

# An Object-Oriented Toolkit for Music Notation

THESIS

Submitted in Fulfilment of the  
requirements for the Degree of

**MASTER OF SCIENCE (APPLIED COMPUTER SCIENCE)**

by

**Andrew Arnold Eales**

November

1998

Supervisor : Dr. R.J. Foss

# Abstract

This thesis investigates the design and implementation of an object-oriented toolkit for music notation. It considers whether object-oriented technology provides features that are desirable for representing music notation. The ability to sympathetically represent the conventions of music notation provides software tools that are flexible to use, and easily extended to represent less common features of music notation.

The design and implementation of an object-oriented class hierarchy that captures the structural and semantic relationships of music notation symbols is described. Functions that search for symbols, and update symbol positions are also implemented. Traditional context-sensitive and spatial relationships between music symbols may be maintained, or extended to provide notational features found in modern music. MIDI functionality includes the ability to play music notation and to allow step-recording of MIDI events.

The toolkit has been designed to simplify the creation of applications that make use of music notation; example applications are created to demonstrate its capabilities.

**Keywords :** object-oriented, music notation, computer music.

# Table of Contents

## 1. Introduction

1.1 Computers and Music Notation .....	1
1.2 Object-Oriented Software Design .....	1
1.3 Designing Music Notation Software .....	2
1.4 Music Notation and MIDI .....	2
1.5 Project Objectives .....	3

## 2. An Overview of Music Notation

2.1 Traditional Music Notation .....	5
2.2 The Notational Hierarchy .....	6
2.2.1 Score and Pages .....	7
2.2.2 Systems and Staves .....	7
2.2.3 Clefs and Key-Signatures .....	10
2.2.4 Time-Signatures .....	11
2.2.5 Bars and Barlines .....	12
2.2.6 Voices .....	14
2.2.7 Notes, Rests, and Chords .....	15
2.2.7.1 Notes .....	15
2.2.7.2 Rests .....	16
2.2.7.3 Chords .....	17
2.2.8 Beams .....	18
2.2.9 Articulation Symbols and Ornaments .....	19
2.2.10 Dynamics, and Auxiliary Symbols .....	20
2.2.11 Parts .....	21
2.3 Classifying Notation Symbols .....	22
2.3.1 Context-Sensitive Relationships .....	22
2.3.2 Semantic Constituents .....	22
2.3.3 Semantic Modifiers .....	23
2.3.4 Notational Conveniences .....	23
2.3.5 Spatially Dependent Symbols .....	24

### **3. Music Notation Software and Software Tools**

3.1 Overview .....	25
3.2 Music Notation Software .....	25
3.2.1 Finale .....	25
3.2.2 Encore .....	27
3.2.3 MUSICA .....	29
3.2.4 Copyist .....	29
3.2.5 Other Notation Programs .....	30
3.3 Sequencers and Educational Software .....	31
3.4 Software Tools for Music Notation .....	32
3.4.1 Notation Interchange File Format .....	33
3.4.2 The Notation Engine .....	34
3.4.3 Music Fonts .....	34
3.5 Object-Oriented Music Representation .....	36

### **4. Designing an Object-Oriented Toolkit for Music Notation**

4.1 Object-Oriented Development Methods .....	37
4.1.1 Selecting a Method .....	38
4.2 Object-Oriented analysis and Design .....	39
4.2.1 The Coad Method .....	39
4.3 Identifying objects and their relationships .....	41
4.3.1 Scores, Pages and Score Templates .....	42
4.3.2 Systems and Staves .....	45
4.3.3 Bars and Barlines .....	48
4.3.3.1 Bar Representation .....	51
4.3.3.2 Barline Relationships .....	53
4.3.3.3 Secondary Bars .....	55
4.3.4 Clefs, Key-Signatures and Time-Signatures .....	56
4.4 Voices .....	58

4.5 Note, Rest and Chord Relationships .....	59
4.5.1 Note Structure .....	59
4.5.2 Rests .....	61
4.5.3 Chords and MusicEvent Objects .....	62
4.6 Beams and Beamed Groups .....	64
4.7 Irregular Groups .....	67
4.8 MIDI Message Objects .....	68
4.9 Part Objects .....	69
4.10 Defining Object Relationships .....	70
4.11 Comparing MUSICA and the OOTMN .....	77
4.11.1 Abstract Classes in MUSICA .....	78
4.11.2 The MUSICA Class Hierarchy .....	78

## **5. Implementing the Toolkit for Music Notation in C++**

5.1 Selecting a Development Environment .....	81
5.2 Real-World Models and Machine Models .....	82
5.3 An Object-Oriented Implementation of the Model .....	83
5.3.1 Creating objects .....	84
5.3.2 Storing Objects .....	85
5.3.3 Searching for Objects .....	86
5.3.4 Moving Objects .....	89
5.4 Implementing Objects .....	93
5.4.1 Encoding Music Symbols .....	93
5.4.2 Representing Notes .....	93
5.4.2.1 Name Class .....	93
5.4.2.2 Pitch Class .....	94
5.4.2.3 Octave Representation .....	95
5.4.3 Line Symbols .....	96
5.4.4 Special Symbols .....	97
5.4.4.1 Slurs, Ties, and Phrase Markings .....	97
5.4.4.2 Beams .....	98

5.5 MIDI Playback .....	99
5.5.1 MIDI Synchronisation .....	100
5.6 Algorithms used by the OOTMN .....	103
5.6.1 Creating Beamed Groups .....	103
5.6.2 Calculating Grouplet Durations .....	105
5.6.3 Implementing MIDI Playback .....	106

## **6. Examples of Toolkit-Created Applications**

6.1 Designing Applications using the OOTMN .....	108
6.2 Notation and Composition software .....	109
6.2.1 User-interface Design .....	109
6.2.2 Using the Notation and Composition Program .....	111
6.2.3 Secondary Bars, Secondary Stems and Split-Stem Chords .....	114
6.2.4 Toolbar and Menu Options .....	115
6.2.5 Context-sensitive Symbolic Relationships .....	116
6.2.6 MIDI Playback and Recording .....	117
6.3 A Harmony Tutor .....	117
6.3.1 Selecting a Development Environment .....	118
6.3.2 Authoring Packages .....	119
6.3.3 Using the Harmony Tutor .....	120
6.4 An Embedded Notation Example .....	121

## **7. Conclusion**

7.1 Functionality of the OOTMN .....	123
7.1.1 Simplifying Application Development .....	124
7.1.2 Manipulating Notation Symbols .....	124
7.1.3 Extensions to Traditional Notation .....	125
7.2 Design of the OOTMN .....	125
7.3 Limitations of the OOTMN .....	126
7.4 The Borland Development Environment .....	127

7.5 A Software Engineering Perspective .....	128
7.5.1 Prototyping .....	128
7.5.2 Partitioning the Object Model .....	128
7.5.3 Simplifying the Development Environment .....	129
7.6 Future Directions .....	129
7.6.1 File Formats .....	130
7.6.2 Algorithmic Enhancements .....	130
7.6.3 Developing a Music Font .....	131
7.6.4 Optimising Display Updates .....	131
7.6.5 Porting the OOTMN .....	132
7.7 Summary of the Capabilities of the OOTMN .....	132

## **Appendices**

A. The OOTMN User Guide .....	133
B. The Object-Oriented Toolkit Library Reference .....	135

<b>Glossary</b> .....	166
-----------------------	-----

<b>Bibliography</b> .....	170
---------------------------	-----

<b>References</b> .....	171
-------------------------	-----

<b>Index</b> .....	175
--------------------	-----

## List of Figures

Figure 2.1 Symbols and their semantic constituents .....	22
Figure 2.2 Symbols and their associated semantic modifiers .....	23
Figure 3.1 The Finale user interface and special tool palette .....	26
Figure 3.2 The Encore user interface and symbol palettes .....	28
Figure 3.3 Piano roll notation .....	31
Figure 3.4 Note duration encoding schemes .....	33
Figure 3.5 Relating ASCII symbols to music symbols .....	35
Figure 3.6 Encoding notes in a music font .....	36
Figure 4.1 An Object and its attributes and methods .....	40
Figure 4.2 Generalisation-specialisation, and whole-part relationships .....	40
Figure 4.3 An instance relationship and its cardinality .....	41
Figure 4.4 Score and Score Template objects .....	43
Figure 4.5 Example of a score template .....	44
Figure 4.6 System, SystemBracket and Staff objects .....	47
Figure 4.7 Bar and Barline objects .....	49
Figure 4.8 Bar Specialisations and Secondary Bar objects .....	55
Figure 4.9 Clef, Time-signature and key-signature objects .....	57
Figure 4.10 Voice objects and their relationships .....	58
Figure 4.11 Structure of a note symbol .....	59
Figure 4.12 Whole-part structure of a Note object .....	60
Figure 4.13 Relationship between a Note and Accidental, Tie and Slur objects .....	61
Figure 4.14 A Rest Object .....	61
Figure 4.15 Chord class as an independent class .....	63
Figure 4.16 Chord class derived from Class MusicEvent .....	63
Figure 4.17 Distinct classes for notes and chord notes .....	64
Figure 4.18 Structure of BeamedGroup and Beam objects .....	65
Figure 4.19 Relationships between note durations .....	67
Figure 4.20 Structure of class Grouplet .....	68



Figure 4.21 A MidiMessage object .....	68
Figure 4.22 A Part Object .....	69
Figure 4.23 Representing notational relationships as classes .....	71
Figure 4.24 Object Relationships between classes BeamedGroup, Grouplet and Chord .....	72
Figure 4.25 Complete OOTMN Object Model .....	73-4
Figure 4.26 Message passing in the OOTMN.....	75-6
Figure 4.27 The Class Hierarchy of MUSICA .....	77
Figure 4.28 MUSICA Staff and MusicObject objects .....	78
Figure 4.29 MUSICA Class MusicObject and its derived classes .....	79
Figure 5.1 Relationships between the Windows API, the OWL and the OOTMN .....	81
Figure 5.2 Structure of a MusicSymbol object .....	83
Figure 5.3 Hierarchical relationships that determine object creation and storage .....	85
Figure 5.4 Searching process modelled by Coad notation .....	87
Figure 5.5 Searching process modelled in psuedocode .....	88
Figure 5.6 Searching using Polymorphism .....	89
Figure 5.7 Updating a symbol and it's dependent symbols modelled using Coad notation .....	91
Figure 5.8 Updating a Note and it's dependencies positions in psuedocode .....	92
Figure 5.9 Pitch classes and note letter names related to the keyboard .....	94
Figure 5.10 Hairpin object .....	96
Figure 5.11 A Cubic Bézier curve .....	97
Figure 5.12 Beams implemented as rectangles or parallelograms .....	98
Figure 5.13 Brace and bracket symbols implemented as bitmaps .....	99
Figure 5.14 Implementation of class MidiMessage .....	100
Figure 5.15 MMT MidiEvent structure .....	101
Figure 5.16 Calculating MIDI starting times .....	102
Figure 5.17 Calculating timestamps as MIDI offsets .....	103
Figure 5.18 Creating BeamedGroup objects .....	104

Figure 6.1	Application interface to the OOTMN .....	108
Figure 6.2	MusicSymbol objects and their associated hitspots.....	112
Figure 6.3	LineSymbol objects and their associated hitspots .....	112
Figure 6.4	Options controlling the movement of symbols .....	113
Figure 6.5	OOTMN command summary.....	114
Figure 6.6	The Score Application toolbar .....	115
Figure 6.7	Palette menus for different music symbols .....	116
Figure 6.8	Multimedia Builder and the Harmony Tutor executing simultaneously.....	120
Figure 6.9	Code example of embedded notation .....	122
Figure 7.1	Display updates using screen and memory device contexts.....	131

## List of Musical Examples

Example 2.1	Pitch determination at different hierarchical levels .....	6
Example 2.2	An unformatted page consisting of five-line and single-line staves .....	8
Example 2.3	A formatted page consisting of two systems .....	9
Example 2.4	Pitch divisions within a staff .....	9
Example 2.5	Leger lines above and below the staff .....	10
Example 2.6	Clef relationships .....	11
Example 2.7	Changing Meter - Leonard Bernstein, <i>America</i> from <i>West Side Story</i> .....	12
Example 2.8	Bars delimited by barlines - Mozart, Piano Sonata in C major, K.545 .....	13
Example 2.9	System Barlines .....	13
Example 2.10	Barlines derived from a logical grouping of staves .....	14
Example 2.11	Parallel voices - Johann Pachelbel, <i>Was Got Tutt, das ist wohlgetan</i> .....	14
Example 2.12	Context-dependencies determining a note's pitch .....	15
Example 2.13	Representation of voices with and without secondary stems .....	16
Example 2.14	Note and Rest symbols .....	16
Example 2.15	Chords and their associated stem .....	17
Example 2.16	Split-stem chords .....	18
Example 2.17	Beams grouping notes according to the meter .....	18
Example 2.18	Extended beams and fan beams .....	19
Example 2.19	Articulation Symbols .....	19
Example 2.20	Notation and execution of ornaments .....	20
Example 2.21	Notation of separate parts .....	21
Example 2.22	Parts of example 2.20 notated using distinct voices .....	22
Example 3.1	Spatial dependencies in Encore - Andrew Eales, <i>The Dark Stream</i> , 3rd mov. measures 1-2 .....	29
Example 3.2	Copyist example score - Debussy, <i>Reflets dans l'eau</i> .....	30

Example 4.1 Placement of Staves, Bars and Barlines - Witold Lutoslawski, <i>Dance Preludes</i> .....	46
Example 4.2 A single staff divided into bars .....	50
Example 4.3 Starting position of the first bar of a staff .....	50
Example 4.4 Barline positions of the first bar of a staff .....	51
Example 4.5 Two representations of a bar .....	51
Example 4.6 Piano / vocal score format .....	52
Example 4.7 System barlines created from overlapping barlines .....	53
Example 4.8 Barline notation used by Igor Stravinsky in <i>Abraham and Isaac</i> .....	54
Example 4.9 Barline notation used by Luciano Berio .....	54
Example 4.10 Shared barlines used by the OOTMN .....	55
Example 4.11 Time-signature positions in modern notation .....	58
Example 4.12 Rest notation in a two-voice canon .....	62
Example 4.13 Brahms, <i>Clarinet Sonata</i> no.1, first movement, measures 145 - 149 .....	66
Example 4.14 Defining notational relationships .....	70
Example 4.15 Unidiomatic notational relationships .....	72
Example 5.1 Staves and their bounding rectangles .....	88
Example 5.2 Name class and staff step relationships .....	94
Example 5.3 Octave representation .....	95
Example 5.4 Implementing MIDI timestamps .....	101
Example 5.5 Durations of irregular groups .....	105
Example 5.6 Incomplete bar and cross-staff beams .....	107
Example 6.1 Dragging Beam objects .....	110
Example 6.2 OOTMN examples of extended and fan beams .....	110
Example 6.3 OOTMN created examples of time-signatures .....	111

## Acknowledgements

I would like to thank all those who provided assistance and encouragement. My supervisor, Richard Foss, who suggested the idea, clarified numerous object-oriented issues, and patiently put up with me. Peter Wentworth of the Rhodes Computer Science Department, who answered my questions regarding embedded C++ code in Java. Windows Help programmers, Paul Arnot and Michael Cessna, who provided information concerning the Microsoft Windows Help System. All the Borland Object Windows Library (OWL) programmers for making their expertise available via the OWL mailing list and archives. InterSystems Concepts Inc., Formula Software Ltd. and MediaChance who provide free versions of their multimedia authoring software for evaluation and academic purposes. Object International Inc., for making their Playground object modelling software available for self-study and academic usage. Dominic Giampaolo, who placed his code for the calculation of Bézier splines in the public-domain. Paul Messick, creator of the Music Quest MIDI Toolkit, and the Maximum MIDI Toolkit. Without the Maximum MIDI Toolkit, implementing MIDI functionality within the OOTMN (Object Oriented Toolkit for Music Notation) would have been an enormously difficult task.

## Chapter 1

# Introduction

### 1.1 Computers and Music Notation

The use of computers to represent and store music notation is a fairly recent development, resulting from the widespread use of graphic user interfaces and the acceptable resolutions provided by graphics hardware. The Apple Macintosh platform pioneered applications for music notation during the mid nineteen-eighties, followed a few years later by software designed for other machine architectures. Applications that use music notation, tend to make use of programming code that is application specific. The absence of libraries supporting music notation that can assist programmers is the primary motivation for this study. Incorporating a library that is application independent into a project provides a partitioning that is desirable in successful software engineering. Dissatisfaction with the capabilities of many existing notation packages provided further ideas for the design of a notation library. Some of the currently available music notation software packages and software tools for the Microsoft Windows environment are discussed in chapter three.

### 1.2 Object-Oriented Software Design

Object-oriented technology models the real world by means of object hierarchies that correspond to real-world objects and their relationships. Emphasis is placed on modular relationships between data and the processes that operate on data. Other software engineering methods place emphasis on the modular design of processes that are not closely coupled to the data that is processed. Music notation is a problem domain that is naturally object-oriented due to the intricate relationships between different symbols (objects), and the widely differing characteristics (data) of each symbol. Consider the following explanation of object-oriented representation given by Stroustrup [Stroustrup, 1988]

“Concepts do not usually come as self-contained entities. On the contrary, most concepts relate to other concepts in a variety of ways. .... Therefore, representing concepts as types in a program also requires ways of expressing the relations between types. C++ lets you specify hierarchically organised classes. This is the key feature supporting object-oriented programming.”

If the term ‘concept’ is regarded as being a music symbol, the above could be directly referring to music notation.

### **1.3 Designing Music Notation Software**

Existing music notation software may be divided into two broad categories, programs that only provide a graphic representation of music notation, and applications that allow a graphic representation to be translated into a sonic realisation. A purely graphic approach is not context-sensitive, while the sonic interpretation of music symbols requires a high degree of context-sensitivity within a hierarchical arrangement of different symbols. These two approaches contain a trade-off regarding the power of visual, symbolic expression, and the ability to unambiguously interpret a given symbol within the context of other symbols.

Software that is not context-sensitive places no constraints on the user, allowing any available symbol to be placed anywhere on a page. Symbols that are not directly supported by the software may be designed in a graphics format and imported into the program. Complex music scores made from a collage of symbols are easily created using a purely graphical approach. Unfortunately, graphical computer scores that do not have underlying context-sensitive representations have no music theoretical meaning at the machine level. The visual score has no underlying machine representation that can be translated into sonic events, or used for useful musicological research. Graphical scores can only be viewed on a terminal or printed.

### **1.4 Music Notation and MIDI**

Music notation software that allows a score to be played on a MIDI (Musical Instrument Digital Interface) device must have an underlying structure that is at least partly compatible with music theory. Such a system must follow restrictions regarding the placement of symbols, allowing the user to only place symbols within a meaningful context. Unfortunately, these restrictions may be so severe as to limit the practical usefulness of the software when confronted with unusual practices commonly found in twentieth-century music. Conventional music theory contains many exceptions

to established convention, while contemporary practices are only limited by the resourcefulness of the composer. Music notation, is however, a dynamically developing system where new practices gain widespread acceptance. The MIDI protocol is not described in detail. A complete description of MIDI may be found in the MIDI specification [International MIDI Association, 1983, or one of the many texts dedicated to MIDI such as Anderton [Anderton, 1986] and Lehrman [Lehrman, 1993].

## 1.5 Project Objectives

Many of the ideas central to this research were conceived during the course of designing a library for music notation, and were not considered prime objectives at the outset. The original conception of an object-oriented representation of music notation was restricted to a context-sensitive representation that would only consider traditional music notation. This conception was later expanded to include non context-sensitive relationships that provide greater flexibility and extensibility, and allow many modern conventions as an extension of traditional practices. Chapter two provides an overview of conventional staff notation, as well as examples of modern practices.

The design attempts to emphasise generality, enabling less common aspects of music notation to be developed from a core set of traditional notational constructs. This design process is discussed in chapter four. It was hoped that a design could be generated that would be flexible enough to allow future additions and enhancements. During the course of the design process a slight shift in emphasis occurred when certain music symbols were discovered to be capable of transformations that allowed different visual arrangements without undermining underlying context-sensitive relationships. These symbols function as visual aids and have no influence on the sonic realisation of the score. Such symbols can be treated in a symbolic, context-free manner once their basic underlying context-sensitive relationships have been defined. Solutions to many of the limitations found in music software, as well as to various contemporary notational practices naturally follow the identification of these symbols. This discovery presented three categories of symbols, context-sensitive symbols, context-sensitive symbols used out of context while retaining an underlying context-sensitivity, and symbols that are always context-free i.e. graphical symbols.



