead objects, value represents the type of note head. The SingleBarline object contains the attribute "dashed", which in the example is set to "no". The dimensions attribute of our analysis model was presented above in the description of the CMNSymbol class. In this case, it is considered to be calculated automatically. Hence, respective XML elements have not been included in the example listing.

The XML listing is presented here in order to clarify further the structure of the object model. The XML example should be considered as only one of many possible XML implementations of the analysis model.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<Score>
  <Page id="pg1">
     <Staff id="st1">
         <origin>0 0</origin><size>8 70</size>
         <StaffLine id="sl1">
           <origin>0 0</origin><size>1 1</size>
         </StaffLine>
         <StaffLine id="sl2">
           <origin>2 0</origin><size>1 1</size>
         </StaffLine>
         <StaffLine id="sl3">
           <origin>4 0</origin><size>1 1</size>
         </StaffLine>
         <StaffLine id="sl4">
```

```
    <origin>6 0</origin><size>1 1</size>
  </StaffLine>
  <StaffLine id="sl5">
    <origin>8 0</origin><size>1 1</size>
  </StaffLine>
  <Clef id="c1">
    <origin>1.5 2</origin><size>1 1</size>
    <value>G</value>
  </Clef>
  <TimeSignature id="t1">
    <origin>8.999 0</origin><size>1 1</size>
    <value>3/4</value>
  </TimeSignature>
  <Note id="n1">
    <origin>16.499 -2</origin><size>1 1</size>
    <Notehead id="h1">
      <origin>0 0</origin><size>1 1</size>
      <value>closed</value>
    </Notehead>
    <Stem id="s1">
      <origin>0 0</origin><size>1 1</size>
      <value>up</value>
    </Stem>
  </Note>
  <Note id="n2">
    <origin>26.678 0</origin><size>1 1</size>
    <Notehead id="h2">
      <origin>0 0</origin><size>1 1</size>
      <value>closed</value>
    </Notehead>
    <Stem id="s2">
      <origin>0 0</origin><size>1 1</size>
      <value>up</value>
    </Stem>
  </Note>
  <Note id="n3">
    <origin>36.875 2</origin><size>1 1</size>
    <Notehead id="h3">
      <origin>0 0</origin><size>1 1</size>
      <value>closed</value>
    </Notehead>
    <Stem id="s3">
      <origin>0 0</origin><size>1 1</size>
```

```
          <value>up</value>
        </Stem>
      </Note>
      <Barline>
        <origin>47.456 0</origin><size>1 1</size>
        <dotted>no</no>
      </Barline>
      <Note id="n4">
        <origin>50.788 2</origin><size>1 1</size>
        <Notehead id="h4">
          <origin>0 0</origin><size>1 1</size>
          <value>open</value>
        </Notehead>
        <Stem id="s4">
          <origin>0 0</origin><size>1 1</size>
          <direction>up</direction>
        </Stem>
        <AugmentationDot id="d1">
          <origin>2 1</origin><size>1 1</size>
        </AugmentationDot>
      </Note>
      <SingleBarline>
        <origin>70 0</origin><size>1 1</size>
        <dotted>no</no>
      </Barline>
      <LedgerLine noteid="n1">
        <origin>15.499 -2</origin><size>1 1</size>
      </LedgerLine>
      <LedgerLine noteid="n4">
        <origin>49.788 -2</origin><size>1 1</size>
      </LedgerLine>
    </Staff>
  </Page>
</Score>
```

Listing 8-1: A sample XML representation

Chapter 9
Discussion

This chapter further examines the decisions made in the analysis. Alternate solutions are also discussed and evaluated, as are some issues concerning design-stage modeling. Explored here is the possible simplification of the analysis model, so as to enhance consistency, and expansion of the model by the addition of new objects and attributes. Finally, questions concerning inclusion of logical and performance information are engaged, and examples of implementation solutions are described.

This chapter presents the model for a generic computer graphics system that is capable of free-form graphics representation. The model is similar to that of a program for general-purpose drawing or for presentation applications. Since music notation is a graphic system, a good notation program should be capable of representing and constructing free-form graphics, in at least a two-dimensional coordinate system. Therefore, the model developed here is based on a free-form, two-dimensional graphics system.

9.1 Commentary on the analysis

The vocabularies used for the analysis model were adapted from Gerou & Lusk (1996), Heussenstamm (1987), and Ross (1970). These detailed and systematic texts provided reliable and thorough descriptions of music notation. The use of additional textbooks might have led to the consideration of other concepts, as either potential classes or relationships. Had such concepts been considered, however, it is unlikely that the model would require major changes to its general architecture. I can thus be confident that my analysis model presents a relevant and reliable model of traditional Western music notation.

The analysis model is fairly liberal with regard to multiplicity requirements in aggregation and association relationships. This liberality was demonstrated in the top-level aggregation structure presented above, in Chapter 7.3. As defined in the general rules of the analysis model, all aggregations are considered one-to-any unless defined otherwise. Hence it follows, for example, that a Score may exist without any Pages. Furthermore, Pages may exist without any Systems,

and Systems are not required to have Staves. This situation, however, violates the general requirement that all objects must have a visual appearance: what makes a Score visible if it has no Pages? Moreover, how is a System visible if it has no Staves? This logical problem might be solved by the use of stricter multiplicity rules.

Such situations – for example a score without any pages – may be logically impossible in a printed document. Yet, in a computer implementation a score could indeed exist, and even be visible, without any pages; for example, a computer application might create an empty window whenever a new score is created. Therefore, a liberal policy was followed here, in solving problems that can be regarded as design- or implementation-dependent.

9.2 A low-level graphics system

As stated above, the aim of the analysis was not to model any specific kind of computer application. Since music notation is a graphic system, however, some example of a generic graphics representation system should be provided. Given the great amount of research done in the field of computer graphics and the amount of applications available, existing systems may well be used as a basis for this task. The following model is based on the architecture of the PostScript graphics model and the Adobe Illustrator drawing program. An alternative, yet in some ways similar, architecture is included in Diener's glyph-based Nutation program (Diener 1990). The graphics model presented here is not part of the analysis model itself. Rather, it should be regarded as a purely hypothetical design model.

Our example graphics model is centered around the class CMNSymbol. The CMNSymbol is divided into general-purpose, graphical subclasses as presented in Figure 9-1. As described in the analysis model, CMNSymbol includes properties common to all notation symbols, including their graphic position, dimensions, the ability to visualize (draw) themselves and to generate a sonic representation of themselves. Figure 9-1 shows only a subset of the properties of the CMNSymbol class as defined in the analysis model.

The structure of the graphics model is derived from the data representation scheme used in existing, general-purpose, vector-based computer graphics programs, especially Adobe Illustrator (Adobe 1993).