From this realisation grew the idea of an environment that combines both the symbolic and context-dependent approaches, with additional relationships that provide graphic flexibility by altering the visual appearance of symbols having context-sensitive relationships.

Implementation of the toolkit in the C++ programming language, and the design refinements required for a successful implementation are discussed in chapter five. Chapter six presents applications created with the OOTMN. Different types of applications require that the toolkit be used in different ways, providing an evaluation of the functional capabilities of the toolkit. An evaluation of the toolkit, as well as a discussion of possible enhancements and extensions is presented in chapter seven. This chapter concludes with a discussion of the software-engineering insights gained from the study and provides a summary of the capabilities of the toolkit. Detailed information concerning the use of the toolkit is provided in appendix A and appendix B.

Throughout the text object names are written as proper names to distinguish the object 'Staff' from the music symbol 'staff'. Object and attribute names consisting of more than one word are joined together, For example, the English description 'music symbol' when used as the name of an object is notated as 'MusicSymbol'.

## Chapter 2

# An Overview of Music Notation

### 2.1 Traditional Music Notation

Music staff notation is one of the most powerful and successful symbolic systems invented by man. A core set of rigorously defined concepts that can be used in a variety of ways, have ensured the system's survival and evolution over many centuries. Staff notation is an abstract system that occurs within the limits of its own conventions, being independent of any individual musical instrument, or sound generating hardware. The following discussion provides an overview of staff notation, emphasising areas that are relevant to a computer-based representation. Although the text covers aspects of elementary music theory, it is not meant to be a music theory tutorial.

'Traditional notation' refers to the notational conventions used in Western music, developed during the seventeenth, eighteenth, and nineteenth centuries. The terms 'music notation', or 'notation' refer to traditional practices. Trends during the twentieth-century have retained many of the aspects of traditional practice while introducing a variety of extensions. These developments have not been standardised, and may even only exist within the works of a particular composer. In addition to the common use of notation for performance purposes, staff notation has been modified to represent purely theoretical concepts found in music theory and analysis. Schenkerian notation, discussed by Salzer [Salzer, 1962] and others, is an analytical system that uses staff notation to represent the structure of a work.

Music symbols are notated within a two-dimensional space, the vertical dimension representing pitch and simultaneity, while the horizontal dimension represents time divisions. Within this basic model, a wide range of contextual dependencies occur within a hierarchy consisting of a local level, intermediate levels, and a global level. Intermediate levels are often nested according to the scope of a particular symbol, in the same way that nested symbols in programming languages have scope and visibility. Symbols may also be influenced by the next higher level in the absence of local information, just as the re-declaration of a symbol at a lower level overrides global declarations in programming languages. Unlike programming language conventions, some symbols such as time and

key-signatures do not have a local scope. Their usage alters the globally defined time and key-signatures, as they are visible from the point of definition to the end of the score. In the following example pitch interpretation is dependent on the global key-signature of A-flat major which is overridden at the staff level, the bar level and the note level :

The diagram illustrates hierarchical levels of key-signature and accidental override in musical notation. It is divided into two parts:

- Global key-signature:** The top part shows a grand staff (treble and bass clefs) with a key signature of two flats (B-flat and E-flat) and a 4/4 time signature. This signature is visible across the entire score.
- Bar level key-signature:** The bottom part shows the same grand staff. In the first bar, the key signature changes to two sharps (F-sharp and C-sharp). In the second bar, the key signature returns to two flats. A note in the second bar has a flat accidental, which overrides the bar-level key signature for that specific note.

Labels in the diagram include: "Global key-signature", "Bar level key-signature", "Staff level key-signature", and "Note level accidental".

Example 2.1 Pitch determination at different hierarchical levels.

Staff and bar level key-signatures remain visible until the end of the score unless redefined. Accidentals have bar scope, they cease to have any influence outside of the bar in which they occur. The staff-level key-signature may also be interpreted as belonging to the first bar of the staff. Chapter four discusses the design of the OOTMN and examines different possible interpretations of relationships between music symbols.

## 2.2 The Notational Hierarchy

Printed musical works are known as scores, and consist of separate pages bound together in book form. Analogies between the layout of printed books and music scores do not exist at levels below the page level. Terms such as 'phrases' and 'sentences' are used in music theory to denote logical divisions within the music itself, not to the physical symbolic layout. The following sections introduce the different components of a printed score, as well as the relationships that exist between different components.

### **2.2.1 Scores and Pages**

A score represents a complete musical work. Large compositions may consist of several movements, sections that are both related to each other and musically self contained. Computer representations that allow playback typically store a complete work or a single movement in a single file.

Global characteristics of a work or movement which occur at the score level include, time-signatures, key-signatures, and instrumental combinations. We may refer to the ‘time’ of a waltz, the ‘key’ of a song, or the instrumentation of a string quartet. Instrumentation influences the physical layout of a score by virtue of the number of instruments and the notational requirements of specific instruments. Other global characteristics include tempo, instrument transposition and page formatting information. Although these global characteristics may change during the course of a work, they are usually (especially within traditional practices) fixed for the entire score, complete movements, or significant portions of a movement.

### **2.2.2 Systems and Staves**

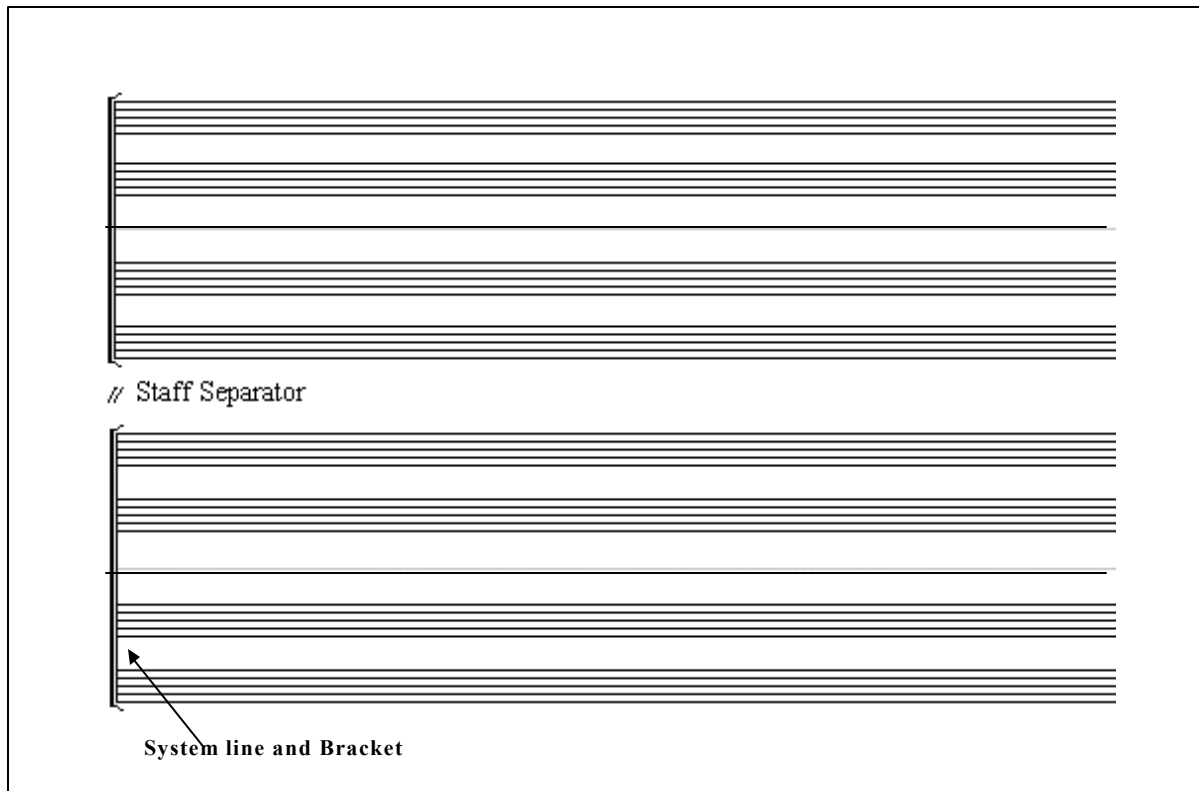
Physical subdivisions of the score occur as pages, which are convenient sub-divisions of a musical work in printed form. Pages have no musical connotation, and thus do not influence the musical content or interpretation of a work. Music manuscript paper traditionally consists of five-line staves. Manual masking and pasting procedures are required to produce an unformatted page that has single line percussion staves added as shown in example 2.2. Traditionally, the composer or copyist adds the required system indications to format the page, as well as the remaining symbols forming the musical work itself. Orchestral scores can consist of pages where each page has a unique structure reflecting the orchestration requirements of a particular musical passage.



Example 2.2 An unformatted page consisting of five-line and single-line staves

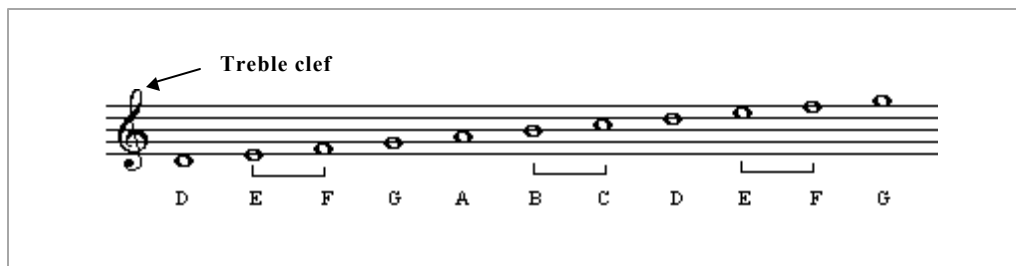
A system denotes a temporal grouping of staves. Musical events occurring on different staves within the same system are executed simultaneously. Computationally, a system is a process where each of its constituent staves represents a parallel thread of execution. Staff events within a system are executed in lockstep according to a fixed temporal division known as the meter. These concepts are explained later in this chapter.

More than one system may be notated on a single page if the page is large enough to represent the total number of staves required by multiple systems. Staves are visually grouped by a system line and a system bracket on the left of the system. Visual separation is enhanced by an optional separator symbol consisting of two short, angled parallel lines. Example 2.3 shows a page formatted into two systems, each system consisting of four staves :



Example 2.3 A formatted page consisting of two systems.

Staves are formed by groups of between one and five parallel horizontal lines. These lines provide a discrete spatial division of the y-axis which is used to indicate pitch. Notes representing different pitches are notated on, and between staff lines. Actual pitch divisions do not correspond to the equidistant spatial divisions. Pitch differences are dependent on the particular alphabetic note names that denote pitch, as well as the use of accidentals. The sequence of note names within a staff is in turn dependent on the clef (discussed in the next section) notated at the start of the staff. In example 2.4 the bracketed pitches have an interval (the musical distance between two pitches) of a semitone, or 100 cents in equal temperament. All other intervals are a tone or 200 cents. This arrangement is unique to a staff using the treble clef.



Example 2.4 Pitch divisions within a staff.

Acoustics makes use of the ‘cent’ unit to denote pitch differences. An octave, the distance between two notes of the same alphabetic name where one note is double the frequency of the other, consists of twelve-hundred cents.

As an aid to rapid visual interpretation, staves do not usually have more than five lines. A staff should not be confused with tablature notational methods that graphically represent the strings of an instrument. These methods use parallel lines on the y-axis to represent strings, the x-axis indicates the position on a string that produces a particular pitch. Notes that occur outside of the staff are notated by means of ledger lines that temporarily extend the staff for a single note as shown in example 2.5. Staves are not bounded from above or below i.e. the lowest and highest lines of a staff are determined by the pitch requirements of the notated notes and are not characteristics of the staff itself.



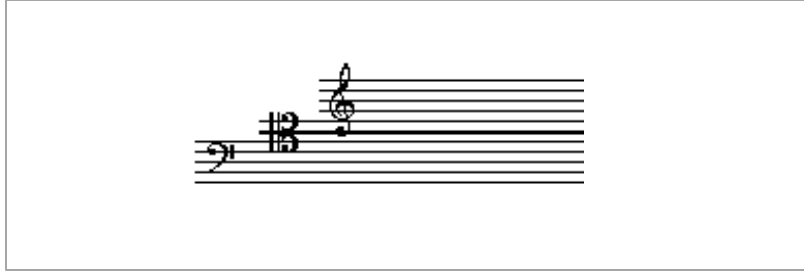
Example 2.5 Leger lines above and below the staff.

It is unusual to find notes notated using more than five or six ledger lines above or below the staff as such passages are difficult to read fluently. Ledger lines are often avoided by indicating that a passage is to be performed an octave higher or lower by placing the signs 8va or 8vb above or below the passage to be transposed.

### 2.2.3 Clefs and Key-Signatures

The portion of the pitch space represented by a staff is indicated by placing a clef at the beginning of the staff. Four different clefs are used according to the range of the instrument and the current tessitura of the music being notated. Example 2.6 illustrates the relative pitch regions represented by the treble  $\text{C}_1$ , tenor  $\text{C}_2$ , and bass  $\text{C}_3$  clefs. The note middle C occurring within the context of each of

the clefs is shown as a bold line :



Example 2.6 Clef Relationships.

The alto clef uses the same symbol as the tenor clef but places middle C on the third, rather than the fourth line of the staff. The range of an instrument describes the upper and lower bounds of pitches that may be produced on a particular instrument. Within this range, the emphasis of a particular tessitura or pitch area often necessitates a clef change to avoid an excessive use of ledger lines.

Tonality (i.e. a key, or pitch centre) is determined by a key-signature that implicitly modifies the pitch interpretation of notes. A key-signature is only notated at the beginning of every staff. This convention is not governed by the scope of the key-signature, which extends from the point of definition to the end of the score, but serves as a reminder to the reader. A key-signature change may occur at the start of any bar in the score. Discussion of local and global levels in section 2.1 illustrated the possibilities governing key-signature changes.

### 2.2.4 Time-Signatures

The number of beats occurring in each bar, as well as the duration of each beat is given by a time-signature consisting of two integers notated one on top of the other within the staff. A time-signature has no relationship to, and should not be confused with a fraction. The top value defines the meter i.e. the number of beats occurring in the bar, while the lower value indicates the durational value associated with each beat. Time-signatures have the same scope as key-signatures, but unlike key-signatures, are traditionally only notated at the start of a work or when a metric change occurs.



Notation of time signatures has been extended by modern composers to include the use of a notehead rather than an integer as the lower value, use of multiple upper values, dual time-signatures, and the placement of time-signatures outside of the traditional position within a staff. Leonard Bernstein's musical *West Side Story* makes use of a changing meter specified by the dual time-signature of example 2.7. The actual meter of each bar is determined by the grouping of notes within the bar.



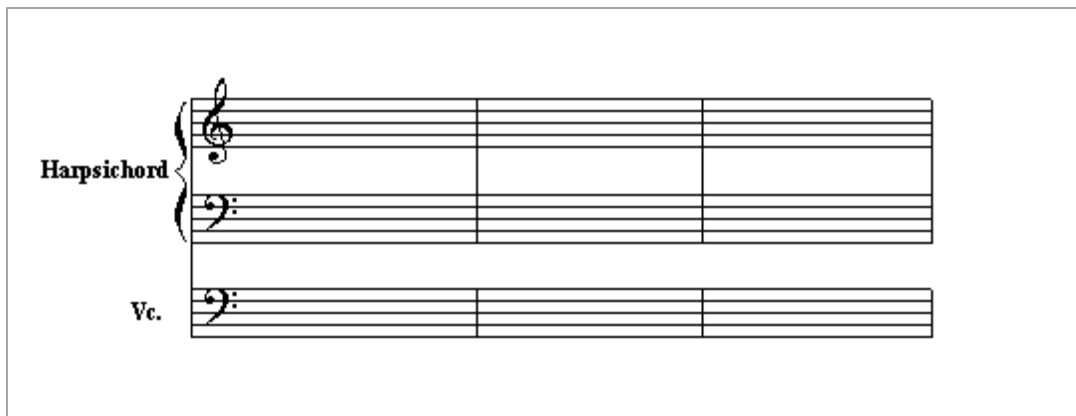
Example 2.7 Changing Meter - Leonard Bernstein, *America* from *West Side Story*.

Grouping of notes in example 2.7 indicates that the meter of bar one is 6/8, while bars two and three should be treated as 3/4 bars. An underlying quaver duration remains constant for both time-signatures. Use of a changing meter is another example of a notational convenience, as only one time-signature is used within a single bar. Use of either time-signature would result in the correct rhythm as both time signatures have a total duration of six quavers. A changing meter emphasises the interpretation of a rhythm (i.e. the correct placement of rhythmic accents), it does not influence the durations forming the meter from which a rhythm is derived.

### 2.2.5 Bars and Barlines

A staff's x-axis is divided according to a fixed sum of durational values indicated by the time signature. Vertical bar lines indicate the start and end of each division. Bar sizes are not proportional to the durational content of a bar, but are determined by the number of notes within a bar. In the example below where the barline positions are indicated by arrows, the size of the last bar can be compared to the size of the first bar :





Example 2.10 Barlines derived from a logical grouping of staves.

By convention bar lines usually overlap so that the right-hand barline of a bar is also the left-hand barline of the next bar. A detailed discussion of the design challenges posed by barlines is given in chapter four.

## 2.2.6 Voices

Voices permit parallel streams of melodies within a single staff. Notes occurring at the same metric division are played simultaneously. This usage of the term ‘voice’ within a notational context differs from the traditional contrapuntal concept of a voice as a separate melody, irrespective of the staff on which the melody is notated. The following example illustrates these differences :



Example 2.11 Parallel voices - Johann Pachelbel “Was Got Tutt, das ist wohlgetan.”

Three contrapuntal voices (separate melodies) occur in example 2.11, the first voice occupies the first staff, the remaining two divide the lower staff into two separate streams of events forming notational voices. This arrangement allows more than one melody to be notated on a single staff. Notational voices are separate streams of notes (not necessarily complete melodies) that are visually separated by the direction of the note's stems, each voice having either ascending or descending stems.

## 2.2.7 Notes, Rests and Chords

Notes represent sonic events and have pitch and duration, while rests represent silence. Chords are groups of notes that sound at the same time.

### 2.2.7.1 Notes

A note is usually placed on a staff where the clef gives the note an alphabetic name that denotes a specific pitch. A note that is not placed on a staff has only relative durational attributes implied by the duration of the symbol itself. Percussion parts that are notated on a single staff provide only durational information. The pitch of notes on a staff may be modified by key-signatures and accidentals. Exact pitch (i.e. a specific MIDI key or frequency) can only be assigned to a note after determining the staff, key-signature and the presence or absence of an accidental. A note's pitch is thus contextually dependent on its position on the staff, the staff's clef, as well as any key-signature and accidental that may be present. These dependencies are shown in example 2.12 :

C and F letter names from the treble clef

D flat falls within scope of the previous D flat

Natural overrides F sharp of the key-signature

C sharp and F sharp according to the key-signature

A Flat overrides A natural of the key-signature

Natural cancels previous sharp symbol

Example 2.12 Context-dependencies determining a note's pitch.

As mentioned in section 2.2, accidentals have bar scope, modifying the pitch of all following notes occupying the same staff step, as well as all notes having the same letter name irrespective of octave.

A single note can have two stems associated with it, allowing the visual representation of two voices having identical pitches occurring at the same time to share noteheads. This style of writing is especially idiomatic of guitar music. Example 2.13 a) illustrates secondary stems associated with a shared notehead. Example 2.13 b) is an equivalent representation using two separate voices :

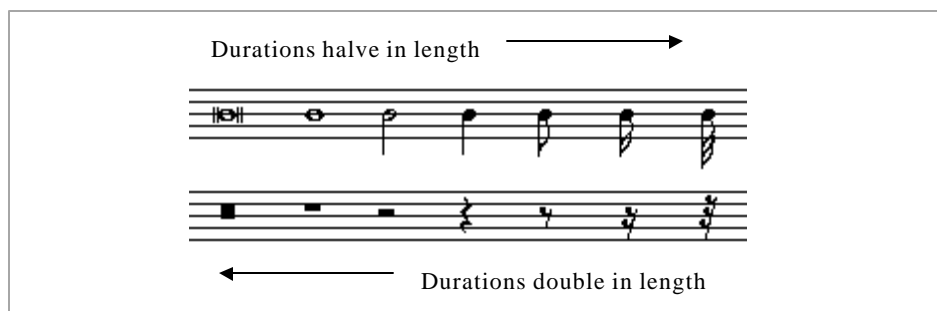


Example 2.13 Representation of voices with and without secondary stems.

Example 2.13 a) is a clearer representation of the passage that uses fewer symbols and beams to logically group notes.

### 2.2.7.2 Rests

Rests are symbols that indicate silence for a specified duration. Vertical placement of rests on a staff is determined by the context in which rest is used. Default positions occur within the staff, it is only necessary to use other positions when rests belong to different voices. Placement on the y-axis serves only to indicate a logical positioning as silence (i.e. a non-event) cannot have a definite pitch. Notes and rests explicitly indicate duration by means of their graphical composition, placement on the x-axis does not influence duration. Example 2.14 shows note durations, and their associated rests occupying their default staff positions :



Example 2.14 Note and rest symbols.

Notes and rests have relative durational values. In the above example each symbol has a duration that is twice the duration of the preceding symbol or, half the duration of the succeeding symbol. The lowest integer of a time-signature defines a reference duration which occurs at a given tempo. All durations are relative to this reference duration. It is important to realise that actual, performed durations are dependent on the time-signature and tempo of the music.

### 2.2.7.3 Chords

The term ‘chord’ has at least three possible meanings. Firstly, a group of notes that sound simultaneously, usually but not always starting at the same time. In this sense a chord may consist of notes that exist on distinct staves or systems. As an example consider a ‘chord’ played by an entire symphony orchestra. Secondly, a group of notes arranged vertically (or as near vertically as possible on the same staff. Such a chord is notated using a single, common stem for all the notes that constitute the chord. Thirdly, a special type of chord that forms the basis of a branch of music theory known as harmony is the four-part chord. Four-part chords consist of four notes that are notated on a grand staff of two staves, two notes occurring on each staff. Four-part chords are used in the harmony tutor application discussed in chapter six. The second type of chord is the most familiar chord, consisting of a vertical grouping of notes having the same duration, with a single upward or downward stem as illustrated in example 2.15 :



Example 2.15 Chords and their associated stems.

Use of the term ‘chord’ may also imply a triad where one or more notes occurs more than once. A chord’s stem starts at the highest or lowest note and traverses all of the notes contained in the chord. The note at the opposite end from the starting position determines the length of the stem. Chords are only notated with a stem if their duration requires a stem.

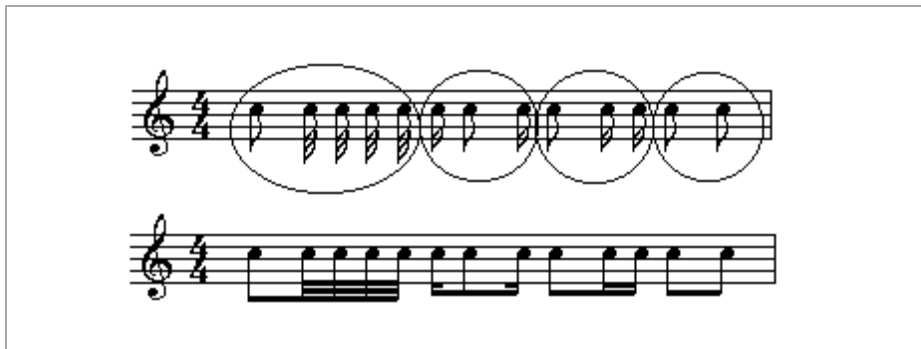
Chords that contain notes which share a staff-step but are not enharmonically equivalent (i.e. they have the same alphabetic name, but have different accidentals) must be specially formatted, as notes occupying the same staff-step cannot be vertically aligned. Such chords require more than one stem and are termed ‘split-stem’ chords. Example 2.16 illustrates a split-stem triad and split-stem dyad :



Example 2.16 Split-stem chords.

### 2.2.8 Beams

Notes containing flags (i.e. notes of a quaver / eighth-note duration or less) that share a common beat are beamed together by replacing shared flags with beams as shown in example 2.17 :



Example 2.17 Beams grouping notes according to the meter.

A single beam replaces the flags of individual notes, serving as a common flag for all notes having the same or shorter durational value. Notes that share common beams are referred to as a beamed group. Beams provide a visual separation of beats in the same way as whitespace visually separates written words. Rhythmic notation that does not make use of beams is extremely awkward to read fluently.

Twentieth-Century extensions to beam notations include extended beams and fan beams shown in example 2.18. Extended beams provide a visual grouping of the notes and rests that form a beat. They typically occur in complex passages that are difficult to read without the logical groupings indicated by extended beams.

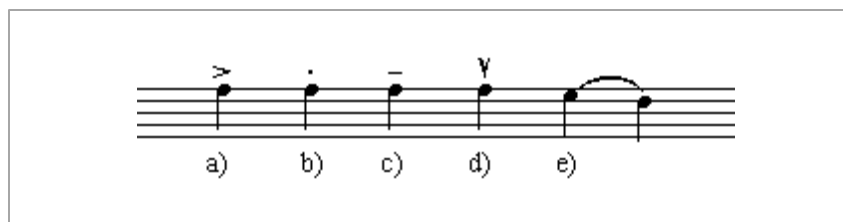


Example 2.18 Extended beams and fan beams.

Fan beams indicate accelerando and ritardando over the length of the beam, shorter-lengthed notes occurring at the apex of the fan.

## 2.2.9 Articulation Symbols and Ornaments

Articulation symbols modify the manner in which a note is performed (i.e. its attack and release in synthesiser terminology), as well as the note's duration. Articulation symbols are usually placed on the opposite side of a notehead from the notehead's stem. Example 2.19 provides examples of articulation symbols :



Example 2.19 Articulation Symbols.



The example at a) is an accent that modifies the attack and duration of the note, b) is a staccato symbol that shortens duration, c) lengthens duration, while d) is a short, sharp accent. The slur at e) implies that the length of the first note carries until the onset of the second note.

Ornaments, developed during the seventeenth and eighteenth centuries, are graphic symbols that represent embellishments of a note or group of notes. These symbols function as a musical shorthand that must be interpreted by the performer. Examples of ornaments and their interpretation are shown in example 2.20. The top staff illustrates the notation and execution of two different turns, the bottom staff illustrates the notation and execution of two different trills.

The image shows two musical staves illustrating ornaments. The top staff shows two examples of turns. The first example is labeled 'Notated' and shows a single note with a turn symbol (an infinity-like symbol) above it. The 'Performed' version shows the note with a series of sixteenth notes above it, representing the turn. The second example is also labeled 'Notated' and shows a note with a turn symbol above it, followed by a slur over two notes. The 'Performed' version shows the first note with a turn, followed by the second note, with a '3' below the second note indicating a triplet.

The bottom staff shows two examples of trills. The first example is labeled 'Notated' and shows a note with a trill symbol (a vertical line with a dot) above it. The 'Performed' version shows the note with a series of sixteenth notes above it, representing the trill. The second example is also labeled 'Notated' and shows a note with a trill symbol above it, followed by a slur over two notes. The 'Performed' version shows the first note with a trill, followed by the second note.

Example 2.20 Notation and execution of ornaments.

Note the context-sensitive interpretation of the turns, depending on whether the turn occurs on a note or between two notes. The ending given to the second trill is often not notated but implied by the style period of the music. Many ornaments, especially trills must be interpreted within a historical, as well as a notational context. Chapter four presents a general solution to the problem of correctly performing ornaments using MIDI.

### 2.2.10 Dynamics, and Auxiliary Symbols

Dynamic symbols such as *p* (piano, soft), and *f* (forte, loud) are used to indicate approximate, relative dynamic levels. Dynamic levels are dependent on the style and tempo of the music, and are

thus difficult to interpret accurately using MIDI. Auxiliary symbols include bowing indications, formats, and symbols characteristic of particular instruments such as guitar string numbers. The OOTMN does not presently implement support for auxiliary symbols. Implementation of these symbols is tedious rather than difficult, due to the vast number of possible symbols.

### 2.2.11 Parts

A part is the portion of a score that is to be performed on a single instrument. Parts may be notated on a single staff for wind, string and percussion instruments or, notated on multiple staves for the harp and keyboard instruments. Instrument parts notated on multiple staves are easily extracted from a score by simply extracting the required staves. String and wind instruments pose greater challenges, as parts may or may not share a single staff. The number of staves used, as well as the notational conventions used within a staff can change from system to system. Example 2.21 shows a staff used to notate two trumpets. The second trumpet is required to perform the first measure as indicated by the roman numerals above the measure.



Example 2.21 Notation of separate parts.

Both trumpets play in bars two and three, although the style of notation differs in each of these bars. A single voice is used to notate the two trumpets in bar two, two separate voices are utilised in bar three. Notation software that must extract two separate parts from these three bars, must determine to which part the first bar belongs, extract the different parts from the single voice of the second bar, and assign a single voice from bar three to each part. The situation is further complicated when instruments (such as the two trumpets in the above example) are initially notated on one staff and then notated on separate staves in another system. Successful part extraction requires that the exact staff and voice, as well as the relative position within a single voice (highest pitched, or lowest pitched notes in bar two of the above example) be known for each bar of the part. Part extraction is simplified by restricting parts to a particular staff and voice as illustrated by the example below which is an alternative way of notating example 2.21 :



Example 2.22 Parts of example 2.21 notated using distinct voices.

Such an arrangement requires that the user abide by the restrictions imposed by the software which are dependent on the underlying representation scheme.

## 2.3 Classifying Notational Symbols

Music symbols can be classified in a variety of ways. Categories that may be useful to computer-based representations are suggested in the following sections.

### 2.3.1 Context-sensitive Relationships

Music notation is a highly context sensitive symbolic system where symbols are interpreted within the context of other symbols. Three types of context-sensitive relationships exist. Firstly, symbols that must occur to successfully determine the meaning of certain other symbols. These symbols function as semantic constituents. Secondly, symbols that function as semantic modifiers may modify the meaning of another symbol, but are not required for interpretation. A symbol that can be modified by another symbol has a default interpretation in the absence of any modifier symbol. A third type of symbol (presented in section 2.3.4) does not alter the semantic interpretation of other symbols but functions as a notational convenience.

### 2.3.2 Semantic Constituents

The following table shows symbols and their associated semantic constituents :

<u>Symbol</u>	<u>Semantic constituent</u>
Staff	Clef
Bar	Barline
Note	Staff, (Leger line)

Figure 2.1 Symbols and their semantic constituents.

Note that partial representations are possible, a staff without a clef or notes not notated on a staff may be used to represent rhythmic durations. These practices are limited to specific situations such as the notation of percussion instruments. In most cases the symbols of figure 2.1 require the presence of a semantic constituent.

### 2.3.3 Semantic Modifiers

The table of figure 2.2 shows symbols and their associated semantic modifiers which alter the meaning of the symbol :

<u>Symbol</u>	<u>Semantic Modifier</u>
Note	Accidental, Dot, Dynamic Sign, Hairpin, Articulation, Octave sign.
Rest	Dot.
Chord	< same as for note >

Figure 2.2 Symbols and their associated semantic modifiers.

Semantic modifiers are only used as required. The symbols that they modify have an unambiguous semantic interpretation in the absence of a modifier.

### 2.3.4 Notational Conveniences

Notational conveniences are graphic symbols having no underlying meaning within the OOTMN. This characteristic only occurs within a computer representation of music notation that consists of an

underlying machine representation and a graphic representation. Altering the graphic view does not influence the machine representation. The duration characteristic of a note, as well as the properties of beams and barlines provide examples. Once a note's duration is fixed, the notehead symbol may be changed without altering the duration of the note. Such an alteration to the traditional printed score would change the meaning of the note, as the graphic (printed) representation is the only representation. Beams exist as a durational indicator (replacing the flags of notes beamed together) as well as a grouping mechanism that aids performers in correct rhythmic interpretation. Computer representations can change the function of a beam to exclude durational information. This is made possible by the existence of the notes that are beamed before the beam is formed. Thus the note durations exist independently of the beam, which only serves as a grouping mechanism. The positions of barlines determines the extent of a bar. A barline's length can be altered without changing its position, allowing modern barline notations that do not influence the size of a bar. The design of barlines and beams are discussed in detail in chapter four.

### **2.3.5 Spatially Dependent Symbols**

Symbols having semantic constituents and semantic modifiers establish their relationships by means of a spatial relationships that provide clear visual interpretations. These relationships can be altered to provide visual arrangements characteristic of modern notation. The spatial relationships between a staff and its clef and time-signature provide an example. Once the association between a staff and a clef, or a staff and a time-signature is formed, the clef or time-signature may be moved outside of the staff without altering the existing relationship.

This chapter illustrated conventional aspects of music notation. Developments during the twentieth century have included both radical departures from common practice, as well as a myriad of extensions that retain traditional elements. Contemporary music notation is discussed by Cole, [Cole, 1974]. Many of these innovations are extremely difficult to achieve in software. Suggestions for solving some of these problems based on the notational relationships discussed in this chapter are presented in chapter five.

## Chapter 3

# Music Notation Software and Software Tools

### 3.1 Overview

A wide variety of commercial notation packages are available, as well as a number of shareware and freeware programs. Some of these programs are discussed and evaluated in this chapter. Other software that utilises music notation, including sequencer software and music education software are also mentioned. Music notation software tools that are discussed include a proposed standard for the storage and representation of musical scores, and a notation library similar to the OOTMN.

### 3.2 Notation Packages

In a competitive market, commercial vendors are reluctant to provide technical information regarding their products. The observations that follow are thus limited to general observations that centre around the user's impression, rather than an evaluation of the software design. Since the late nineteen-eighties two of the leading commercial notation programs have been Finale from Coda Music Technology, and Encore from Passport Designs. Although both products are able to typeset and playback music notation, they have widely differing capabilities as well as different approaches to the design of a user interface.

#### 3.2.1 Finale

During 1997, Coda Music Technologies released Finale97 which followed version four of Finale. The most interesting aspect of Finale97 is a design architecture that allows third-party developers to provide software modules termed 'plugins' that can be seamlessly integrated with Finale. This feature allows Finale to be extended for specific tasks such as the analysis of musical works. Finale makes use of TrueType fonts, allows MIDI playback and real-time transcription of MIDI input, and is able to represent an astounding variety of notational complexities. Finale's weakest point is the awkward user interface that is not always intuitively constructed. The user of Finale97 is presented with the opening window shown in figure 3.1 An incomplete staff, that does not occur within the context of a page is a disconcerting introduction to the user. The main tool window on the right-hand

side may change the menu options depending on the category of operations selected. A Special Toolbox category opens the palette shown to the left of the main palette that provides a variety of beaming and stem movement options.

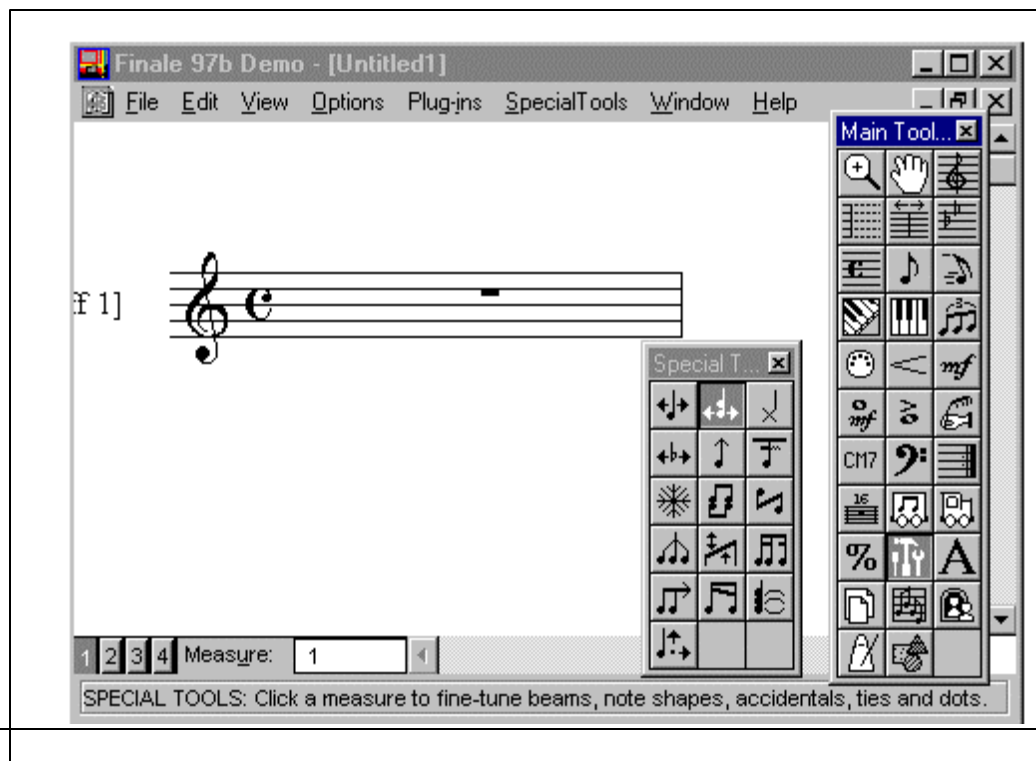


Figure 3.1 The Finale user interface and special tool palette

Use of these different options is not always intuitive, creating a steep learning curve for the user. Despite these difficulties, Finale is extremely powerful, and has developed a loyal following of sophisticated users.

Finale's design philosophy, as illustrated by the options in figure 3.1 is to create high-level solutions to a wide variety of notation problems. The user cannot directly interact with music notation symbols without first selecting the desired operation from a menu. By contrast, the OOTMN attempts to implement sophisticated operations from a collection of elementary click and drag operations within a general WYGIWYS ('What You Get Is What You See') page-oriented view.

Many of the less common notational constructs do not require separate functionality to be built into the programming

code. Chapter four and five illustrate how many modern extensions to conventional music notation can be implemented as natural extensions of traditional practice.

It is probable that Finale does not have an object-oriented design, the awkward user-interface is the result of a tight coupling between the user interface and the different modules that make up the program. Coda Music Technologies has never made a serious attempt to improve the user interface. Finale users, who tend to be musically sophisticated and computer literate, are prepared to spend the time and effort required to gain proficiency in the use of the software.

### **3.2.2 Encore**

Encore has an intuitive interface that allows tasks to be achieved quickly and effortlessly. The page-oriented user interface, as well as click and drag insertion and editing are the strongest features of Encore's design. Figure 3.2 shows the user interface with symbol palettes on the left, and toolbar at the top. Most symbols are simply dropped into position and adjusted by dragging. The style of the windows and palettes suggests that the software was developed using Borland C++. The score processor presented in chapter six makes use of a user interface that closely resembles the user interface design of Encore. Many of the ideas behind the design of the OOTMN were conceived as a result of personal experience using Encore. The software has a rigid context-sensitive design that makes any unusual use of notation difficult to implement. Examples include staves which cannot be individually moved, time-signatures and key-signatures that are given default positions that cannot be altered, beams that are constrained to be parallel to each other, and stems that cannot be manually adjusted. This rigidity can be extremely frustrating to the user, who is forced to add graphic symbols, and to develop unorthodox methods of using the available tools to achieve the desired layout.