In Figure 9-1, the CMNSymbol is presented as a superclass having two subclasses: CompositeSymbol and PrimitiveSymbol. CompositeSymbol is a construction of CMNSymbols, which can be either other CompositeSymbols or PrimitiveSymbols. PrimitiveSymbol is either a Shape, Text, or ImportedGraphic. Shape is a construction of Paths, a basic element in PostScript and

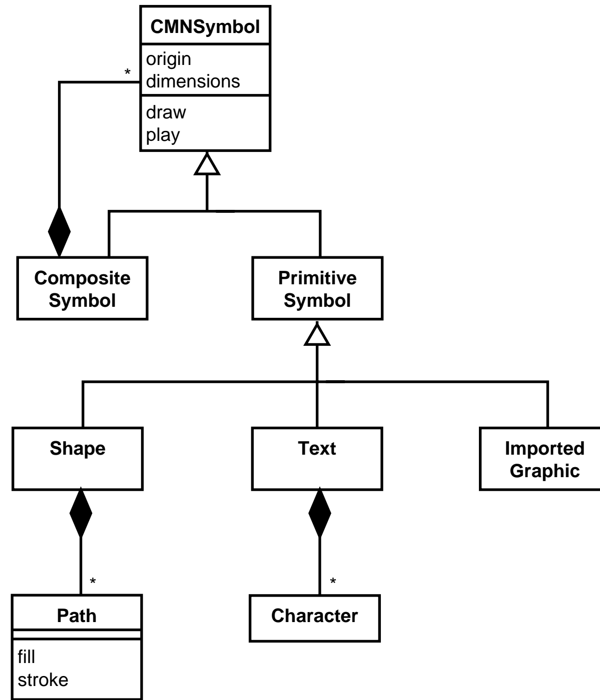


Figure 9-1: An example low-level graphics model

Adobe Illustrator. Path includes a set of coordinates that define the route through which a virtual (i.e., invisible) line or curve is drawn. When a path is “filled”, the area surrounded by this logical line or curve is filled with a given color. When a path is “stroked”, a visual line of a given width and color is drawn. Color and line width may be stored as attributes of Path or Shape.

Text consists of Characters. PostScript, for example, provides primitives for character shapes and basic text layout, such as the spacing between consecutive characters. ImportedGraphic is a symbol created with external software. ImportedGraphic is used in the analysis as a means of including symbols that are impossible or overly difficult to produce with the music notation program itself.

To implement the draw operation, a CompositeSymbol would invoke the draw operation of all of the objects it contains. If the object that receives the invocation call is a CompositeObject, it would in turn invoke its contents. Then a PrimitiveSymbol would execute appropriate graphics routines, such as PostScript commands, in order to draw the symbol on an output device.

Implementation of the play operation should follow the same basic principle of delegation, but, since most music notation symbols are CompositeSymbols, a sonic representation would be generated on a higher level in the symbol-aggregation hierarchy. The sections below describe the class structures of these higher-level objects.

9.3 Notes, stems and chords

As pointed out by Kurkela (1986: 101), pitch is not a parameter of a note symbol. In music notation, a note gains a pitch only when it is placed on a staff. This leads to an interesting question: should, as proposed in my object model, a Note "know" its position on a staff? that is, should it have coordinates that identify its position on a staff as its attributes? An alternative solution would be to store the position, and possibly size and dimensions, of each Note object as parts of its aggregate Staff, and somehow bind them to the Notes. This could be arranged in a generalized way by using a container class for all CMNSymbols, e.g., a class named "SymbolContainer".

This arrangement could yield a model that would be semantically closer to music notation. At the same time, it would also make the model more complex, not only by making the aggregation structure deeper, but also by making it more difficult to access Notes, because accessing would have to be done indirectly through the container object. In particular, this would make it more difficult for a Staff to manage its parts. Thus, the placement of the position and dimensions of a Symbol within the Symbol itself can be seen as a less tangled, more convenient solution.

As another possible modification to the analysis model, a generalization could be defined for some parts of Note; e.g., Arpeggio, Articulation, and Ornament. However, these classes differ, not only in shape, but also in their placement with respect to a note. Therefore, it is difficult to find a sufficient amount of common properties to justify a generalization.

The analysis model lacks a class for chords. Although "chord" is part of common musical vocabulary, its meaning is ambiguous and may be regarded as esthetic in nature. For example, in describing Palestrina-style counterpoint, it would anachronistic to call vertically aligned notes "chords", although they might appear visually as such in a printed score. Despite such difficulties, the

analysis model includes an association named "chord". The term chord was chosen because of its familiarity, despite the above-mentioned esthetic connotations. Association was chosen over aggregation structure in order to point out that chords are ambiguous, since they are not purely graphic constructs. According to the analysis model, Notes are always direct parts of a Staff. If an aggregation structure for chords had been included, Notes would have been defined as parts of either a Staff or a Chord. This would have led to an inconsistency like the one in the Page/System/Staff aggregation structure, for which an alternative design model is presented in Chapter 9.5.

9.4 Beams

Complex beam structures are difficult to represent in a computer program. This is demonstrated by how SCORE implements beams (SCORE 1992b: 69-88; SCORE 1993: 6.07). SCORE's beaming capabilities have become more highly developed in revisions of the program. This also bears upon the parametric structure, where new parameter fields and values have been added to enable more complex beaming arrangements.

For our analysis model, a deeper aggregation structure could be considered, such that a PrimaryBeam and SecondaryBeams would be parts of a container class. In Chapter 7, a shallow aggregation structure was chosen, because it is simpler (requires less classes) and because, in the analysis model, each class involved with beaming has an explicit visual representation.

Another alternative solution for organizing complex beam structures would be the inclusion of a container class for a group of beams, which could be called BeamGroup or BeamStructure. Eales included a class called BeamGroup in his analysis model (2000: 101). In our model, this class was not included because it is not part of common vocabulary of the problem domain. In a design model, however, BeamGroup might prove to be a relevant and useful class.

Another alternative would be to use Beams as aggregates of Notes and Rests. In light of Dannenberg's general view on hierarchy and structure (1993: 21), however, the use of an aggregation structure can complicate the model considerably if other, similar structures (such as slurs) are also treated as aggregations. Furthermore, the consistent use of such connector symbols as aggregates would require several interweaving aggregation structures. Such a method would eventually violate the principles of linear logic. Also, this procedure would blur the distinction between representation that is consistently graphically-oriented and representation that is consistently logically-oriented.

Yet another alternative could be to model Beams as parts of Notes. This would be consistent with how Notes relate to Stems. A possible way of model-

ing this relationship could be to define Beams as an association class for Notes or Stems. There, an association (called, say, “beamed”) could be defined between successive Notes or Stems, and the Beam symbol objects would be instances of the association class Beam. There, the Beam symbols would only exist when the respective association between Notes or Stems exists.

Conceptually, the modeling of Beams as parts of Notes constitutes the opposite to modeling Notes as parts of connector objects. For the present analysis model, a neutral relationship was chosen, such that a Beam is not part of a Note, but neither are Notes part of Beam structures. Instead, the next higher-level aggregate, Staff, is used as the container for both classes.

9.5 Design issues

The design stage often involves expansion of the analysis model with new classes and relationships required by a specific computer application. The designers should at the same time consider omitting unnecessary constructs from the analysis model. Moreover, ambiguities and overly complex structures should be questioned and simplified, if possible. One way to simplify the analysis model deals with the organization of the top-level aggregation structure. As shown in Figure 7.3, the analysis model defines a parallel aggregation relationship between Page and both System and Staff. Yet, a Staff can alternatively be part of a System. In music notation, it is permissible to have staves that do not belong to any system. Nonetheless, a structure of more consistency would be achieved, if the top-level one were modeled as shown in Figure 9-2.

In this simplification, the direct aggregation relationship between Staff and Page has been removed. As a result, each Staff is always part of a System, even if there is only one Staff per System. Although this arrangement might lack similitude to the problem domain, it would result in a more efficient computer implementation, because a Page would only be responsible for manipulating Systems and not both Systems and Staves. If this modification was made to the design, other relationships involving System and Staves would also have to be modified, in particular to allow (single-Staff) Systems to exist without a SystemicBarline.

An additional modification to the analysis model appears in Figure 9-2, where all aggregation structures are defined as ordered. As discussed in Chapter 7.3, although the parts of a symbol are already implicitly ordered by their position within their aggregate’s coordinate system, explicit ordering of logically one-dimensional structures (such as Systems within a Page, or Staves within a System) is likely to help in organizing the data. There would be disadvantages, for example, due to added redundancy of information; but these would likely be

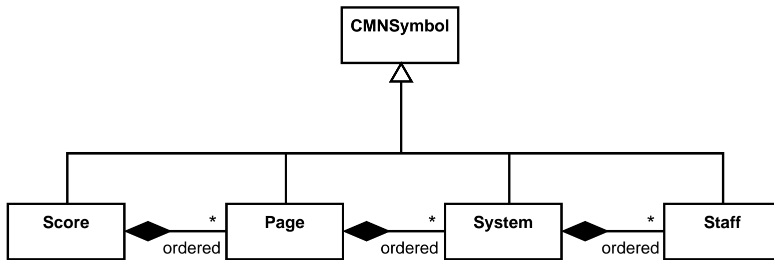


Figure 9-2: A simplified top-level aggregation structure

small as compared to the benefits. Therefore, the ordering of other aggregation structures should also be considered in the design stage.

Another simplification of the analysis model would involve the SystemicBarline class. The only difference between a SystemicBarline and a regular Barline is that the former is justified to the left edge of system staves, and it always spans the whole system. However, other system-wide barlines are modeled as regular Barlines. If the design model of Figure 9-2 is accepted, the SystemicBarline class could well be omitted from the object model. If needed, a generic specialized class for barlines connecting more than one Staff could be defined instead.

The complexities of automatic spacing are only partly helped by the object model itself. The inheritance hierarchy provides some aid in the making of spacing decisions. For example, CoreSymbol represents a generalization of symbols that have a higher precedence in spacing compared to Attachments, which are generally placed according to some CoreSymbol. The analysis model does not, however, make assumptions about which object is responsible for spacing.

If an initialization and data transmission protocol similar to that of Sound Processing Kit were used as the basis of a music-notation object system, the notation objects would allocate space for themselves and negotiate with nearby objects to determine their position on the staff, system, or page. This would enable implementation of a hierarchical, delegated, spacing procedure that starts

from the root aggregate and descends into the subparts of its lowest level. As mentioned in Chapter 2.7, however, Gerou and Lusk suggest that, to enhance readability, barlines on successive staves should be misaligned horizontally, unless they belong to the same system. This suggestion can be extended to apply to successive systems as well. This indicates that even a system cannot fully determine the spacing of its contents without checking the spacing of other, nearby systems. Therefore, spacing should be performed, at least partly, by an object belonging to a high level in the aggregation hierarchy. It was considered risky to make that or any other assumption about spacing procedure during the analysis phase. Instead, the analysis model makes it possible to use a variety of different spacing procedures – both automatic and manual.

9.6 Implementation issues

As noted above, the object model does not require the use of a linear-logic based programming language. Similarly, the analysis model is liberal with regard to the implementation programming language in general. In principle, any language that supports at least single inheritance, aggregation and association can be used to implement the model. If a linear-logic-based programming language is used, the implementation of associations is one of the main concerns.

In music notation, many symbols may expand beyond a single staff; for example, chords and beams. Therefore, associations in the object system are also expected to span more than one staff. Similarly, there may be cross-references between other types of objects belonging to different parts in the aggregation hierarchy.

This type of complex cross-referencing is common in conventional object-oriented programming languages based on reference semantics. For a genuinely linear-logic-based system, cross-referencing between different aggregation structures may be difficult or computationally expensive to implement. One implementation solution, which would respect the basic principles of linear logic, would be an object structure similar to the UNIX hierarchical file system (Ritchie & Thompson 1974), with the exception of multiple "hard links" to a file. There, aggregations would be implemented as file system directories (sometimes called "folders"). Associations would be implemented as "soft links" (UNIX 1989) or as "aliases", as in the Apple MacOS file system (Apple 1992). UNIX soft links allow links to files or directories to exist without the actual file or directory. Attempts to access a nonexistent file would be blocked by the file system.

9.7 Score processing and dynamic behavior

Although our focus has been on structural modeling, some principal issues concerning the dynamic behavior of the object model should also be discussed. This involves application of the analysis model for the performance of common tasks in computer-based notation.

Computer processing of music notation may be divided into five main tasks:

1. Data input and storage retrieval
2. Editing
3. Data storage
4. Printing
5. Playback

Each of the tasks involve some form of dynamic behavior. Data input and storage retrieval create new objects; e.g., by commands issued by a user or from a data input or storage representation. Editing involves making changes to an existing score, including the moving, copying, or deletion of objects. These tasks change the state of the object system. In Chapter 2.5, it was pointed out especially that editing, storage, and playback capabilities are not needed in every music notation application. At the least, however, data input causes dynamic activity in the system.

Data storage involves saving the state of an object system, for example, to a file. Printing and playback require the production of graphic and sonic representations of the score, respectively. Whereas storage, printing, and playback require iteration or interpretation of the score, the state of the object system representing the score does not necessarily change. There, dynamic behavior may involve creation and manipulation of temporary objects that are not part of the music-notation object system itself, but are used to store and transmit information from the notation objects during an iterative interpretation process.

As an example, a music playback algorithm would iterate through each Page in a Score, each System on a Page, each Staff in a System, each CoreSymbol and Attachment on a Staff, and so on. Each Symbol encountered in the iteration process may affect the consequent CMNSymbols. The effect of each CMNSymbol would be maintained through the iteration. In an object-oriented system, an obvious solution is to use an object for data storage. We shall call the class of this hypothetical object “PlaybackEnvironment”.

According to the analysis model, CMNSymbols implement an operation named “play”. This operation would be invoked for each CMNSymbol encountered during the playback process. The playback process would be started by invocation of the play operation of a Score object. The Score object would iter-

ate through its Pages and successively call their respective play operations. This process of delegating the play operation would continue down to the lowest-level objects in the aggregation hierarchy. A PlaybackEnvironment object would be passed along as a parameter of the playback operation and returned in an updated state by each invocation of “play”. The state of the PlaybackEnvironment would thus change according to the effect of each Symbol (such as changes in key or time signature, changes in dynamics, effects of accidentals, etc.). In a multi-staff system, a separate PlaybackEnvironment instance might be needed for each Staff. Note objects would react to play operations by producing sound events or an instruction for generating them. Most other types of Symbols would only alter the state of PlaybackEnvironment.

A similar iterative process would be needed in all the tasks listed above. In printing, the Score would invoke a “draw” operation on all of its parts, and each part would invoke the draw operation on their parts, etc. In this process another dynamic object, say, “GraphicalEnvironment”, could be used to pass on information about the position and scale of each Symbol to be drawn.