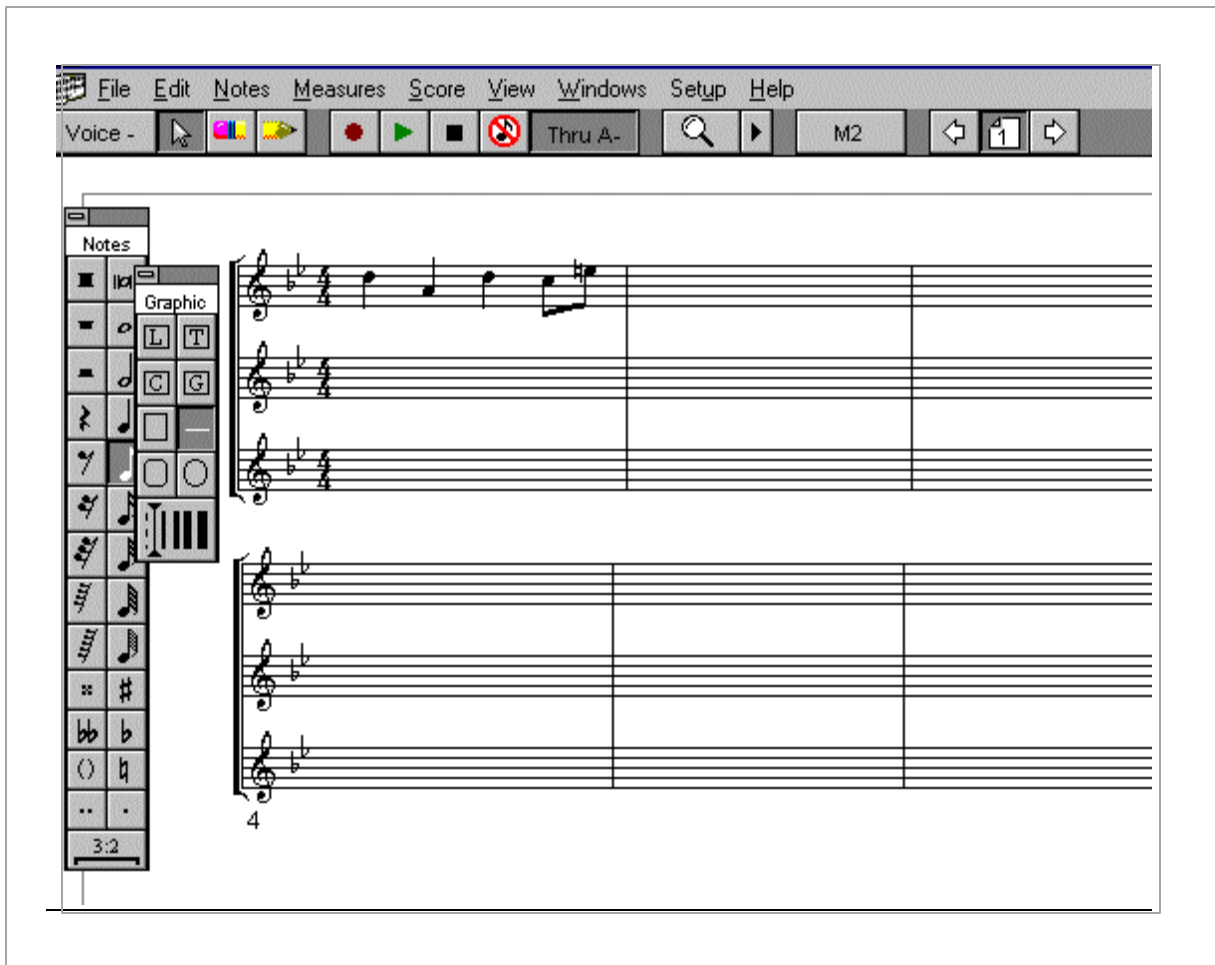


Figure 3.2 The Encore user interface and symbol palettes.

The weakest aspect of Encore's design is the failure to implement a formatting scheme that retains spatial relationships among symbols. Example 3.1 illustrates the breakdown of spatial relationships after Encore determines note spacing and attempts to align symbols vertically following the opening of a file. The slurs, dynamic indications and hairpin in the clarinet part have not retained their original spatial relationships to the notes that they are associated with. Accidental spacings are also not handled correctly, as can be seen by the second note in the bassoon part which is preceded by a flat which is placed on the stem of the first note. Notation software cannot rely on algorithms to correctly format symbols when loading a score from a file. File storage schemes should store the absolute positions of symbols or calculate symbol offsets from absolute reference points.

Example 3.1 Spatial dependencies in Encore  
Andrew Eales : *The Dark Stream* 3rd mov. measures 1-2

Despite the well-designed user interface, Encore is not suitable for typesetting scores of even moderate complexity.

### 3.2.3 MUSICA

MUSICA is a music notation package developed by the computer music laboratory of the Faculty of Electrical Engineering at the Israel Institute of Technology. The latest, and possibly last version of the software is version three. Of particular interest is version two, which was made available with the source code and limited design documentation. From the design documentation a partial object model can be constructed. A discussion of this object model and the design of MUSICA is deferred until the end of chapter four, where a useful comparison to the object model of the OOTMN can be made.

### 3.2.4 Copyist

Sion Software's Copyist does not support MIDI playback or recording. It is a typesetting program, that uses graphical symbols having no relationship to each other. A symbol editor that allows users to add user-defined symbols to the software provides the user with almost unlimited visual possibilities. Example 3.2 is an example of a complex score created with Copyist :

Example 3.2 Copyist example score - Debussy, *Reflets dans l'eau*.

The OOTMN provides a graphics mode that ignores context-sensitive symbolic relationships, allowing scores to be created as collages of symbols. Activating graphics mode destroys existing symbolic relationships, allowing the software to provide similar functionality to Copyist. A score created by OOTMN graphics cannot be converted into an equivalent context-sensitive representation.

The OOTMN attempts to make use of the expressive power of graphic symbols as well as a level of context-sensitivity that allows MIDI realisation of a score.

### 3.2.5 Other Notation Programs

The above discussion omits mention of many notation programs that are available, by focusing on three programs that illustrate three broad categories of software design. Important developments in the evolution of Windows music software include the current development of Sibelius by Sibelius-Software [Sibelius-Software, 1998], and Igor by Technoligor [Technoligor, 1998]. Sibelius has an excellent reputation [Sibelius-Software, 1998], but currently only runs on the Acorn Archimedes RISC architecture. A Windows version is scheduled to be released in September 1998. Technoligor [Technoligor, 1998], provides a detailed comparison between Igor and Finale and claims that Igor

will be the most sophisticated notation software available, and at the same time extremely user-friendly.

Many shareware and freeware packages are available via the Internet. Unfortunately, most of these programs have limited functionality and are not suitable for professional use. A notation program that may be important in the future development of notation software deserves mention. RoseGarden [Cannam, 1997] is a system consisting of a free notation editor and separate sequencing program that are able to communicate with each other [Computer Music Journal, 1997]. Originally written for machines running UNIX, the alpha Windows version 1.0 is difficult to evaluate as it is obviously not complete. The concept of separate notation and sequencing programs that are able to communicate with each other is a development that may well become a standard practice in the future.

### 3.3 Sequencers and Educational Software

MIDI Sequencers are programs that implement multi-track recording facilities using MIDI. The first Windows sequencers provided a numeric display of MIDI data, or graphic displays where notes are represented as horizontal bars. Length of a graphic bar denotes duration, vertical positioning represents the pitch by means of the position relative to the corresponding key on the keyboard. Figure 3.3 illustrates this 'piano roll' notation which derives its name from the paper rolls used to

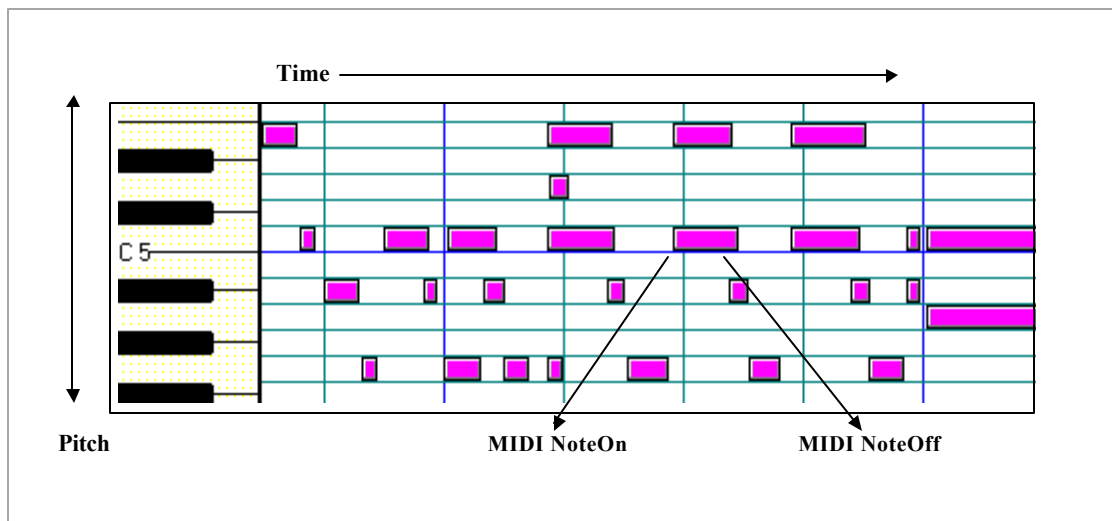


Figure 3.3 Piano Roll Notation

store pianola performances. Piano roll notation avoids the problems encountered with staff notation by reducing both pitch and durational information to a single symbol notated on a two-dimensional graph. Unfortunately, all semantic connotations associated with traditional notes are lost in the process, as MIDI data does not provide sufficient information to accurately represent staff notation. MIDI does not differentiate between enharmonic spellings of pitches which have different semantic connotations in music theory as each key of the keyboard is represented by an integer. A note's starting and ending times are used to determine MIDI durations, no representation of the actual notated note duration exists. Notated staccato crotchets performed as quavers separated by quaver rests cannot be unambiguously represented by MIDI data. Many sequencers such as Cakewalk from Twelve-Tone systems, and MasterTracks from Passport Designs have implemented notation windows that have limited editing and formatting capabilities. Future software releases will no doubt integrate sequencer and notation software into an environment that is not wholly dependent on MIDI data allowing greater control over notational layouts.

Educational products such as the MIBAC Music Theory lessons from MIBAC Corporation implement music notation as fixed graphical symbols represented by Windows bitmaps. Playback is accomplished by associated MIDI or Wave files. Software that implements notation as fixed graphical images allows no interaction from the user. Exercises are restricted to drills that are of a multiple-choice or yes/no format. Exercises that allow the creation of answers, such as the construction of a scale by placing notes and accidentals at their correct positions are of greater instructional value. Users may also correct only those portions of an exercise that are incorrect. Chapter six illustrates the use of the OOTMN to implement a harmony tutor that allows the user to interactively complete the given exercises.

### 3.4 Software tools for Music Notation

At the start of this study in 1996 no software tools that supported the creation of music notation were available. The only software that was available was a limited set of command-line tools that were able to read and write NIFF (Notation Interchange File Format) scores [Grande, 1996]. NIFF is an emerging standard for the binary representation of musical scores. In early 1998 the Notation Engine, a class library in C++ that has identical goals to the OOTMN appeared.

### 3.4.1 Notation Interchange File Format

NIFF is a standard file format for representing music notation data. Development of NIFF began in 1994 when six leading music notation software vendors agreed to develop a standard file format for representing music notation data. In January 1995, one of the co-sponsors, Coda Music Technology withdrew from the project and announced that they would publish information concerning the different file formats used by Finale. Coda expressed concern that the NIFF project was behind schedule and doubted the ability of NIFF to provide adequate representation of Finale scores. At the beginning of 1998 Coda started to provide information concerning its proprietary file formats.

[Coda Music Technology, 1998]

The proposed NIFF standard emphasises efficient data storage by using representations that require the fewest number of bits. Consecutive integers represent durations, rather than a more intuitive representation such as reciprocal representation used by the DARMS encoding scheme described by Brinkman [Brinkman, 1990] shown in figure 3.4. A trade-off occurs between a representation that is semantically meaningful, and a representation that efficiently utilises computer storage.

Note	Reciprocal Duration	NIFF Duration
♪	2	3
♪	4	4
♪	8	5

Figure 3.4 Note duration encoding schemes.

The current version of NIFF, version 6a appeared in August 1995, and is available from the Internet web site <http://mistral.ere.umontreal.ca:80/~belkina/NIFF.doc.html> (note that the ftp site [blackbox.cartah.washington.edu](http://blackbox.cartah.washington.edu) that appears in the NIFF documentation, and is found at some Internet sites is no longer used for the NIFF project).

### 3.4.2 The Notation Engine

The Notation Engine is a C++ class library that facilitates the development of applications using music notation in the Microsoft Windows or Apple Macintosh environments. The developer, Mark Walsen [Walsen, 1998] describes the design of the library :

“The Notation Engine has been designed from the start as a library with a clean separation from application user interface code. Yet, the Notation Engine also performs many user interface related tasks, such as: hit detection of objects in a window; selection and highlighting of objects; and direct manipulation of objects, such as dragging the shape of hairpin crescendo marks.”

The above quotation could easily have appeared in the introduction to this study. The Notation engine currently supports full or partial implementations of the following features :

- i) makes use of a proprietary True-Type font.
- ii) hit detection of notation objects.
- iii) management of currently selected objects.
- iv) highlighting of selected objects.
- v) functions that add, delete, and update the properties of objects.
- vi) recording and playback of MIDI.
- vii) importing and exporting of MIDI Files.
- viii) transcription of MIDI to Notation.
- ix) printing of scores.

Notation software supplies the notation engine with source code. It would be most interesting to undertake a detailed comparison of both the design and implementation techniques used in the Notation Engine and the OOTMN. Unfortunately, the high cost of the Notation Engine excludes an evaluation of the product at the present time.

### 3.4.3 Music Fonts

Symbols used in music notation may be drawn using existing graphics primitives, designed as bitmaps, or make use of current TrueType or Postscript font technology. Unlike bitmaps, TrueType

and Postscript fonts have the advantage of being scaleable. TrueType has become the defacto standard under the Microsoft Windows environment with software vendors developing their own proprietary music fonts. TrueType and Postscript technologies offer similar performance characteristics. [McGowen, 1993]. TrueType fonts can distort or disappear at certain reductions, depending on the printer hardware, while Postscript fonts require a font manager, such as the Adobe Type Manager from Adobe Systems to download fonts to the printer. Postscript fonts can create a situation where complex scores result in printer memory overflows during the downloading process.

Designing a True-type music font is a tedious, rather than a difficult process that requires expensive commercial software. According to Petzold [Petzold, 1992] a font has no copyright, only the font name may be copyrighted. Petzold points out that to avoid copyright infringements, Microsoft has had to rename the traditional typefaces found in its software products. As an example, the Times Roman font appears as MS Sans Serif in Microsoft software. True-Type fonts in the Microsoft Windows environment usually use the standardised ASCII character set and not the OEM standard for character representation. No standard exists for defining a set of music symbols, or for the mapping of symbols to the ASCII character set. Despite the lack of standardisation, different fonts often make use of the same ASCII symbols to represent music symbols in a logical manner. Some ASCII symbols suggest a visual relationship to music symbols as shown in figure 3.5 :

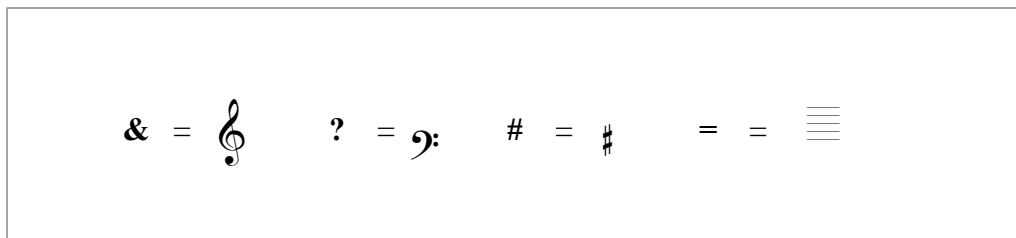


Figure 3.5 Relating ASCII symbols to music symbols.

Other symbols are represented by alphabetic characters that suggest the name of a symbol such as 'q' for quarter-note and 'h' for half-note. Uppercase and lowercase letters represent the note's stem direction as illustrated by figure 3.6 :





Figure 3.6 Encoding notes in a music font.

Huron [Huron, 1992] describes these pictorial and alphabetic mappings between symbols as pictorial mnemonics and initialisms.

The metrics (i.e. the proportional characteristics) of a font are important, as these values control the placement and alignment of symbols. Commercial music fonts align characters on the left-hand top or bottom positions of the area occupied by a symbol. Alignment characteristics differ from font to font and may even differ for different symbols within a single font. A standard governing the placement and alignment of characters would greatly simplify the programming of applications that make use of music fonts.

### 3.5 Object-Oriented Music Representation

A clear distinction should be drawn between representations of musical notation such as the OOTMN and the Notation Engine, and representations of musical structure such as the MUSIKUS system [Lande, 1995]. MUSIKUS encodes the musical structure of a work in purely musical terms, without providing any information regarding the visual notation of the work. Pope [Pope, 1996] discusses different object-oriented music representations, some of which provide various types of graphical representations of a score's structure. Pope points out that graphical representations, including music notation are an application dependent characteristic that are not part of the musical representation itself. i.e. the graphical symbols forming the notation of a score are determined by an underlying machine representation. This characteristic is discussed in chapters five and six. The next chapter discusses the design of an object-oriented representation of music notation that is centred around the relationships that exist between different music notation symbols.

## Chapter 4

# Designing an Object-Oriented Toolkit for Music Notation

### 4.1 Object-Oriented Development Methods

During the nineteen-nineties a variety of methods have emerged that can be used for the analysis and design of object-oriented software. The object paradigm is generally regarded as an alternative, superior approach to the traditional software engineering methods of functional decomposition, or 'top-down' structured development [Stroustrup, 1988]. Different methodologies emphasise different aspects of the system, and also have different notational conventions. The Unified Modelling Language (UML) is a welcome attempt to standardise the graphic notation used in object-oriented modelling. Developed by Jim Rumbaugh, Grady Booch, and Ivar Jacobson [Project Technology, 1997], UML places emphasis on the design and implementation of the design, rather than the conventions used to produce a design. A valuable discussion of various approaches and methods used in object-oriented development can be found in Carmichael [Carmichael, 1994]. Especially interesting is the contribution by Hodgson [Carmichael, 1994] that attempts to place different development methods in perspective by considering the meaning of object-oriented development, rather than the characteristics of individual development methodologies.

Different approaches to software development within an object-oriented framework are found by viewing the system from different perspectives. System design can be governed by three different points of departure; an event driven approach, a structural approach, and an approach that emphasises the data within the system. These distinctions do not exclude the use of a particular method, as all object models consist of objects, as well as the methods and data that they encapsulate. Rumbaugh [Rumbaugh, 1991] combines different views of a system by means of different models that relate an object model to data-flow and functional representations of the system.

An event-driven view derives objects from the grouping of external events and system responses. This approach has its origins in methods developed for real-time software development such as the method developed by Ward and Mellor [Ward and Mellor, 1985]. Objects, by their very nature, lend themselves to this type of development; as a single object can be seen as a closed system that communicates with the outside world by sending and receiving messages. Structural approaches proceed by first identifying the objects in a system; followed by the data requirements of each object i.e. its attributes, and the methods that manipulate object data. This type of development is advocated by Coad and Yourdon [Coad, 1991], Coad [Coad, 1997], and Booch [Booch, 1991]. Data-driven designs are typically found in database systems that emphasise the logical grouping of data to form objects that encapsulate related data. Viewed from the object level, event driven and data-driven approaches proceed as a bottom-up process, while a structural approach proceeds top-down.

#### **4.1.1 Selecting a Method**

Choice of a particular method depends mainly on the level of detail required at both the analysis and design stages. The level of detail is governed by the development environment, as well as the problem domain. Many design methods are intended to be used for the most complex applications, where extremely detailed models, as well as documentation that facilitates communication during development is desirable. Smaller systems do not require such detailed specification, and may require little communication between members of a small development team.

One of the main features of successful development is understanding what is to be done, as well as how to do it. A level of detail that is not appropriate to the domain and development environment is likely to hinder development. A model that lacks required information cannot provide sufficient understanding, while unnecessary detail only serves to add complexity to the system. Experience gained during the design of the OOTMN indicated that the process of design should be a fluid process - rigorous, yet flexible. Design procedures should strike a delicate balance between these two aspects. The former providing clarity and focus, while the latter allowing changes to be easily made when necessary. The Coad/Yourdon Analysis Method [Coad, 1991], and its successor developed by Coad [Coad, 1997] are development methods that emphasise clarity and simplicity.

These methods allow a clear, yet rigorous design to be developed without sacrificing necessary detail or flexibility.

## 4.2 Object-Oriented Analysis and Design

Many object-oriented development methods clearly separate analysis and design. The Coad/Yourdon method covers object-oriented design in a separate volume [Coad, 1992]. Later work by Coad [Coad, 1997] abandons a rigid separation of the analysis and design phases, treating these two aspects as complementary, alternating processes.

### 4.2.1 The Coad Method

Coad recommends that the purpose of the system be clearly identified before embarking on the design. The following defines the purpose of the OOTMN as conceived at the start of the project :

“To design an object-oriented library for music notation that accurately captures the hierarchical ordering of relationships between music symbols.”

At a later point when design and implementation were proceeding simultaneously and a greater insight into system objects and their relationships had been achieved the purpose of the system was expanded to include a definition of system functionality :

"To design an object-oriented library for music notation that

1. accurately captures the hierarchical ordering of relationships between music symbols, and
2. preserves context -sensitive and spatial relationships between symbols and their musical meaning, but is not constrained by these relationships."

Design decisions during software development should attempt to capture the elementary functionality of objects, as well as anticipating less common situations. Although it is common to consider software functionality within the context of its most common usage, this context should not be so constrained that it does not allow the software to be used in other ways. A trade-off occurs between completeness of implementation within a limited context, and the ability to be useful within a wider context. Object-oriented technology specifically emphasises completeness of

implementation as a process that develops from a generalised, limited implementation to specific, detailed implementations.