Environment objects could be also used as a way of expanding the capabilities of the object representation to cope with different musical styles. As pointed out by Selfridge-Field (1997c: 11-12), performance of a Western art music score requires knowledge of the conventions of the specific musical style and historical era of the composition and/or notation. For example, the correct interpretation of durational values in Baroque music scores may differ from those conventional to a present-day reader. In the object model, specialized PlaybackEnvironment objects could serve to represent the performance conventions of specific musical styles.

In interactive computer-based editing, a typical task is to find and select a symbol that a user clicks on with a mouse or other pointing device. If a single page of a score is visible, the search operation would be invoked on a Page object with the x and y coordinates of the pointing device as parameters for the operation. Using those coordinates, the Page object would invoke a search operation for its parts, and so on, until the object pointed at is found. When an editing operation is completed, adjustment of spacing may be necessary. Again, an automatic spacing algorithm would iterate, possibly in multiple passes, through each adjusted System and Staff.

In some editing tasks, the environment of symbols may also have to be taken into account. For example, if notes are moved or copied within a score, it may be necessary to preserve their pitches and durations so that they can be intelligently placed into a new environment; for example, into another key or transposed for another instrument. There are at least two ways to accomplish an “intelligent”, copy-/paste-style editing task: (1) The environment of the symbols (key signa-

ture, preceding accidentals, etc.) is carried with the copied notes; e.g., by means of a dedicated environmental object. (2) A temporary logically-oriented or hybrid representation is created by calculating the logical pitches and durations for each note. The temporary representation would be converted back into purely graphical representation when the notes are inserted into the new environment.

A simpler editing task is described below in more detail. Figure 9-3 shows a a sample UML sequence diagram (Booch *et al.* 1999: 245-247). It displays a hypothetical interaction process in which a user first selects a symbol then issues a command to delete it.

On the object level, interaction takes place between four objects of four classes: Page, System, Staff, and Note. It is assumed that the Page of the Score has already been selected. Therefore, a Score object is omitted from the diagram. The diagram shows a simplified two-step process, where one note symbol is first selected and then deleted from a staff. The diagram roughly simulates a situation in which a user selects a note on a page of a score (e.g. by clicking on the note symbol with a mouse) and then issues a "delete" command (e.g., presses the DELETE key on a keyboard).

In the diagram, the objects are shown on top. The program execution time flows from top to bottom. Communication between objects is shown with arrows. The name and arguments of the corresponding operation are attached to each arrow. An arrow drawn with a dashed line indicates the return of control to the caller of the operation. The execution time of an operation is indicated by a narrow box. Each object has a "life span" shown by a dashed vertical line. A cross symbol indicates the deletion of an object.

During the phase of symbol selection, a "selectSymbol" operation, with the coordinates of the pointing device as its parameters, is invoked on a Page object. The Page object compares the coordinates with the positions and dimensions of the System objects that it contains. When the appropriate System is found, Page calls the selectSymbol operation of that Symbol. This process continues until the appropriate CMNSymbol, in this case a Note, is found. Control is then returned to the user. During the deletion phase, the user invokes a "deleteSelection" operation of the Page object, which in turn iterates through its parts to delete all of the selected symbols that they contain.

The diagram is meant only to serve as an example of a hypothetical editing task. The exact need and behavior of algorithms depend on the application area of a particular computer program. The structure of the analysis model allows many different solutions for, say, selecting and deleting objects. Here, the use of the model and detailed forming of algorithms are regarded as design-stage issues.

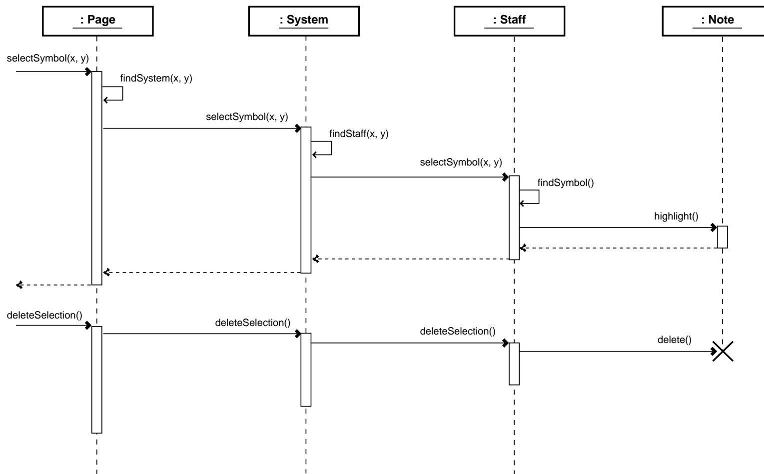


Figure 9-3: Sample sequence diagram: Deletion of a note from a staff

9.8 Logical and performance information

"Part" and "voice" are common concepts in musical representations. Both are supported in many notation-related representations, including NIFF and Tilia. Part and voice, if needed within a particular application, can be added as attributes of or associations with existing classes in the model. DurationalSymbol and CoreSymbol could be suitable generalizations for adding such features. Care should be taken, however, not to introduce unnecessary redundancy into the object system. In many cases, indication of part, and also voice, is contained in the graphical layout of the system.

Explicit encoding of logical information may be needed, especially if the notation program is designed to assist a user in composing or arranging music. There are also applications in which a graphic representation alone is sufficient. Therefore, representation of purely logical information can be regarded as a

design issue, where assumptions are made about the type of use(rs) of the notation program. In that case, the inclusion of purely logical objects may also be considered.

Performance information presents an additional challenge to the application of the analysis model. Addition of performance-related properties or relationships to the analysis model would enable implementation of a simple, “proofreading” kind of playback system. Sophisticated performance systems, and especially support for multiple, alternate performances of a single score, would require a more complex design.

One solution, one that keeps the notation object model mostly intact, would be to use a separate, performance-oriented representation for performance information. In the representation, performance(s) and notation could be linked to or synchronized with the notation system by the defining of associations between the two representations. If none of the existing performance-oriented representations are applicable to this purpose, a new analysis task should be undertaken; it would be based primarily on representation of performance information.

9.9 Macro statements

The “simile” articulation is a good example of problems involved in transforming graphic data into logical and performance information. Simile is a “macro” statement, which indicates a repetition of a group of previous articulations. Repeat signs and multiple rests are other examples of macro statements in music notation (see Heussenstamm 1987: 132-134). Another example is a vertically expanded, or “tall”, time-signature that spans several staves (ibid.: 118).

Compared to macro statements in computer programming languages, simile and repeat signs can be described as implicitly defined macros. They instruct the reader to repeat a previously represented gesture or pattern. In computer programming languages, such as C (Kernighan & Ritchie 1978), macros are defined explicitly before use. In a computer-processed representation, implicit macros cause similar difficulties in automatic spacing.

When performance data are generated by parsing a graphically-oriented representation, such as my analysis model, a simple one-pass iteration process would require all possible groupings of articulations to be remembered, in case a simile statement should appear. As a solution, a two-pass iteration process could be used, which would search for simile statements in the first pass and to generate performance in the second pass, where appearances of simile can be detected. Another solution would be to parse music backwards (i.e., from right to left) when a simile or repeat sign is encountered.