Objects and object relationships are represented by diagrams that use a specific notation. Objects are represented by a box that contains the object's name, attributes and methods (functions) :

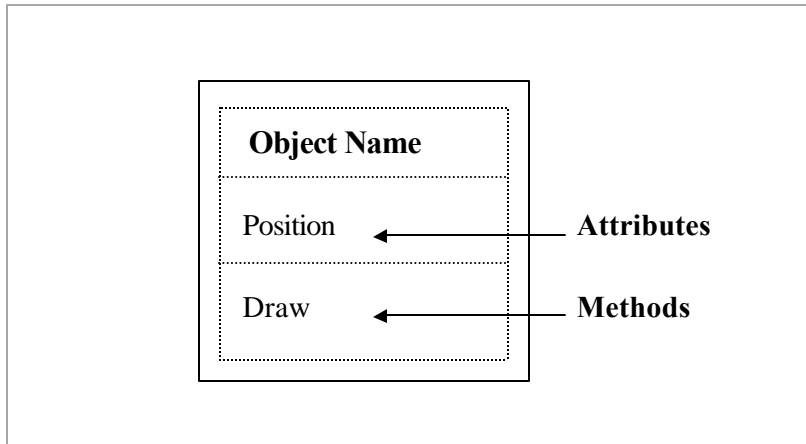




Figure 4.1 An Object and its attributes and methods.

The symbol  represents specialisation. For example, a class Table is a specialisation of the more general class Furniture. The symbol  represents a whole-part structure. For example, a Wheel object is a part of a Bicycle object. In the examples below, object B is a specialisation of object A, and object D is a part of object C.

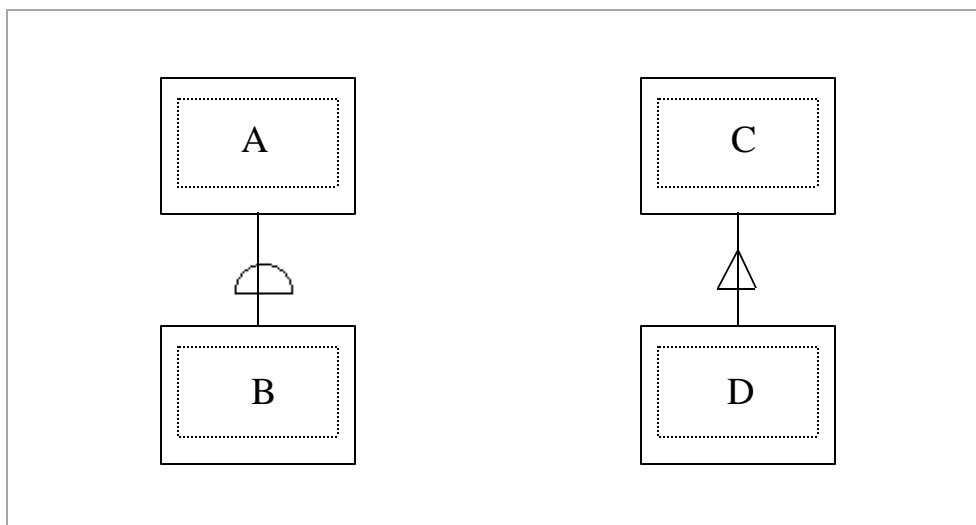


Figure 4.2 Generalisation-specialisation, and whole-part relationships.

A line between two objects indicates an instance relationship. Instance relationships indicate a ‘knowledge of’ relationship between two objects. For example, an Accidental is not part of a Note, but influences a note. Viewed another way, the Accidental is ‘visible’ to the Note object. All relationships also have a cardinality, as shown by the numerals used in figure 4.3 :

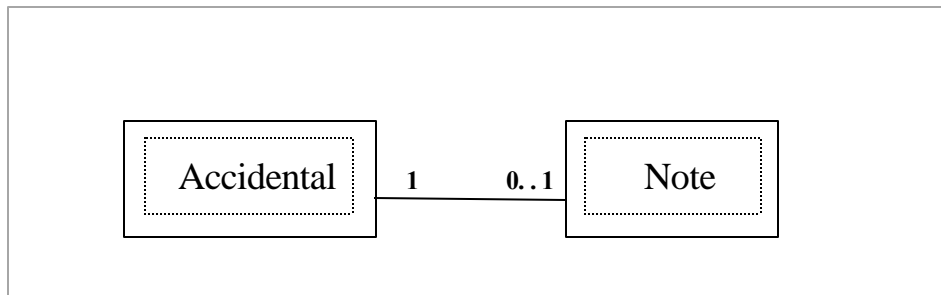


Figure 4.3 An instance relationship and its cardinality.

A note is related to zero or one accidentals, while an accidental must have a relationship to a single note. i.e. accidentals cannot exist on their own.

### 4.3 Identifying objects and their relationships

Determining what objects should be represented in a system, as well as the relationships between objects is crucial to a successful design. The process of finding objects ranged from identifying trivially obvious objects, to considering less obvious objects that were sometimes later discarded as their implications for the system as a whole were comprehended. Groups of objects may have structural relationships that can be represented in different ways. These different possibilities must be evaluated to obtain the most useful representation, as well as the representation that most faithfully captures relationships inherent in the subject matter being modelled.

The descriptions that follow are limited to the core objects and their attributes identified during object analysis. For the sake of brevity and clarity, many attributes that are self-explanatory are not discussed. The description progresses from examining elementary object attributes and their relationships to more complex interactions, mimicking the evolution of the design.

### 4.3.1 Scores, Pages and Score Templates

Using the conventions of the printed score as the basis for design, a Score object forms the highest level of the object hierarchy. A Score is an aggregation of Pages, and also holds information which is required by objects at lower levels of the object hierarchy. Attributes within the Score object are usually global attributes that are required at different levels of the hierarchy. Details that influence specific lower-level objects are stored in a ScoreTemplate object which defines the structure of a score. The relationship between Score, ScoreTemplate and Page objects is shown in figure 4.4.

The music font attribute is the name of the MS Windows font used by the score. It is assumed that the font is installed in Windows, and is available to the OOTMN. MetronomeState and AutospaceState are switches that turn the MIDI metronome on or off and determine whether notes and rests must be automatically spaced or not. The other attributes are global values that are used as default values at lower levels of the object hierarchy. Score attributes define global variables (corresponding to musical abstractions) that can be used to create appropriate objects where required within the object hierarchy. Attributes such as key and time-signatures should not be confused with objects having the same name. The “key” or “time” of a musical work defines, but does not represent the actual key and time-signatures found in a score. Left and right margins are the default margins for all pages, staff size is the default spacing between the lines of a staff. MIDI resolution is the number of clock ticks per beat. MIDI implementation and MIDI clocks are discussed in chapter five. ScoreState is a value used by methods that search for objects and update the positions of objects. These methods are explained in detail in the next chapter. Transpose is a method that transposes a set of notes by a specified interval. The remaining methods are self-explanatory.

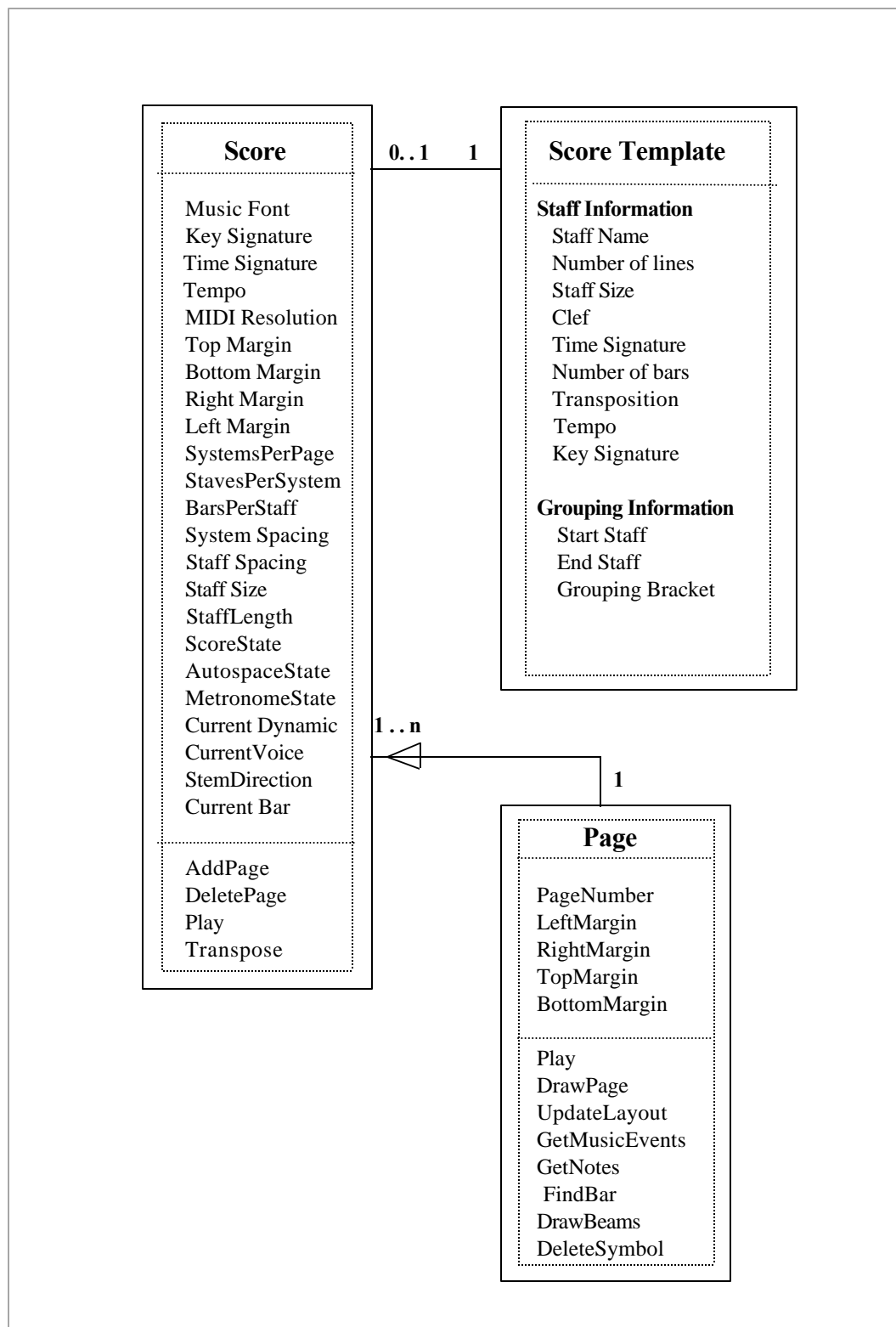


Figure 4.4 Score, Score Template and Page Objects.



A ScoreTemplate object holds detailed information for each staff found within a system. This information is used when creating the staff throughout the score. Scores that are not template-based, make use of global Score attributes when creating Staff objects. Specifying the time-signature and number of bars for each staff may appear to be unusual, but is included to allow the future development of multi-metric music. i.e. music that consists of different, metric strands occurring simultaneously. Grouping information allows staves to be visually grouped using square brackets or braces. Consider the following example of a template for a work written for violin, Bb clarinet, tambourine, and piano in F major. The score template would be :

Staff Information					
Staff Name	Violin	Clarinet	Tambourine	Piano	Piano
Number of lines	5 (default)	5	1	5	5
Staff Size	----- default -----				
Clef	Treble	Treble	none	Treble	Bass
Key Signature	F maj	G maj	none	F maj	F maj
Time Signature					
Number of bars	----- global value from Score -----				
Transposition	none	Maj 2nd down	none	none	none
Tempo	----- global value from Score -----				
Grouping Information					
Start Staff	Piano / RH				
End Staff	Piano / LH				
Grouping Bracket	Brace				

Figure 4.5 Example of a score template.

Computer representations have the ability to distort the traditional page-oriented format of a score by providing virtual views that are not bounded by the physical display. A score may scroll horizontally or vertically with intermediate positions that destroy traditional conceptual divisions provided by the page. It is often extremely difficult for the trained composer or performer to visualise larger sections

of a work without page divisions. Sequencer software packages that have traditionally used graphic methods to represent music commonly make use of virtual linear views. Linear views are not appropriate to the subject matter of music notation, and may be used as an option, but should never replace a page-oriented ordering scheme.

### **4.3.2 Systems and Staves**

Traditionally, all staves occurring within a system are aligned vertically at their starting and ending points. In works such as Lutoslawski's *Dance Preludes*, shown in example 4.1, the composer notates staves only when required for each instrument. i.e. bars that only contain rests are left blank. To ensure that scores such as this example can be represented, great care must be taken not to constrain relationships between objects. For example, a system that requires its constituent staves to start at the same x co-ordinate, does not allow the starting position of individual staves to be adjusted. Such a design constraint excludes representation of the score of example 4.1. System objects serve to provide a semantic, and visual association between their constituent staves. This visual association traditionally groups staves by means of a system line and bracket. Utilisation of a different visual arrangement does not undermine the semantic integrity of a System object. It is interesting to note the unusual positioning of time-signatures in this example, a feature supported by the OOTMN which is discussed in section 4.3.4 and illustrated in chapter six.

**DANCE PRELUDES • TÄNZERISCHE PRÄLUDIEN**

**Witold Lutosławski**  
(1955)  
(3rd version, 1959)

**I**

Allegro molto (♩=168-176)

The image shows a page of a musical score. At the top, the title 'DANCE PRELUDES • TÄNZERISCHE PRÄLUDIEN' is centered. To the right, the composer's name 'Witold Lutosławski' is written, followed by '(1955)' and '(3rd version, 1959)'. Below the title, the tempo 'Allegro molto' and a metronome marking '(♩=168-176)' are given. The score is divided into two systems. The first system contains staves for Flute (fl.), Oboe (ob.), Clarinet in Sib (cl. in Sib), and Bassoon (f). The second system contains staves for Violin (vno), Viola (vla), Violoncello (vc), and Contrabass (cb.). The woodwind staves have dynamic markings of *f* and *mf*. The string staves have dynamic markings of *f* and *dim.*. Time signatures of 2/4 and 3/4 are indicated below the string staves. The notation is dense and complex, with many accidentals and slurs.

Example 4.1 Placement of Staves, Bars and Barlines - Witold Lutoslawski, *Dance Preludes*.

To ensure that scores such as the above example can be represented, great care must be taken not to constrain relationships between objects. For example, a system that requires its constituent staves to start at the same x co-ordinate, does not allow the starting position of individual staves to be adjusted. Such a design constraint excludes representation of the score of example 4.1. System objects serve to provide a semantic, and visual association between their constituent staves.

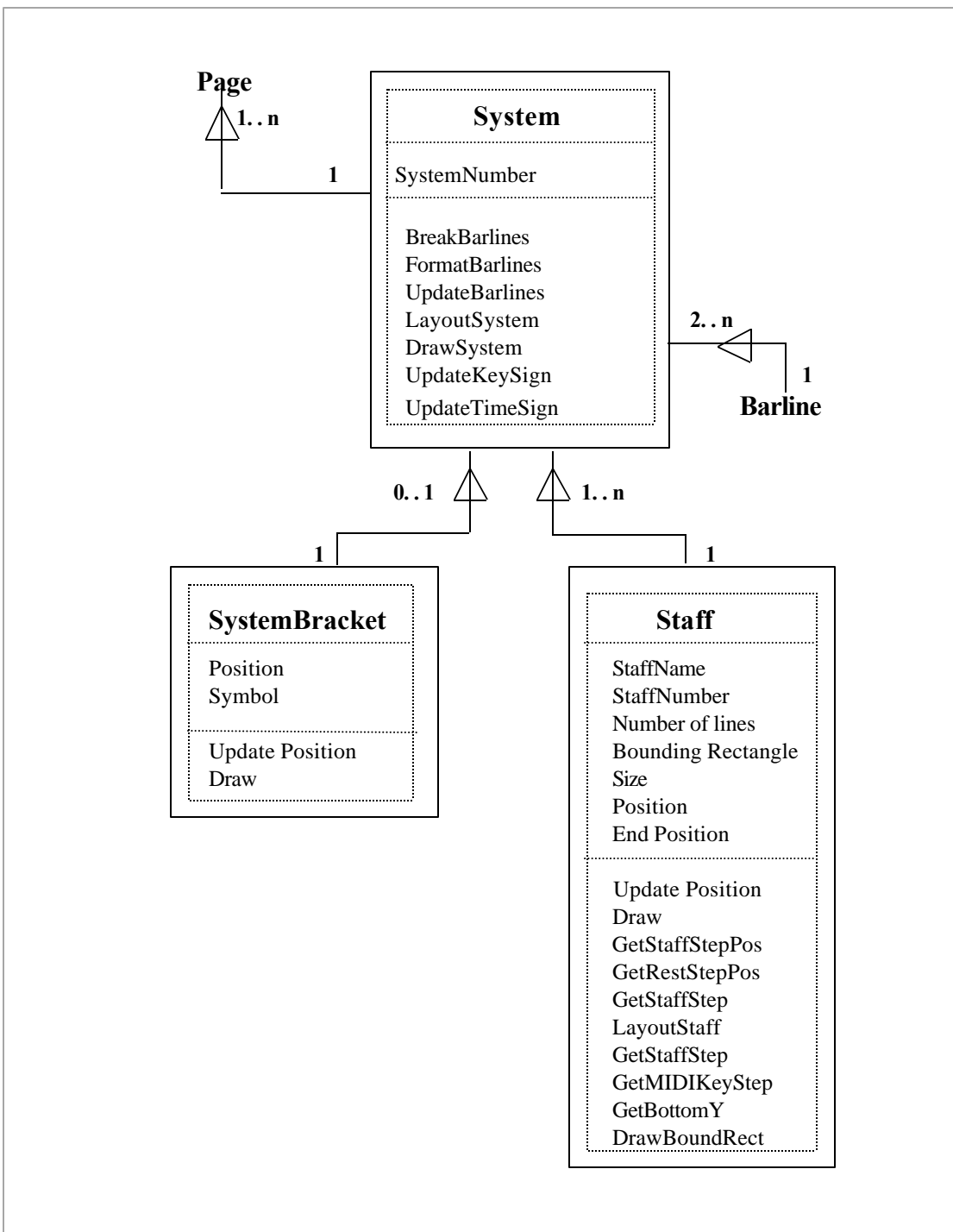


Figure 4.6 System, SystemBracket and Staff objects.

A system bracket that consists of a line, and either a square or curly bracket enhances the visual grouping of staves within a system. Semantically, all events occurring within a vertical time-slice are

performed simultaneously. Systems and staves have a number attribute. SystemNumber is the number of a System object within a Page, StaffNumber is the number of a Staff within a System. Numbers start at the top of a Page or System and proceed downwards. A Staff's end position attribute gives the length of each staff. To provide maximum flexibility none of the positional attributes of a staff are constrained in any way. The bounding rectangle attribute defines the actual vertical starting and ending positions of each Staff. MusicSymbol objects falling within this rectangle are related to a single Staff. FormatBarlines ensures that all barlines are correctly aligned. LayoutSystem divides the x-axis of System by the number of bars and places Barlines at appropriate x co-ordinates. BreakBarlines allows barlines to be broken within a single system. UpdateKeySign and UpdateTimeSign set the key and time-signature for all staves belonging to the System. GetStaffStepPos and GetStaffStep translate a staff step to a screen position and vice-versa relating pitch information to staff positions and screen co-ordinates. GetMIDIKeyStep determines the MIDI key for a given staff step.

### **4.3.3 Bars and Barlines**

The relationship between staves, bars and barlines presented some of the most challenging problems during the development of the OOTMN. A bar has no position of its own, the left and right barlines associated with each bar give the starting and ending positions of the bar on the x-axis. Bars have two different bar numbers, a local bar number that indicate the bar's order within the staff on which it occurs, and a global bar number representing its number from the start of the score. Global numbers are required to locate unique bars, and to facilitate playback. The duration attribute specifies the total duration of the bar, which can be used to ensure that a bar does not contain more notes and rests than allowed by the time-signature.

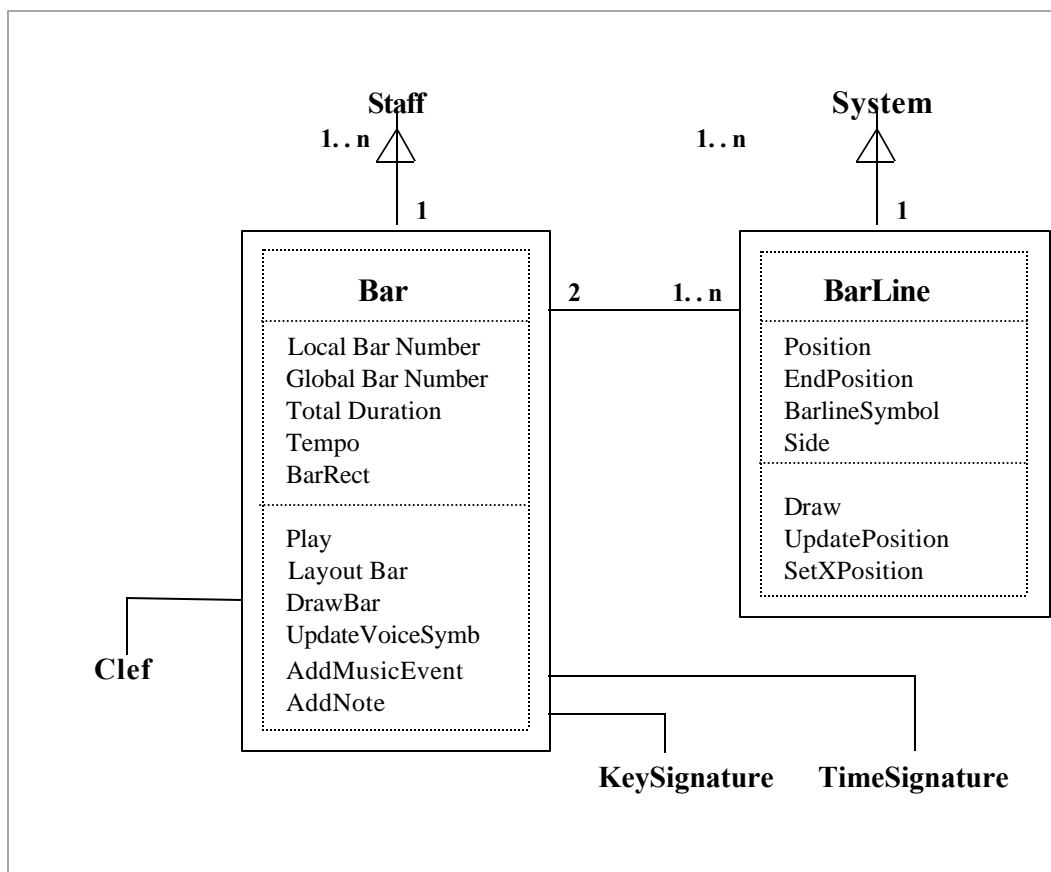
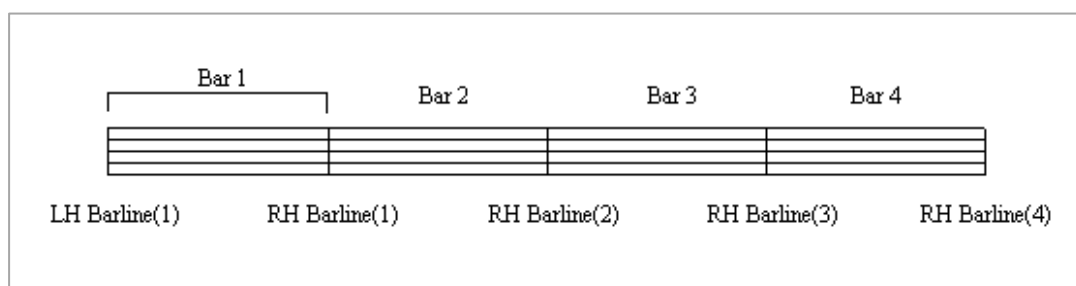


Figure 4.7 Bar and Barline objects.

A tempo attribute allows tempo changes to occur on bar boundaries. Initially, each bar was time-stamped according to the NIFF standard with a time-tag indicating the total elapsed time from the start of the score. During MIDI implementation, it was found that this information was not required to correctly implement timing, and posed certain limitations on the software. Implementation of MIDI timing is discussed in greater detail in chapter five. Instance connections between a Bar and its Clef, Key and TimeSignature objects are discussed in section 4.3.4. LayoutBar formats all notes and rests within a bar using a simple positional formula derived from rhythmic durations.

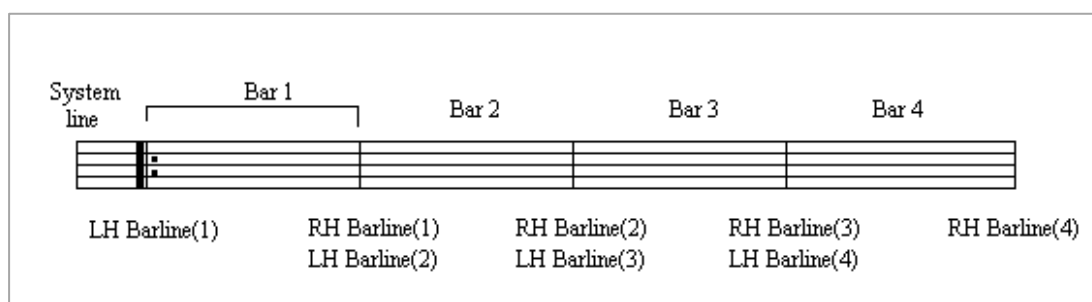
By default, a single barline object represents both the left and right-hand barlines of adjacent bars. A flexible design must, however, allow different barlines to occupy bar boundaries where required by the score. Bars can divide a single staff equally as shown in example 4.2. Left and right-hand

By default, a single barline object represents both the left and right-hand barlines of adjacent bars. A flexible design must, however, allow different barlines to occupy bar boundaries where required by the score. Bars can divide a single staff equally as shown in example 4.2. Left and right-hand barlines belonging to adjacent bars overlap each other by sharing the same horizontal position. This arrangement places the starting position of the first bar at the start of the staff. The key-signature, time-signature and clef that are traditionally notated at the start of each staff form part of the first bar of the staff.



Example 4.2 A single staff divided into bars.

This scheme is intuitively appealing, as all bars may denote key, time, and clef changes at the start of a bar. However, the addition of a repeat barline as the left-hand barline of the first bar in example 4.3 suggests that the portion of a staff to the left of the left-hand barline is not part of the bar itself :



Example 4.3 Starting position of the first bar of a staff.

The line at the left-hand starting position of a staff is not a barline (as assumed in example 4.2), but a system line that denotes the start of a system. This is further illustrated by example 4.4, where the clef, key and time-signatures are behind the left-hand first barline of the first bar. Changes that

occur in the first and third bars are in front of the left-hand barline. In practice, the key change in the first bar would usually be notated at the end of the previous staff, avoiding the redundancy of the key-signature at the start of the staff. Although unusual, example 4.4 is not theoretically incorrect.

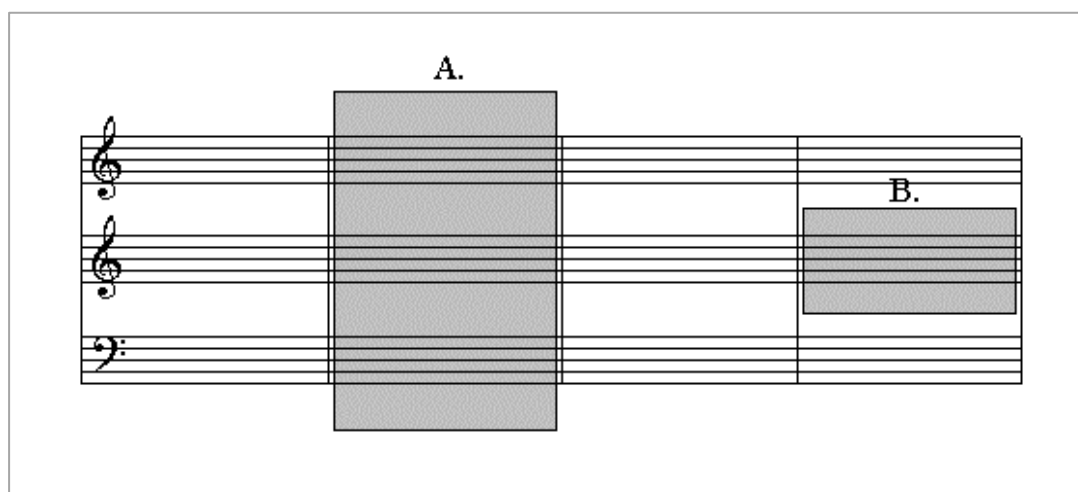


Example 4.4 Barline positions of the first bar of a staff.

Clef and signature changes at the start of a staff belong to the staff and not to the first bar, as the area between the system line and the start of the first bar belongs to the staff. If this area is interpreted as belonging to a bar, a distinction must be drawn between the first bar of a staff and all other bars, resulting in two distinct bar objects. Such a representation is more complex than necessary and should be discarded in favour of a single bar object representation. This scheme also distorts the meaning of a staff, denying that a staff may have clef, time and key-signature attributes.

#### 4.3.3.1 Bar Representation

The concept of a bar can have two interpretations as illustrated in example 4.5. A bar can encompass either the vertical area across all staves forming a system (area A), or the area within a single staff (area B). Both of these areas are bounded by barlines.



Example 4.5 Two representations of a bar.



English language reference to a bar across multiple staves is simply achieved by use of the bar number. Reference to a bar within a specific staff qualifies the bar number with the staff number or instrument name, providing an unambiguous reference to either area A or area B. The terms ‘system- bar’ and ‘staff-bar’ provide a convenient distinction, the former referring to area A, and the latter to area B.

Defining system-bars provides a vertical grouping of events which may be useful should the parsing of events during playback be performed on each bar in turn. All notes occurring at a given metric division of the system are easily accessible. Barlines that traverse the entire system such as the barlines of example 4.5 logically flow from system-bars. Unfortunately, system-bars create problems regarding the constituent parts of a staff, as well as the relationships between barlines and a bar. If a system consists of staves, a system-bar consists of vertically adjacent areas of multiple staves, requiring that a staff be constrained to occur within a bar. Such a definition of a staff does not occur within music theory, and is not conceptually satisfying. Blank manuscript paper does not consist of bars which must be filled in with staves, the bar is always subordinate to the staff. Barlines are often not contiguous lines across an entire system, being broken to reflect individual instruments, or groups of instruments as shown by the common piano-vocal or piano-solo instrument format of example 4.6 :

The image shows a musical score for Tenor (Ten.) and Piano. The Tenor part is written on a single staff with a treble clef, a key signature of two flats (B-flat and E-flat), and a 4/4 time signature. The Piano part is written on two staves, with a treble clef on the upper staff and a bass clef on the lower staff, sharing the same key signature and time signature. The score is divided into three measures by vertical barlines. The barlines are broken at the instrument level, meaning they do not span across the entire system of staves.

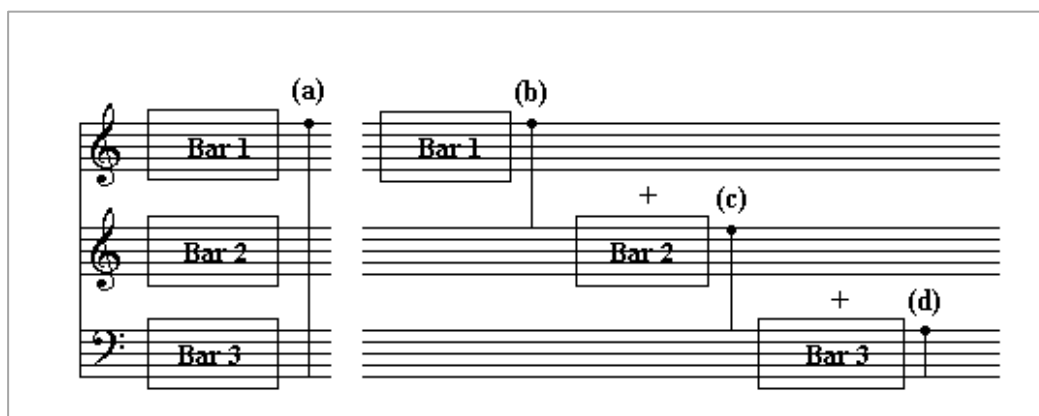
Example 4.6 Piano / vocal score format.

Where barlines are broken within a system, system-bars are delimited by multiple barlines.

System-bars implicitly require that left and right-hand barlines share x co-ordinates. Multi-metric music (music where different parts, or instruments have different time-signatures) cannot be accommodated by a system-bar representation that requires barline alignment. Different durational contents would require multiple barlines at mutually exclusive positions, destroying the metric function of the barline. Given the above limitations, bars are represented in the OOTMN as staff-bars. The use of staff-bars provides greater flexibility and is consistent with notational practice, but poses interesting problems concerning the use of barlines.

#### 4.3.3.2 Barline Relationships

Staff bars can only have a single left-hand and a single right-hand barline. Barlines that occur across multiple staves, or an entire system pose the opposite problem to the problem of breaking barlines discussed in the previous section. Barlines must be joined to encompass more than a single staff as shown in example 4.7 :



Example 4.7 System barlines created from overlapping barlines.

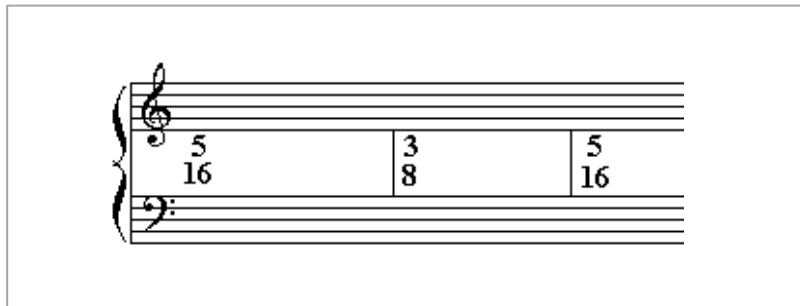
The barline at (a) extends across the entire system, and is created by the vertical alignment of the barlines at (b), (c), and (d). Note that barlines occurring on staves preceding the last staff must be extended across the space between staves. Design of the OOTMN initially followed this scheme which was also implemented in C++. The NIFF specification [Grande, C, 1995, p.46] allows a barline to have a flag that specifies its extent from a starting position at the top of a staff. This flag can indicate the starting and ending point of a barline as :

- a) the top of a staff, and the bottom of the last staff of a system, or
- b) the top of a staff, and the top line of the next staff, (as in figure 4.7b) and c), or

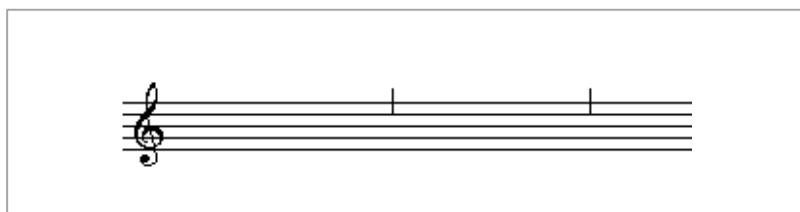
- c) the bottom of a staff, and the top of the next staff i.e. the barline occurs only in the spaces between adjacent staves (not through staves)

This specification suggests that many commercial products use the representation scheme of example 4.7 described above. Despite a successful implementation of this design, it was felt to be conceptually unsatisfactory and was abandoned.

Throughout the design process an attempt was made to place as few constraints as possible on both the placement of, and relationships between different objects. Examples from the literature such as the opening of Lutoslawski's *Dance Preludes*, Stravinsky's *Abraham and Isaac*, and the barline notations used by Luciano Berio described by Cole [Cole, 1974] require unusual spatial relationships between symbols. The following examples illustrate these unconventional notations:



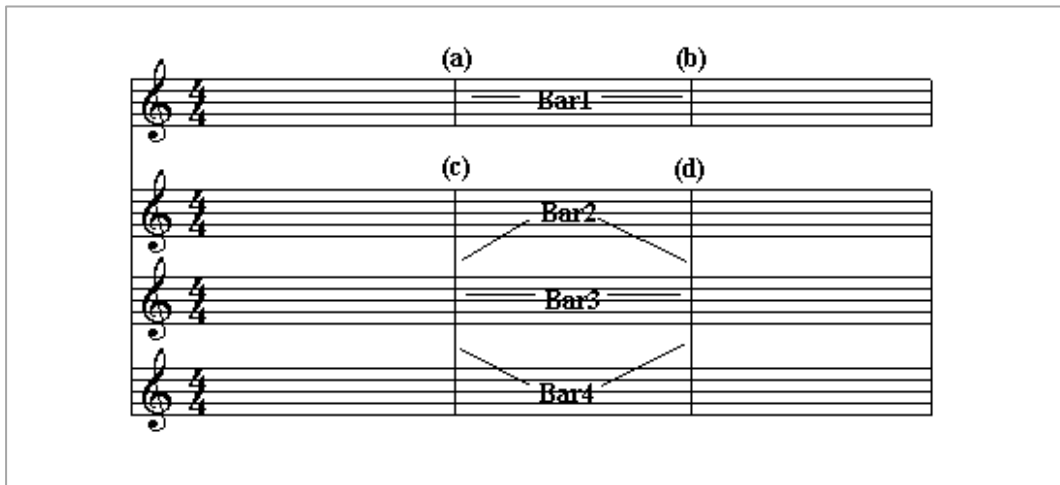
Example 4.8 Barline notation used by Igor Stravinsky in *Abraham and Isaac*.



Example 4.9 Barline notation used by Luciano Berio.

It was later realised that a bar need not have unique barlines, unless adjacent bars required different symbols for their respective left and right-hand barlines. A single barline may be shared by multiple bars occurring vertically across a system, as well as adjacent bars. The OOTMN implements

barlines as shared barlines where any barline that crosses a staff is deemed to delimit the bars occurring on that staff as shown in example 4.10 :



Example 4.10 Shared barlines used by the OOTMN.

The barlines at (a) and (b) determine the starting and ending positions of Bar1 and extend across a single staff. Barlines at (c) and (d) extend across multiple staves and are shared by bars two, three and four. This view of barlines is consistent with the musician's view of a barline as an indivisible symbol. Where barlines are broken within a system, new Barline objects are added to the system. The single system in figure 4.10 has six visible barlines within three system bars. Barline objects have no direct relationship to Staff objects. A System object stores and maintains Barlines, which are utilised by Bars to indicate the starting and ending positions of a bar.

#### 4.3.3.3 Secondary Bars

A secondary bar is an interesting concept, specially developed for the OOTMN. Secondary bars allow ornaments to be correctly notated and played by implementing in standard notation, both the notated and performed versions of the ornament. A secondary bar occupies the same position as its parent bar from which it is derived.

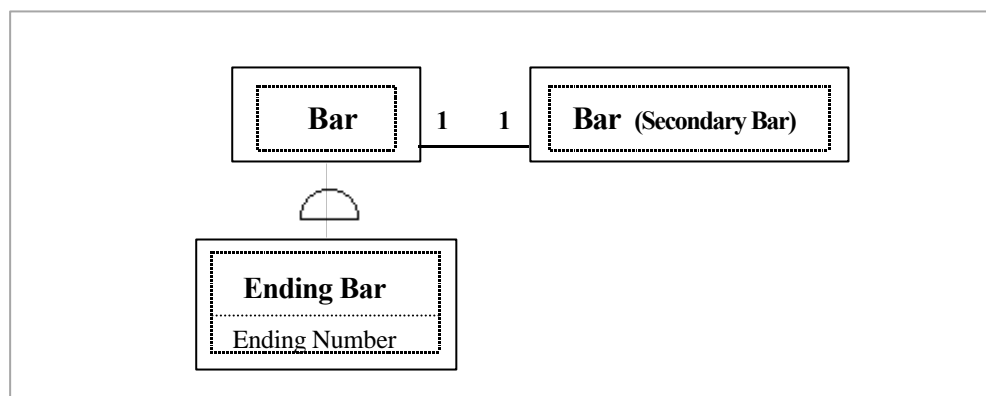


Figure 4.8 Bar specialisations and SecondaryBar objects.

Only the parent bar can be performed via MIDI, the secondary bar functions solely as a graphic representation of the ornament. This scheme allows the performance version of the ornament to be notated exactly as required. A secondary bar and its associated primary bar cannot be visible at the same time, as they occupy identical positions within a staff. Bars that have additional functionality such as first and second-time bars are derived from class Bar. The OOTMN does not presently support different types of Bar objects having specialised functions such as repeat, and first and second time bars. These specialisations can easily be derived from a successful bar representation.

#### 4.3.4 Clefs, Key-Signatures and Time-Signatures

The clef, key-signature and time-signature are by default inherited (in the English, and not the object-oriented sense of the word) by staves from the global attributes contained in the Score object. A staff may override the global settings as required by the score structure, or the changing musical context. There is, however, a crucial distinction between a clef, key-signature, or time-signature at the score and staff levels. The Clef, Key-signature and Time-signature associated with a Staff are actual objects, influenced by the corresponding score attribute information. This distinction results from the difference between a higher-level concept of “key” (such as the key of F minor) and the physical manifestation of F minor as four flats occupying specific positions on a particular staff. Similarly, a global numeric representation of a clef or time-signature does not provide any information about the placement of symbols, or any other information which may be useful to the software.