In an object system, associations could model the relationships between the "simile" symbol and the preceding symbols to which it refers. Alternatively, the relationships could be resolved when a performance is generated through automated interpretation of the score. The latter procedure must be used if the analysis model is implemented without extensions, because simile is modeled there as a regular articulation and does not include associations with other, preceding articulations.

9.10 Extendibility of the model

As mentioned above, music notation is an evolutionary system. Composers and music publishers occasionally need to expand the vocabulary of music notation for new applications. Therefore, a computer representation of music notation should also be extendable, which numbers among the properties of good representations listed by Huron (1992). My analysis model is extendable through inheritation; that is, it provides both concrete classes and abstract superclasses as bases for new, specialized subclasses. Also, attributes may be added to existing classes so as to extend the model's capabilities.

The model itself does not guarantee that a computer program that implements it is extendable. Typically, class-based, static programming languages do not contain built-in means of defining subclasses at runtime. Therefore, extra effort would be required in the design and implementation stages, to provide a means for end-users to define new classes (or new types of objects) according to a prototype-like principle. This kind of capability – which would enable users not only to design new graphical shapes, but also to give those shapes a musical function – should be a basic requirement for a computer program that attempts to emulate the flexibility of music notation.

9.11 Representational aspects

The analysis model can also be assessed according to Huron's list of properties for good representations. Some qualities of the analysis model are common, general object-oriented features. Others are products of the influence of linear logic. Some qualities are, in turn, products of specific decisions made during the analysis process. Many of the qualities apply both to the model and to music notation. There follows a brief description of the analysis model from the perspective of Huron's requirements.

The model is unique; i.e., no two signifieds share the same signifier. Uniqueness is achieved not only through classification and inheritation relationships,

but also through the strictly linear aggregation structure which guarantees that an object is at all times a direct part of one and only one other object.

The model is literally mnemonic. Class and property names are descriptive and typically not abbreviated. In being graphically-oriented and totally literal, the model is also consistent. Some of its details, however, can be regarded as inconsistencies. For example, Beams are modeled as part of Staves, whereas Stems are modeled as part of Notes (even though several notes can share a single Stem). At the same time, Beams were represented according to principles similar to those which define symbols that connect objects horizontally (or in time), such as Slurs or Ties. In this respect, consistency was achieved.

The object representation is, at least in principle, totally reversible. This should apply to all well-formed object representations of any target. The object model is not economical. This is the trade-off for literalism and descriptiveness. The descriptive names make the model verbose, which, viewed positively, guarantees that the model is not cryptic.

Structurally, the model is not particularly isomorphic. Better and more explicit isomorphism could be achieved by the addition of more features (e.g., ordering) to aggregation and association relationships. Here, structural isomorphism has been considered primarily as a design issue.

As music notation itself, the object model is highly context-dependent. On the other hand, the model is highly explicit in representing the graphical symbols of music notation. The model is not optional (as "optional" is explained by Huron), because all objects and all of their attributes are assumed to be represented explicitly.

As mentioned above, our model is extendable through inheritation. Superclasses, both abstract and concrete, are used in the model having features in common with their subclasses. They also serve as "mounting points" on which to add new classes if the model needs to be extended. For example, if a new type of time symbol should be needed – one that in some way differs from a rest or note – then it could be added as a new subclass of DurationalSymbol. Similarly, the model might be extended to support Schenkerian analysis; for example, a new subclass for Arc could be defined with the name "SchenkerianProgression". This class would have a similar shape but a function different from Slur or Tie.

The properties just discussed characterize how the analysis model represents the signifiers of music notation. The properties of music notation itself as a representation were considered in Chapter 3.3. Theoretically, for example, the analysis model can be a consistent representation of a representation that is itself inconsistent. Some properties of music notation, such as context-dependency, apply to the analysis model as well.

Chapter 10
Conclusions

Object-oriented analysis can serve as a tool not only for software design, but for theoretical examination of a complex system. Although the main intention of this study was to provide a representational basis for developing music notation software, the analysis model can also be regarded in some ways as a theoretical study of the structural relationships of music notation symbols. A central constraint placed on the model was that it be consistent, while respecting the vocabulary and behavior of the problem domain.

The use of a rule set based on linear-logic aids in the making of consistent decisions during the analysis stage. Linear logic can, however, be criticized for imposing a stiff and inflexible object structure. Also, linear logic itself has not yet gained wide acceptance in software engineering. For this study, an additional systematic method was needed, so that analytical decisions could be tested against generic and predefined principles, rather than every decision being treated as a special case. For this purpose, linear logic provided an efficient tool. Moreover, it was not presupposed that a programming language based on linear logic would be required for implementing the model.

The graphic notation languages of object-oriented methods serve as a convenient and compact tool with which to analyze systems and design software. UML, in particular, is so widely accepted that educated software designers and programmers can be expected to produce analyses and designs even without including explanations of the notation conventions. For this study, only a subset of UML was needed. A description of the subset was included in order to make the text methodologically self-contained.

The key features of the analysis model can be summarized as follows:

1. It presents an object-oriented representation of the problem domain
2. It uses terminology that can be found in the established vocabulary of music notation
3. It is consistently graphically-oriented
4. It uses a hierarchical class inheritation structure for categorizing notation symbols

5. It defines a systematic and coherent aggregation structure influenced by linear logic
6. It suggests that logical information should be modeled as object properties and associations, not as objects
7. It puts little emphasis on explicit representation of purely performance-related information

The object-oriented representation is described with a set of UML class diagrams. The latter show various types of relationships between classes and some of their central properties. The analysis model is a static, structural representation. Dynamic behavior of the class system is not described except for the few design suggestions and examples given in Chapter 9. The analysis model is not a complete and sufficient representation to be used as specification for a programming task. An additional design task is needed to derive a usable specification. The analysis model presents a basic class structure upon which a practical computer representation can be designed.

The analysis model employs terms commonly found in textbooks describing music notation and its uses. This should make the representation understandable to a musically educated user. Departing from traditional terminology, newly-coined descriptive names were given to some superclasses. These invented names describe the common role or function of their subclasses (e.g., DurationalSymbol, Attachment, Connector), and names of some specialized subclasses were derived from the names of their superclasses (e.g., PrimaryBeam, SecondaryBeam, SingleBarline).

Because it is graphically-oriented, the analysis model is a relatively low-level, iconic representation of music notation. The signifiers of the object model attempt to represent, through one-to-one mapping, the respective signifiers of music notation. This is in contrast with the many logically-oriented representations of music notation that represent certain signifieds of their source representation.

The hierarchical class inheritation structure demonstrates the similarities among related objects through the use of superclasses. The superclasses also serve as "mounting points" for extending the representation with additional classes. The class hierarchy itself is static, as are class-based object systems in general. Examples were nevertheless provided to show how the class structure might be modified in the software design stage.

The rule set based on linear logic-based proved especially helpful in the analysis of "part-of" relationships. Linear logic also helped indirectly, in systemizing the modeling of inheritation structures. Nevertheless, many analytic

situations still had to be defined and argued on an individual basis; simple, unambiguous solutions were not always found.

The principle of modeling logical constructs as properties and associations, instead of as objects and aggregations, is a direct consequence of the consistently followed graphics-orientation of the analysis. The analysis model represents logical constructs in a manner similar to the way they are represented by music notation, such that logical information is implied by visual symbols. The defining of logical associations between visual objects enabled logical information to be expressed explicitly and without the graphic orientation of the representation being compromised.