Representation of time-signatures divides the upper and lower values of a time-signature into distinct objects. This scheme allows for greater flexibility when displaying a time-signature or altering its position. Modern conventions that change the appearance of a time-signature's lower value, or display a time-signature in a different position from the conventional placement within the staff are easier to implement with a time-signature consisting of two objects. Clef, Time-signature and Key-signature objects form instance relationships with Staff and Bar objects.

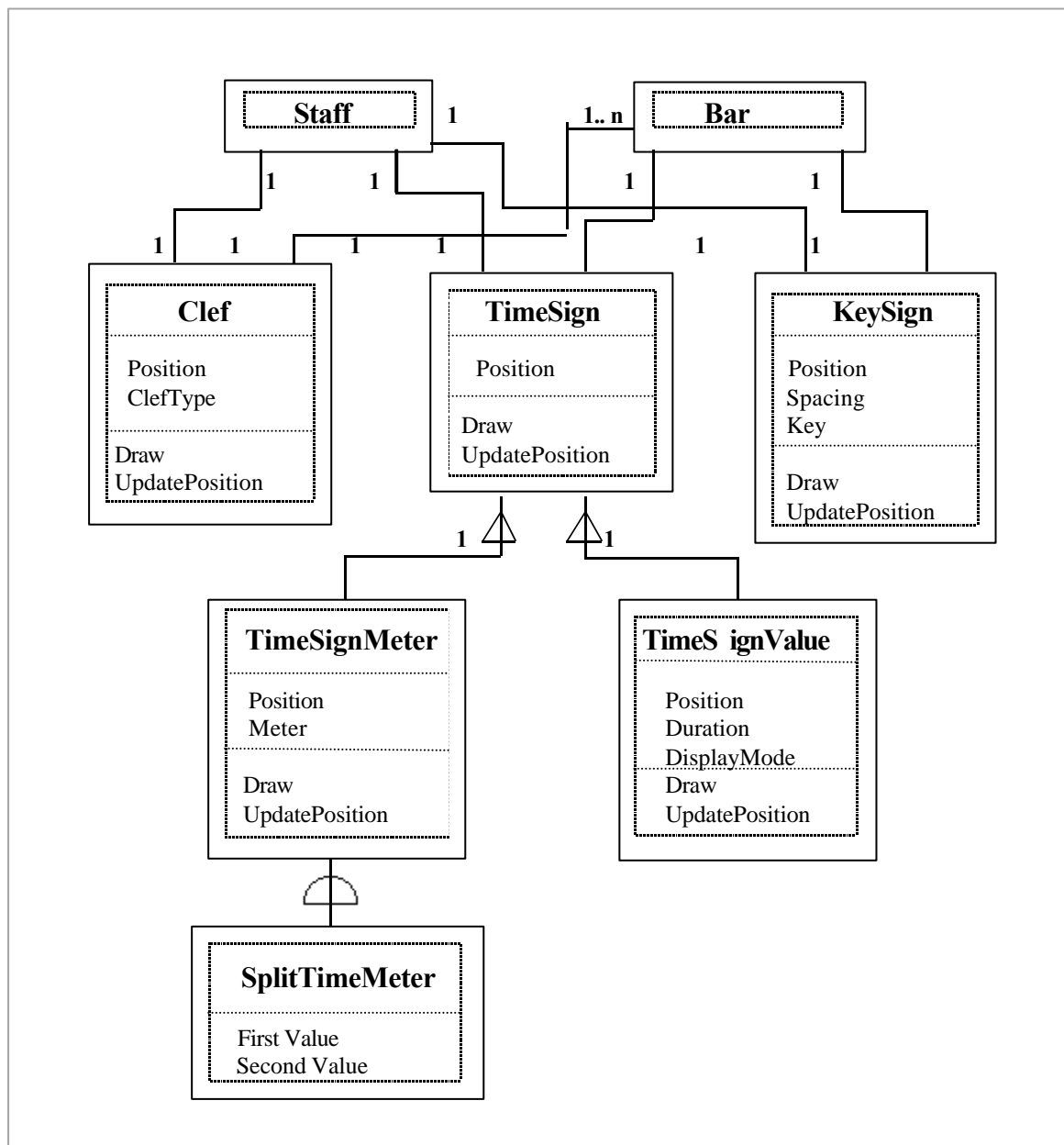
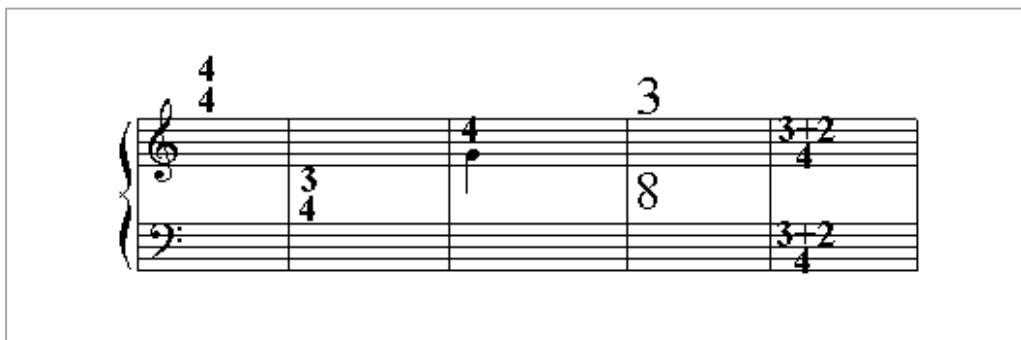


Figure 4.9 Clef, Time-signature and Key-signature objects.

Multiple clefs can be associated with a single Bar object, allowing more than one clef change to occur within the bar. All other relationships are unary relationships as a staff can only have a single clef or key-signature or time-signature. (excluding the use of changing meters discussed in chapter two). The DisplayMode attribute of class TimeSignValue determines whether a numeric or symbolic (i.e. note) representation is to be used for the lower value of a time-signature. Class SplitTimeSignMeter allows a composite representation of the form  $x + y$  to be used as the top value

of a time-signature. Example 4.11 shows modern time-signature conventions made possible by the time-signature representation of figure 4.9 :



Example 4.11 Time-signature positions in modern notation.

## 4.4 Voices

Voice objects function only as containers that separate the streams of music events that make up a musical voice within a bar. Separation of musical voices allows editing and playback of individual voices, as well as simplifying the formatting of the notes and rests within a bar. Stem directions can be set within a particular voice to provide correct visual separation between voices. Different MIDI timbres (i.e. synthesiser patches) can also be assigned to independent voices.

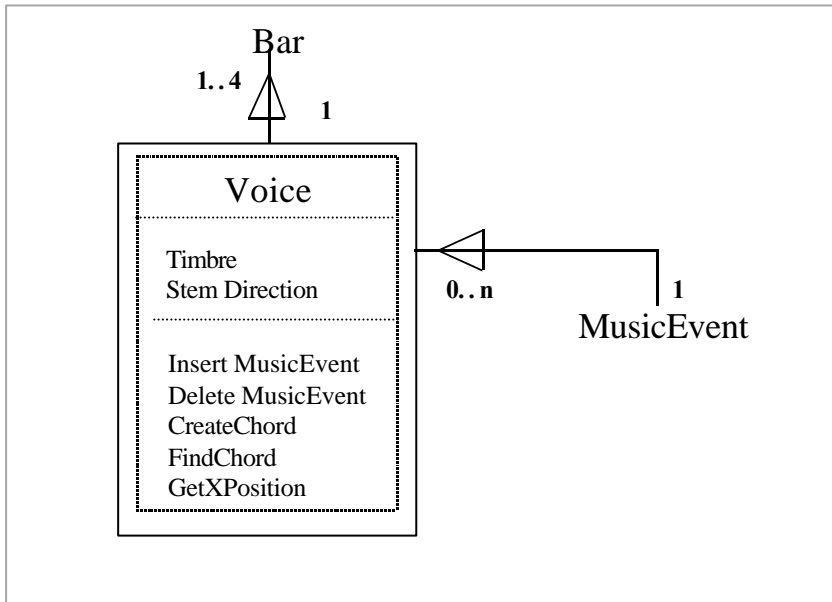


Figure 4.10 Voice objects and their relationships.

## 4.5 Note, Rest and Chord Relationships

Class MusicEvent is a class from which objects that have a sonic realisation are derived. Notes correspond to the musical concept of a tone, Rests correspond to silence. Note objects that have a starting and ending time are directly translated into MidiMessage objects. Rests are regarded as having no starting or ending time, a Rest's duration attribute only provides the starting time of the next note following the rest.

### 4.5.1 Note Structure

A note may consist of only a note-head, or a note-head with a stem, or a stemmed note with one or more flags as shown in figure 4.11 :



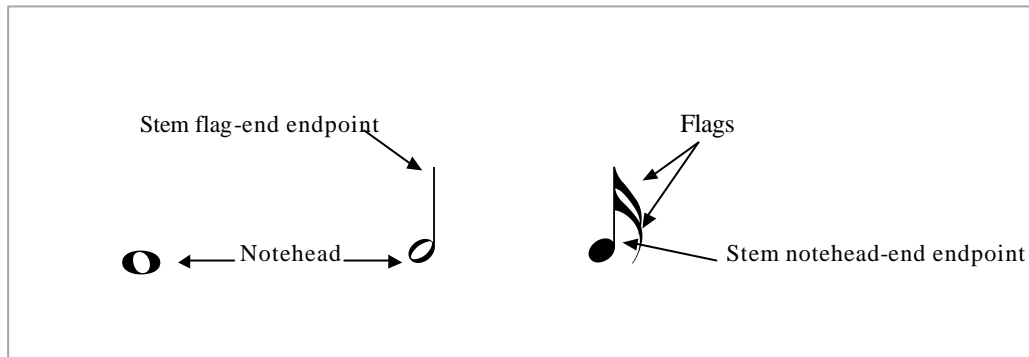


Figure 4.11 Structure of a note symbol.

An object-oriented design consistent with music theory views a Note object as a whole-part structure consisting of a notehead, one or two stems and one or more flags as shown in figure 4.12. This representation is simplified by considering the notehead as an attribute of class note, rather than a separate object. A Notehead class would have only two attributes, the symbol that represents the notehead and its position. The position attribute is made redundant by the Note object having a position attribute. Single attribute classes often indicate a redundancy where the class as a whole can be represented as an attribute rather than an object, hence we view notehead as an attribute of Note and not as a class.

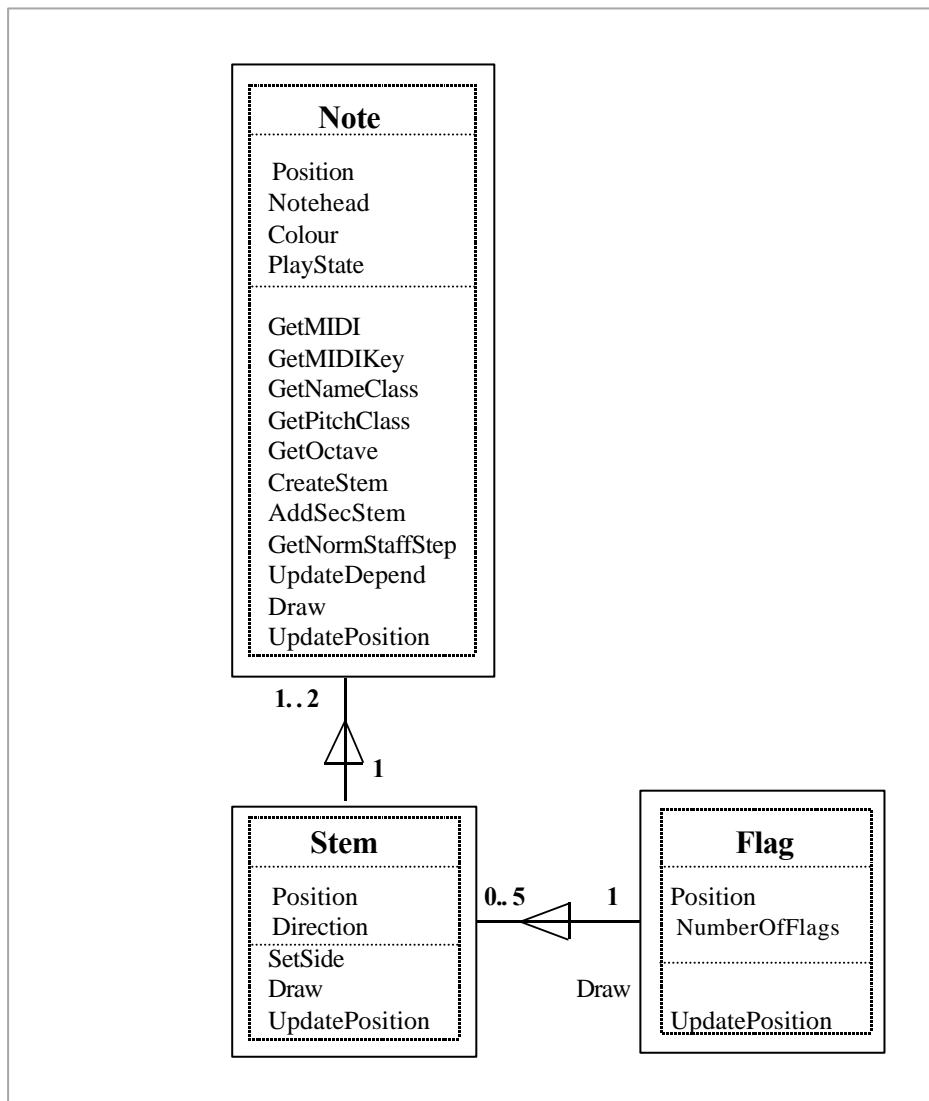


Figure 4.12 Whole-part structure of a Note object.

Stem objects use the flag-end endpoint co-ordinates as a primary reference point. To ensure maximum flexibility, the notehead-end endpoint is not related to the position of the notehead. A Note may have a secondary stem, allowing ascending and descending stems to be attached to a single notehead. Flags are dependent on the existence of a stem, a note cannot consist of a notehead, no stem and flags. A single flag object encapsulates all of the flags associated with a particular duration by storing the number of flags. Thus a semi-quaver note has a single flag object that records the number of flags as two. It is unnecessary to associate multiple flag objects with notes having durations shorter than a quaver, as class Flag does not require any specialised functionality.

In fact, a Flag object could easily be made redundant and implemented as an attribute of class Note.

Allowing Flag objects to be moved independently of a Note or Stem object, provides maximum flexibility, but requires that Flags be implemented as an object rather than an attribute. Note objects also have instance connections to Tie and Slur objects that function as semantic modifiers discussed in section 2.33. A Slur or Tie object connects two Notes as shown in figure 4.13 :

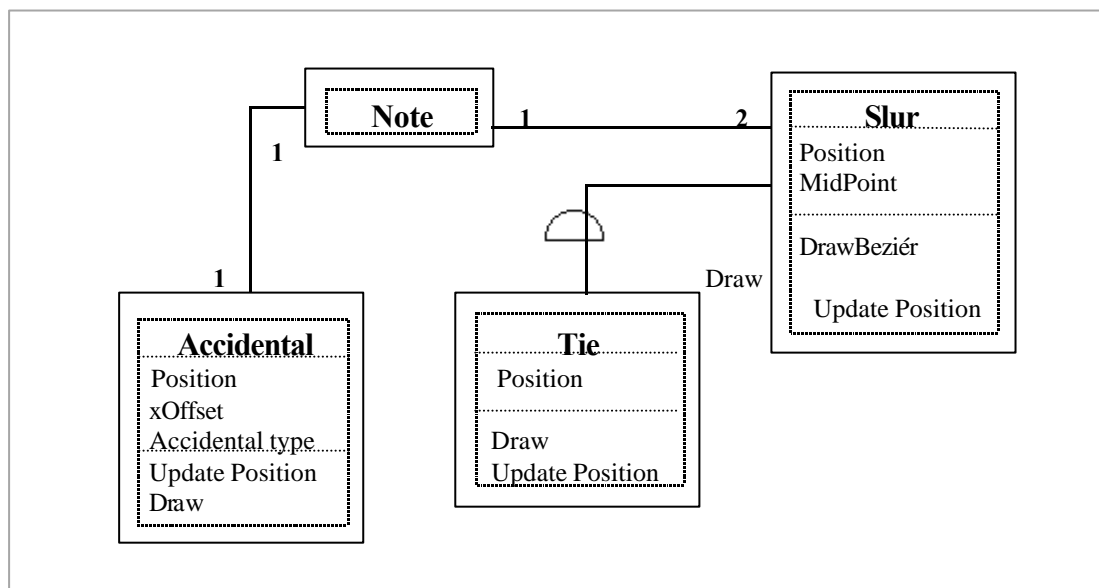


Figure 4.13 Relationship between a Note and Accidental, Tie and Slur objects.

#### 4.5.2 Rests

While being simpler objects than notes, Rest objects have an interesting property in that they do not have a specific pitch, but have a staff step attribute as shown in figure 4.14 :

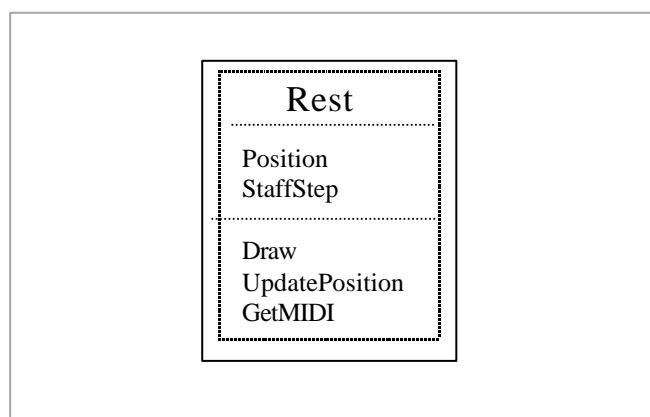


Figure 4.14 A Rest Object.

A Rest's staff step is used to position a rest on the y-axis. Rests have default positions in the centre of the staff (shown in example 2.9) that are only used when a single voice is notated on a staff. Multiple voices require that rests be positioned so as to achieve a satisfactory visual separation within the tessitura of the voice to which the rest belongs. Placement of rests on the y-axis is achieved by positioning the rest at a staff step. Especially important are semi-breve and minim rests, that are distinguished by their position directly above or below a staff line. Example 4.12 shows a canon with a minim rest in the upper voice notated above the staff, clearly indicating that it belongs to the upper voice :



Example 4.12 Rest notation in a two -voice canon.

### 4.5.3 Chords and MusicEvent Objects

All objects that have a sonic realisation are derived from class MusicEvent. Notes correspond to musical tones, while rests represent silence. Notes are implemented by being directly translated into MidiMessage objects, while Rests serve only to separate Notes. MIDI implementation of MusicEvents is discussed in detail in section 5.5. Chord objects can be directly derived from class MusicEvent (i.e. a chord is a symbol that can be directly translated into a sonic event) or, a Chord can be viewed as an aggregation of notes. In the latter case, the individual Notes form the sonic representation of the chord while the Chord object itself has no base class. These two representations are shown in figure 4.15 , and figure 4.16. Both representations are consistent with music theory, viewing a chord as a collection of notes.

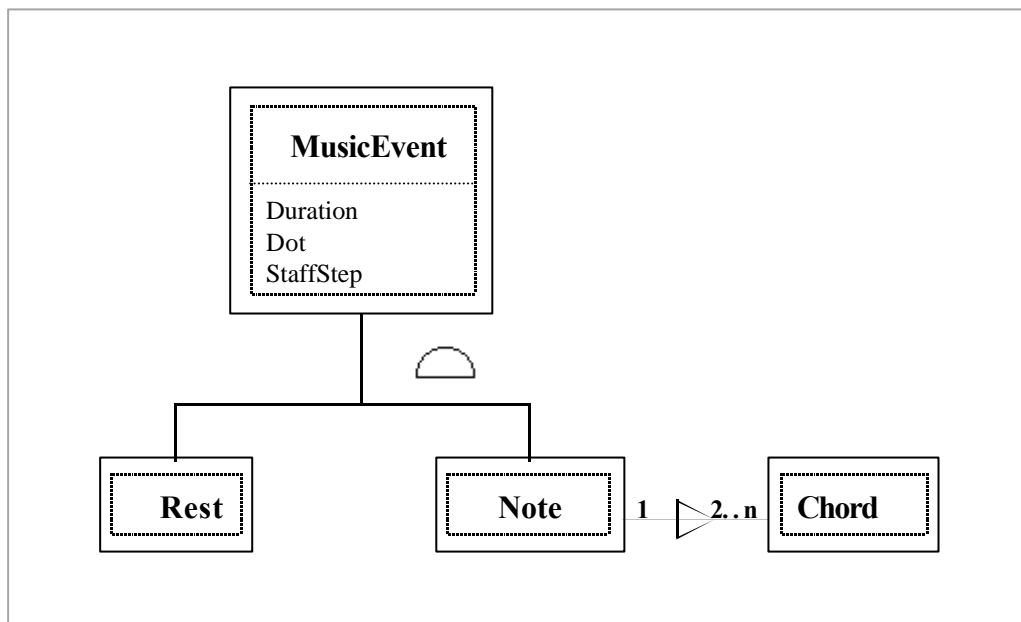


Figure 4.15 Class Chord as an independent class.