The analysis model puts little emphasis on the representation of performance-related information. This is a deliberate decision. With regard to music notation, performance information is regarded as external data that can be derived through interpretative processes, but that should not be considered an integral or mandatory property of music notation itself.

Of the existing representations described in this study, the analysis model most closely resembles the SCORE parameter list. Thus, it could be argued that the deficiencies of SCORE also apply to the analysis model. On the other hand, it can also be argued that, in SCORE, good representation was attained despite the primitive data-structure capabilities of its implementation programming language. It can be argued further that many representational details of SCORE can be successfully adapted to object-oriented representations.

There are also ways in which the analysis model departs significantly from the SCORE parameter list. In particular, the hierarchical aggregation structure of the analysis model is deep, and the parts of objects are themselves full-featured objects. Also, the class inheritation structure of the analysis model is hierarchical and contains named, abstract classes that represent the common properties of their subclasses. Aggregation-like and inheritation-like structures can also be found on the SCORE parameter list, but there they are coded less consistently and less systematically than they are in my model. Moreover, the analysis model makes use of explicit associations between symbols, an operation which SCORE does not support.

It is in no way claimed that the analysis model is the only possible, or even best possible, object-model of music notation – even if only graphics-oriented representation is concerned. Rather, the main asset of this study lies in the methodological principles and the representational approach to music notation upon which the analysis is based. One conclusion of this study is that music notation is truly a complex system and thus cannot be analyzed with a simple model. Nevertheless, modeling was required, not only to recognize that fact, but also to

see which problems of music notation can be solved elegantly by generalization and which ones require special, complicated solutions.

Music notation is itself a representation. Therefore, a computer representation of music notation can be regarded as a meta-representation. A fundamental analytic decision for this study was the choice between representing either the signifiers or the signifieds of music notation. A mixture of both types would have resulted in a hybrid, logical/graphic representation, wherein consistency and explicitness are difficult or impossible to achieve.

Although a graphic orientation was chosen as the basis for the analysis model, it must be acknowledged that graphically-oriented representation is not optimal or sufficiently explicit for all musical uses. Still, because explicit visual evidence of the signifieds is represented graphically, it can be concluded that this type of representation rests on firm and objective conceptual bases.

References

Adobe 1986: *PostScript Language Reference Manual.* Reading, Massachusetts: Addison-Wesley.

— 1993: *Adobe Illustrator 5.5 User Guide*. California: Adobe Systems Incorporated.

— 1994: *Tutorial for the Adobe Illustrator Plug-in Architecture for version 5.0 & 5.5*, California: Adobe Systems Incorporated.

Apple Computer Inc. 1992: *Inside Macintosh: Files.* Reading, Massachusetts: Addison-Wesley.

Andreoli, Jean-Marc – Pareschi, Remo 1990: LO and Behold! Concurrent structured processes. *ACM SIGPLAN Notices, Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications.* Volume 25, Issue 10.

Asperti, Andrea, Roversi, Luca 2002: Intuitionistic Light Affine Logic. *ACM Transactions on Computational Logic* (TOCL). Volume 3, Issue 1.

Baker, Henry 1992: Lively Linear Lisp – 'Look Ma, No Garbage'. ACM *Sigplan Notices* 27,8.

— 1994a: Linear Logic and Permutation Stacks – The Forth Shall Be First. ACM Sigarch *Computer Architecture News* 22,1.

— 1994b: A "Linear Logic" Quicksort. ACM *Sigplan Notices* 29,2.

— 1994c: 'Use-Once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading. ACM *Sigplan Notices.* September 1994.

Belkin, Alan 1994: Macintosh Notation Software: Present and Future. *Computer Music Journal*. Volume 18, Number 1.

Bellini, P. Nesi, P. 2001: WEDELMUSIC Format: An XML Music Notation Format for Emerging Applications. *First International Conference on WEB Delivering of Music (WEDELMUSIC'01).* Florence, Italy: IEEE.

Bellini, P. – Nesi, P. – Spinu, M. B. 2002: Cooperative Visual Manipulation of Music Notation. ACM Transactions on Computer-Human Interaction (TOCHI). Volume 9, Issue 3.

Blostein, Dorothea – Haken, Lippold 1991: Justification of Printed Music. *Communications of the ACM*. Volume 34, Number 3.

Booch, Grady 1994: *Object-Oriented Analysis and Design with Applications*. Second Edition. Santa Clara, California: Benjamin/Cummings.

Booch, Grady – Rumbaugh, James – Jacobson, Ivar 1999: *The Unified Modeling Language User Guide*. Second Edition. Santa Clara, California: Benjamin/Cummings.

Boulanger, Richard (ed.) 2000: *The Cound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming.* Cambridge, Massachusetts: The MIT Press.

Boynton, Lee – Jaffe, David 1991: An Overview of the Sound and Music Kits for the NeXT Computer. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* S. T. Pope (ed.). Cambridge, Massachusetts: The MIT Press.

Branagan, Linda – Serra, Michael 1994: *The Frame Handbook – Building FrameMaker Documents That Work*. Sebastopol, California: O'Reilly & Associates.

Byrd, Donald 1984: *Music Notation by Computer.* PhD thesis. Ann Arbor: Indiana University.

Byrd, Donald 1994: Music Notation and Intelligence. *Computer Music Journal*. Volume 18, Number 1.

Castan, Gerd 2001 [1999-2000]: NIFFML: An XML Implementation of the Notation Interchange File Format. *The Virtual Score – Representation, Restrival, Restoration.* Walter B. Hewlett, Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Coad, Peter – Yourdon, Edward 1991a: *Object-Oriented Analysis*. Second Edition. Englewood Cliffs, New Jersey: Prentice-Hall.

— 1991b: *Object-Oriented Design*. Englewood Cliffs, New Jersey: Prentice-Hall.

Coad, Peter – Nicola, Jill 1991: *Object-Oriented Programming*. Englewood Cliffs, New Jersey: Prentice-Hall.

Cole, Hugo 1974: *Sounds and Signs.* London: Oxford University Press.

Cooper, David – Ng, Kia-Chuan – Boyle, Roger D. 1997: MIDI Extensions for Music Notation (2): Expressive MIDI. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Cox, Brad J. – Novobilski, Andrew J. 1991: *Object-Oriented Programming: An Evolutionary Approach*. Reading, Massachusetts: Addison-Wesley.

Crosvenor, Jonathan – Morrison, Kay – Pim, Alexandra (ed.) 1992: *The PostScript Font Handbook*. Revised Edition. Wokingham, Berkshire: Addison-Wesley.

Cycling74 2004: Max/MSP. http://www.cycling74.com/products/maxmsp.html (Accessed 2004-03-10).

Dannenberg, Roger 1993: Music Representation Issues, Techniques, and Systems. *Computer Music Journal*. Volume 17, Number 3.

Déchelle, Francois 1999: jMax: an environment for Real-Time Musical Applications. *Computer Music Journal.* Volume 23, Number 3.

— 2004: A Brief History of MAX. http://freesoftware.ircam.fr/article.php3?id_article=5 (accessed 2004-03-07).

Diener, Glendon Ross 1990: *Modeling Music Notation: A Three Dimensional Approach*. PhD thesis. Stanford University.

Dodge, Charles – Jerse, Thomas A. 1985: *Computer Music – Synthesis, Composition, and Performance.* New York: Schirmer Books.

Eales, Andrew 2000: The Music Notation Toolkit: A Study in Object-Oriented Development. *Proceedings of the NACCQ 2000.* http://www.naccq.ac.nz/conference01.html?page=13 (accessed 2004-01-12).

Eckstein, Robert 1999: *XML Pocket Reference.* Sebastopol, California: O'Reilly.

Foley, James D. – van Dam, Andries – Feiner, Steven K. – Hughes, John F. 1992: *Computer Graphics – Principles and Practice*. Second edition. Reading, Massachusetts: Addison-Wesley.

Foxley, Eric 1987: Music – A Language for Typesetting Music Scores. *Software - Practice and Experience* 17(8): 485-502.

Forsberg, Andrew – Mark Dieterich, Mark – Zeleznik, Robert 1998: The music notepad. *Proceedings of the 11th annual ACM symposium on User interface software and technology.* New York: ACM Press.

Fraunhofer 2004a: Audio & Multimedia. MPEG Audio Layer-3. http://www.iis.fraunhofer.de/amm/techinf/layer3/index.html (accessed 2004-03-07).

— 2004b: Audio & Multimedia. MPEG-4 Overview. http://www.iis.fraunhofer.de/amm/techinf/mpeg4/index.html (accessed 2004-03-07).

Gerou, Tom – Lusk, Linda 1996: *Essential Dictionary of Music Notation.* Los Angeles: Alfred Publishing Co.

Girard, Jean-Yves 1987: Linear Logic. *Theoritical Computer Science* 50.

— 1995: Linear Logic: Its Syntax and Semantics. *Proceedings of the Workshop on Advances in Linear Logic.* J.-I. Girard, Y. Lafond, L. Regnier (ed.) Ithaca, New York: Cambridge University Press.

— 1998: Light Linear Logic. *Information and Computation.* Volume 143, Issue 2.

Goldberg, Adele – Robson, David 1989: *Smalltalk 80: The Language.* Reading, Massachusetts: Addison-Wesley.

Good, Michael 2001 [1999-2000]: MusicXML for Notation and Analysis. *The Virtual Score – Representation, Restrival, Restoration.* Walter B. Hewlett, Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Goodman, Nelson 1985: *Languages of Art – And Approach to a Theory of Symbols*. Second Edition.Indianapolis: Hackett Publishing Company.

Gourlay, John S. 1987: Spacing a line of music. *Technical Report* OSU-CISRC-10/87-TR35, Department of Computer and Information Science. The Ohio State University.

Grande, Cindy – Belkin, Alan 1996: The Development of the Notation Interchange File Format. *Computer Music Journal*. Volume 20, Number 4.

Grande, Cindy 1997: The Notation Interchange File Format: A Windows-Compliant Approach. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Grove 1980: Notation. *The New Grove Dictionary of Music and Musicians.* Stanley Sadie (ed.). London: Macmillan Publishers Limited.

Haken, Lippold – Blostein, Dorothea 1993: The Tilia Music Representation: Extensibility, Abstraction, and Notation Contexts for the Lime Music Editor. *Computer Music Journal*. Volume 17, Number 3.

Heussenstamm, George 1987: *The Norton Manual of Music Notation.* New York: W. W. Norton & Company.
Hewlett, Walter B. – Selfridge-Field, Eleanor 1987: *A Directory of Computer Assisted Research in musicology.* Menlo Park, California: Center for Computer Assisted Research in the Humanities.
— (ed.) 1989: *Computing in Musicology – A Directory of Research*. Menlo Park, California: Center for Computer Assisted Research in the Humanities.
Hewlett, Walter B. – Selfridge-Field, Eleanor (with David Cooper, Brent A. Field, Kia-Chuan Ng, and Peer Sitter) 1997: MIDI. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.
Huron, David 1992: Design Priciples in Computer-Based Music Representations. *Computer Representations and Models in Music.* Alan Marsden & Anthony Pople (ed.). London: Academic Press.
Jacobson, Ivar 1992: *Object-Oriented Software Engineering: A Use Case Driven Approach* . Reading, Massachusetts: Addison-Wesley.
Jacobson, Ivar – Booch, Grady – Rumbaugh, James 1999: *The Unified Development Process.* Reading, Massachusetts: Addison-Wesley.
Kippen, Jim – Bel, Bernard 1992: Modelling Music with Grammars – Formal Language Representation in the Bol Processor. *Computer Representations and Models in Music.* Alan Marsden & Anthony Pople (ed.). London: Academic Press.
Kervinen, Jukka-Pekka – Lassfolk, Kai 1993: Helsinki Music Tools, *Proceedings of the 1993 International Computer music Conference*, Tokyo: The International Computer Music Association.
Kernighan, B. W. – Lesk, M.E. – Ossanna, J. F. Jr. 1987 [1978]: Document Preparation. *UNIX System Readings and Applications Volume I. UNIX Time-Sharing System.* AT&T Bell Laboratories. Englewood Cliffs, New Jersey: Prentice-Hall.
Kernighan, Brian W. – Pike, Rob 1984: *The UNIX Programming environment.* Englewood Cliffs, New Jersey: Prentice-Hall.
— 1999: *The Practice of Programming.* Reading, Massachusetts: Addison-Wesley.
Kernighan, Brian W. – Ritchie, Dennis M. 1978: *The C Programming Language.* Englewood Cliffs, New Jersey: Prentice-Hall.
Klapuri, Anssi 2003: Automatic Transcription of Music. *Proceedings of the Stockholm Music Acoustics Conference,* August 6-9, 2003 (SMAC 03). Stockholm. http://www.cs.tut.fi/sgn/arg/klap/smac2003_klapuri.pdf. (Accessed 2004-05-21.)
Krasner, Glenn 1991: The Design of a Smalltalk Music System. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* S. T. Pope (ed.). Cambridge, Massachusetts: The MIT Press.
Kurkela, Kari 1986: *Note and Tone – A semantic analysis of conventional music notation*. Helsinki: Suomen Musiikkitieteellinen Seura.
Kuuskankare, Mika – Laurson, Mikael 2003: ENP-Expressions, Score-BPF as a Case Study. *Proceedings of International Computer Music Conference*. Singapore: The International Computer Music Association.

Lassfolk, Kai – Lehtinen, Timo 1988: *The Time Stamped Music File Format Standard*. ANSI Document X3V1.8M/88-7. San Francisco, California: International Computer Music Association.

Lassfolk, Kai 1995: Sound Processing Kit. *Proceedings of the 1995 International Computer music Conference*. Banff: The International Computer Music Association.

— 1999: Sound Processing Kit – An Object-Oriented Signal Processing Framework, *Proceedings of the 1999 International Computer music Conference*. Beijing: The International Computer Music Association.

Laurson, Mikael 1996: *PATCHWORK: A Visual Programming Language and some Musical Applications.* Doctoral Thesis. Studia Musica No.6, Helsinki: Sibelius Academy.

Lazzaro, John – Wawrzynek, John 2004: MPEG-4 Structured Audio: Developer Tools. http://www.cs.berkeley.edu/~lazzaro/sa/ (accessed 2004-03-07).

Leppard, Raymond 1988: *Authenticity in Music.* London: Faber Music.

Loy, Gareth 1989: Composing with Computers – A Survey of Some Compositional formalisms and Music Programming Languages, *Current Directions in Computer Music Research*, Max V. Mathews – John R. Pierce (ed.). Cambridge, Massachusetts: The MIT Press.

Mathews, Max 1969: *The Technology of Computer Music*. Cambridge, Massachusetts: the MIT Press.

Maxwell, J. – Ornstein, S. 1984: Mockingbird: A Composer's Amanuensis. *BYTE*, January 1984.

McNab, Rodger J. – Smith, Loyd A. – Witten, Ian H. – Henderson, Clare L. – Cunningham, Sally Jo 1996: Towards the Digital Music Library: Tune Retrieval from Acoustic Input. *Proceedings of the first ACM international conference on Digital libraries.* New York: ACM Press.

Microsoft 1991: *Microsoft Windows Multimedia Programmer's Reference.* Redmond, WA: Microsoft Press.

MIDI 1985 [1983]: MIDI 1.0 Detailed Specification. Date: August 5, 1983. *Synthesizers and Computers.* Tom Darter (ed.). Cupertino, California: GPI Publications, Hal Leonard.

— 1988: *MIDI 1.0 Detailed Specification*, Document version 4.1. Los Angeles, California: The International MIDI Association.

MOODS 2004: http://www.dsi.unifi.it/~moods/. (Checked 2004-03-01.)

NIFF 2002: *NIFF 6b – Notation Interchange File Format.* June 11, 2002. http://www.musique.umontreal.ca/personnel/Belkin/N/NIFF6b.html. (Checked 2002-11-27.)

Nordli, Kjell E. 1997: MIDI Extensions for Music Notation (1): NoTAMIDI Meta-Events. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Oppenheim, Alan V. – Schaefer, Ronald W. 1975: *Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall.

Ornstein, Severo M. – Maxwell, Jonh T. III 1983: *Mockingbird: A Composer's Amanuensis.* CSL-83-2. Palo Alto: Xerox Palo Alto Research Center.

Pope, Stephen Travis 1991a: Object-Oriented Software Design. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* S. T. Pope (ed.). Cambridge, Massachusetts: The MIT Press.

— 1991b: Introduction to MODE: The Musical Object Development Environment. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* S. T. Pope (ed.). Cambridge, Massachusetts: The MIT Press.

Read, Gardner 1982: *Music Notation – A Manual of Modern Practice*. London: Victor Collancz Ltd.

Renz, Kai 2000: Design and Implementation of a Platform Independent GUIDO Notation Engine. *Proceedings of the 2000 International Computer music Conference*. Berlin: The International Computer Music Association.

— 2002: An Improved Algorithm for Spacing a Line of Music. *Proceedings of the 2002 International Computer music Conference*. Göteborg: The International Computer Music Association.

Ritchie, Dennis M. – Thompson, Ken 1974: The UNIX Time Sharing System. *Communications of the ACM.* Volume 17, Issue 7.

Roads, Curtis 1985: Grammars as Representations for Music. *Foundations of Computer Music.* Curtis Roads – John Strawn (ed.). Cambridge, Massachusetts: The MIT Press.

— 1996: *The Computer Music Tutorial*. Cambridge, Massachusetts: The MIT Press.

Roland, Perry 2001: MDL and MusiCat: An XML Approach to Musical Data and Meta-Data. *The Virtual Score – Representation, Restrival, Restoration.* Walter B. Hewlett, Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Rouch, Dean K. 1988: Music Formatting guidelines. *Technical Report* OSU-3/88-TR-10. Department of Computer and Information Science, The Ohio State University.

Ross, Ted 1970: *The Art of Music Engraving and Processing*. Miami Beach, Florida: Hansen Books.

Rowe, Robert 1993: *Interactive Music Systems – Machine Listening and Composing*. Cambridge, Massachusetts: The MIT Press.

Rumbaugh, James – Blaha, Michael – Premerlani, William – Eddy, Frederick – Lorensen, William 1991: *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall.

SCORE 1992a: *SCORE User's Guide & Draw Manual*. Palo Alto, California: San Andreas Press.

— 1992b: *SCORE Reference Manual*. Palo Alto, California: San Andreas Press.

— 1993: *SCORE Version 3.10 Manual Additions*. Palo Alto, California: San Andreas Press.

Sebeok, Thomas A. 1975: Six species of signs: Some propoositions and strictures. *Semiotica* 13 (3).

Selfridge-Field, Eleanor 1997a: DARMS, Its Dialects, and Its Uses. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

— 1997b: Beyond Codes: Issues in Musical Representation. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

— 1997c: Describing Musical Information. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Schottstaedt, Bill 1997: Common Music Notation. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Slaer, Sally – Mellor, Stephen J. 1988: *Object-Oriented Systems Analysis: Modeling the World in Data.* Englewood Cliffs, New Jersey: Yourdon Press.

Sloan, Donald 1997: HyTime and Standard Music Description Language: A Document-Description Approach. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Smalley, Denis 1986: Spectro-morphology and Structuring Processes. *The Language of Electroacoustic Music*. Simon Emmerson (ed.). London: MacMillan Press.

Steele, Guy L. Jr. 1990: *Common LISP: The Language,* Second edition. Bedford, Massachusetts: Digital Press.

Sterling, Leon – Shapiro, Ehud 1986: *The Art of Prolog – Advanced Programming Techniques.* Cambridge, Massachusetts: The MIT Press.

Stickney, Kimball P. 1987: Computer Tools for Engraving-Quality Music Notation. *The Proceedings of the AES 5th International Conference, Music and Digital Technology.* Audio Engineering Society.

Smith, Leland 1973: Editing and Printing Music by Computer, *Journal of Music Theory* 17(2).

— 1997: SCORE. *Beyond MIDI – The Handbook of Musical Codes.* Eleanor Selfridge-Field (ed.). Cambridge, Massachusetts: The MIT Press.

Stickney, Kimball P. 1987: Computer Tools for Engraving Quality Music Notation. *The Proceedings of the AES 5th International Conference, Music and Digital Technology.* The Audio Engineering Society.

Stone, Kurt 1980: *Music Notation in the Twentieth Century – A Practical Guidebook*. New York – London: W. W. Norton & Company.

Stroustrup, Bjarne 1991: *The C++ Programming Language*, Second Edition. Reading, Massachusetts: Addison-Wesley.

— 1994: *The Design and Evolution of C++*. Second Edition. Reading, Massachusetts: Addison-Wesley.

UNIX 1989: LN(1). *UNIX Programmer's Manual.* More/bsd Version. Chapters 1, 8 and 7. May 1989. Berkeley, California: MT XINU.

Ungar, David – Smith, Randall B. 1991: SELF: The Power of Simplicity. *LISP and Symbolic Computation: An International Journal*, 4,3, 1999. The Netherlands: Kluwer Academic Publishers.

Wadler, Philip 1991: Is there a use for linear logic? *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM).* New Haven, Connecticut: ACM Press.

Winikoff, M. – Harland, J. 1996: Some Applications of the Linear Programming Language Lygon. *Australian Computer Science Communications,* 18(1), Kotagiri Romamohanarao (ed.).