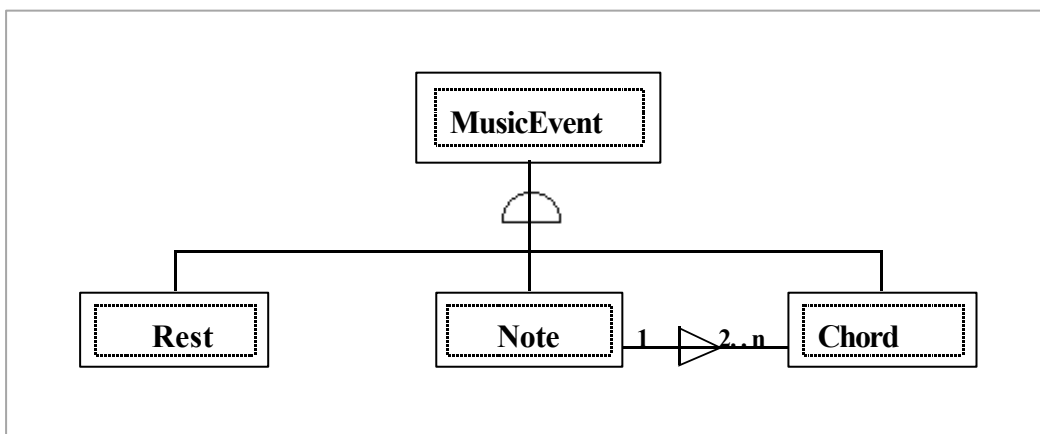


Figure 4.16 Chord class derived from Class MusicEvent.

It is generally preferable to favour higher-levels of conceptual abstraction and to give preference to whole-part relationships where the whole has direct control over its constituent parts. Figure 4.16 provides the highest level of abstraction (a Chord is a MusicEvent in its own right, even though it consists of Note objects which are also MusicEvent objects), while also allowing direct control over its constituent parts. An alternative arrangement that clearly separates individual notes from the notes that form a chord is shown in figure 4.17 :

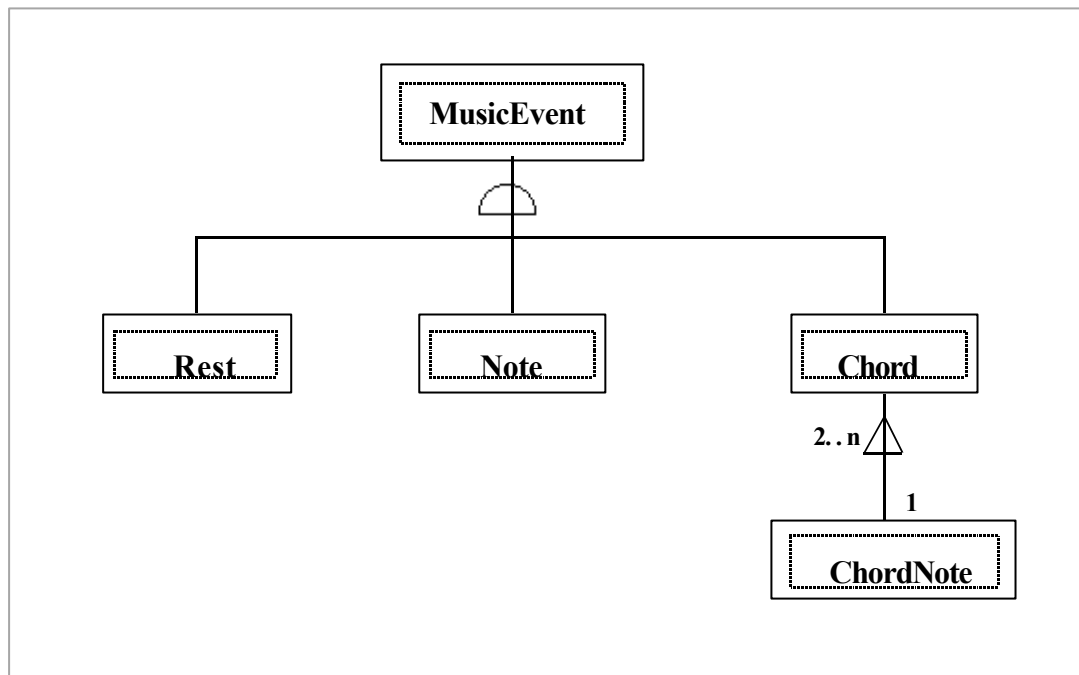


Figure 4.17 Distinct classes for notes and chord notes.

This representation has merit in its clear functional separation, but is unnecessary, as all the attribute requirements for a ChordNote that is a part of a chord are contained within a Note object.

Chord representation is also influenced by the choice of note representation. A chord can be viewed as a collection of noteheads and a single stem, or a collection of notes where one note has a stem. As mentioned in section 4.4.1 the OOTMN does not make use of notehead objects, and thus represents chords as a single note containing a stem (i.e. the stem note of the chord) and one or more notes without stems. The stem extends from the highest note downwards, or from the lowest note upwards. Notes can be translated into sonic events, while noteheads are purely graphic symbols that cannot be mapped onto a sonic realisation.

## 4.6 Beams and Beamed Groups

Semantic connections between flags and beams and especially their relationship to notes, provided an insight that became an important aspect of the design process. Computer implementations of music notation differ from the printed score in that the visual representation made up of symbols on the display screen has an equivalent underlying machine representation. Printed scores have no other representation scheme, the graphic symbols are the complete representation. The beams of a

beamed group thus function to indicate both the durations of notes and their grouping in a printed score.

In many computer representation schemes (including the OOTMN) beams do not determine note durations as all Note durations have already been specified before the creation of a beamed group. i.e. Notes are placed onto the staff and then beamed together. Duration is an attribute of a Note object and is not influenced by whether the note is attached to a beam or has flags. Beams are only graphic symbols that visually group notes. Beams function as the ‘notational conveniences’ discussed in section 2.3.4. Because beams do not influence the notes that they group, they can be graphically manipulated without influencing the semantics of the score. BeamedGroup objects create a relationship between a set of beams and a set of notes as shown in figure 4.18. A BeamedGroup has no attributes but consists of a large number of methods that are required to manage the complex relationships that exist between the notes and the beams. FindBeamPoints and SetStems calculate Beam positions and shrink or stretch stems when creating a BeamedGroup.

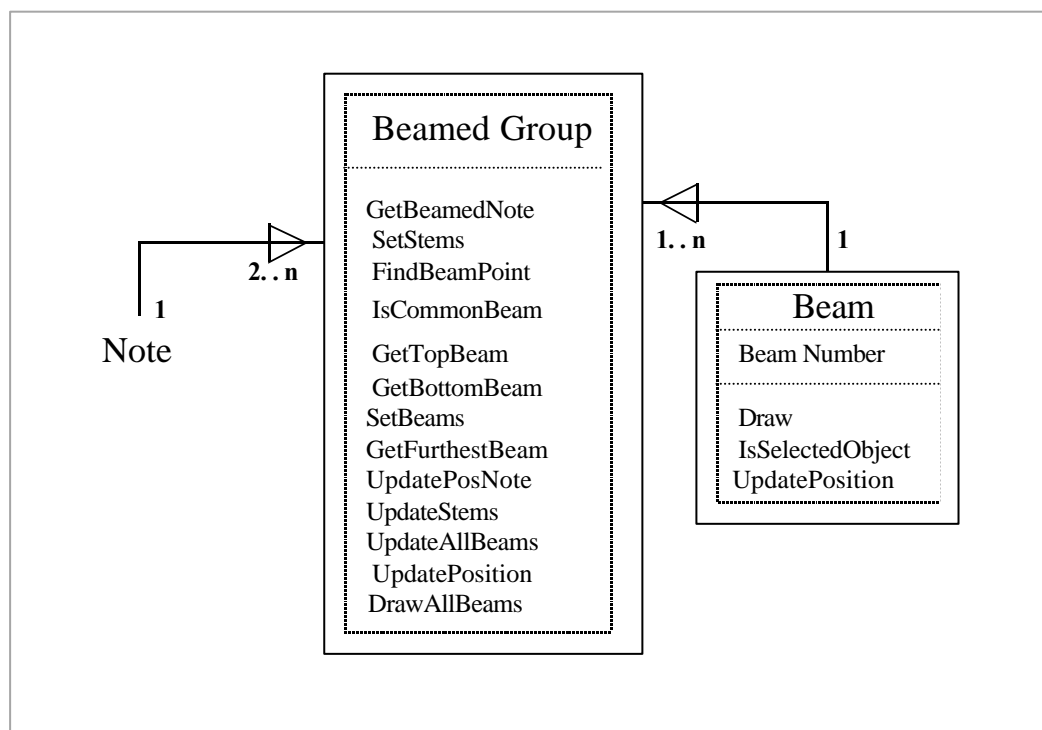


Figure 4.18 Structure of BeamedGroup and Beam objects.

CommonBeam determines which notes of a BeamedGroup have the same duration and thus share a Beam. UpdatePosNote updates a BeamedGroup's state after a Note has been moved. SetBeams and UpdateStems adjust Beams or Stems after a Beam is moved. UpdatePosition alters the positions of all Beams in a BeamedGroup. The remaining methods are self-explanatory.

Although beams are usually used to group notes according to metric divisions indicated by the time-signature, they should not be restricted to metric divisions i.e. consist only of notes that occur on the same beat. Composers use beams to indicate logical divisions of musical material, or to emphasise articulations. The example below from Brahms's first clarinet sonata illustrates the use of beams to emphasise articulation. The lower staff is a reconstruction of the original material of the top staff, but adheres to conventional practice.

The image shows two staves of musical notation. The top staff features a sequence of notes with beams that group notes across beat boundaries, emphasizing articulation. The bottom staff shows the same sequence of notes but with conventional metric beaming, where beams are aligned with the beat structure. Both staves include dynamic markings 'dim.' and 'p'.

Example 4.13 Brahms, *Clarinet Sonata no.1*  
First movement, measures 145 - 149.

Notes are articulated in groups of two, consisting of the quaver on the second-half of a beat and the quaver on the first-half of the next beat. Note how the emphasis on the articulation is lost in the version notated on the lower staff which uses a conventional metric beaming. Beams forming a beamed group are constructed by an algorithm that creates the beams required by the set of notes forming a beamed group. This original algorithm is presented in chapter five.



## 4.7 Irregular Groups

Irregular groups are groups of notes, rests, and chords whose durations are altered to represent durations that cannot be accommodated by the limited number of durational symbols used in music notation. Symbols representing duration have binary relationships as shown in figure 4.19 :

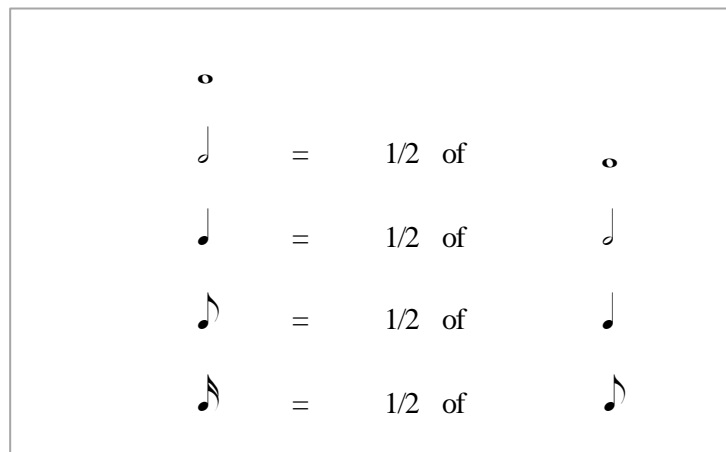






Figure 4.19 Relationships between note durations.

Durations that do not have a binary relationship to one of the available symbols (i.e. they have a duration that is as a third, fifth, or some other non-binary related value) must be notated as an irregular group. Grouplet objects encapsulate all irregular groups i.e. duplets, triplets, etc. Accurate representation is achieved by using four durations that correspond to the NIFF specification. These four integral values are the face number, face duration, actual number, and actual duration. The face number and duration represent the number of notated (i.e. written) notes and their durations. Actual number and duration represent the actual number of notes and their durations.

Three  durations performed in the time of two  durations is the most commonly encountered triplet. Using the representation scheme of four values :

Face Number = 3, Face duration = 

Actual number = 2, Actual duration = 

Thus three notated quavers are to be played in the actual total time occupied by two quavers. Each notated quaver has a performed duration of one-third of a crotchet. Class Grouplet is shown in figure 4.20 :

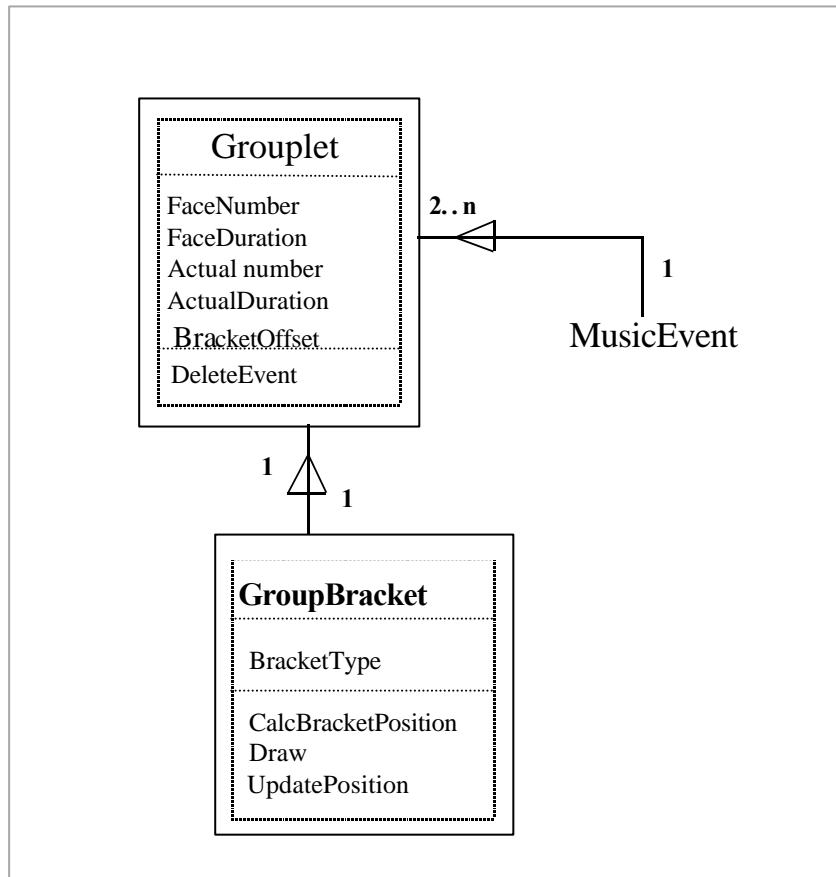


Figure 4.20 Structure of class Grouplet.

## 4.8 MidiMessage Objects

Notes have instance relationships to MidiMessage objects. MidiMessage objects form an interface between the OOTMN and the Maximum MIDI Toolkit (MMT) by encapsulating the MidiEvent structure of the MMT. The MMT is described in detail in the following chapter.

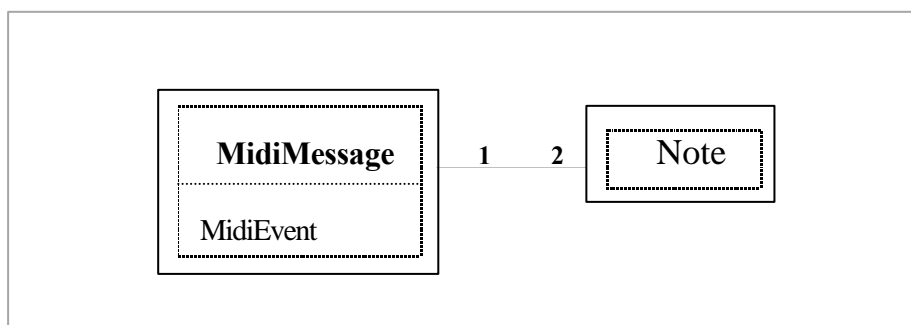


Figure 4.21 A MidiMessage object.

Only Note objects are translated into MidiMessage objects, a chord being represented in MIDI terms by its constituent notes. Rests cannot be translated into MidiMessage objects as MIDI does not have messages that represent silence. Rests only update the time before the start of the next note event following the rest. This process is discussed in chapter five.

## 4.9 Part Objects

A part, introduced in section 2.2.11 is usually a portion of a score that is to be performed on a single instrument. A Part object contains a complete Score object that represents the required notation extracted from the main score. Figure 4.22 shows a Part object.

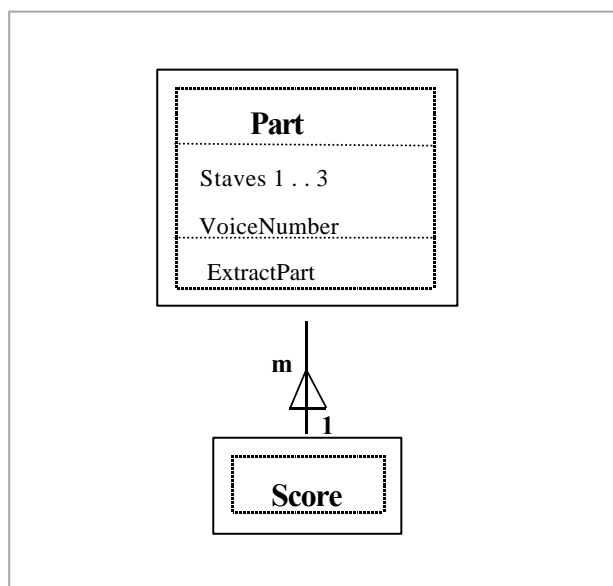
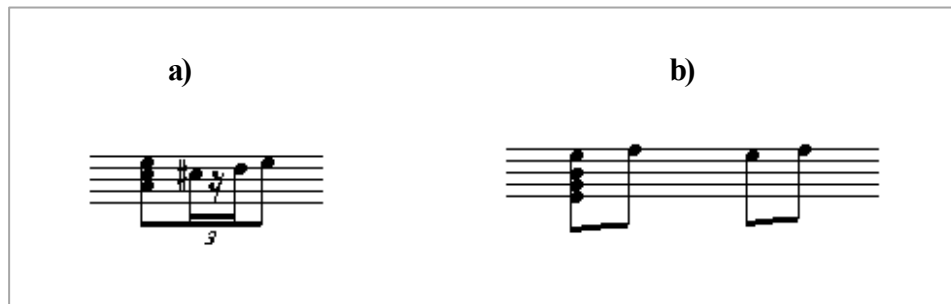


Figure 4.22 A Part Object.

A Part can extract notation from any voice using one to three staves. Method ExtractPart inserts a new Score object representing the part into the Part object. The OOTMN presently only extracts single staves.

## 4.10 Defining Object Relationships

Defining relationships between objects from the viewpoint of the musician often leads to relationships that, while logical, are complex and difficult to implement. A musician would regard example 4.14 a) as a grouplet embedded within a beamed group :



Example 4.14 Defining notational relationships.

Example 4.14 b) illustrates that graphic beam relationships between a single note and a chord are implemented as a relationship between two notes. As discussed in section 4.5.3 only one Note of the Chord object contains a Stem.

Musicians rely on groupings and sub-groupings to provide information for the correct interpretation of relationships. Computer representations directly capture relationships by logically organising related data, relationships are not dependent on hierarchical orderings derived from human visual groupings. For example, the OOTMN class BeamedGroup organises the required data by relating Beam and MusicEvent objects. Initially, object analysis regarded a beamed group as consisting of notes, rests, chords and grouplets following the visual, hierarchical interpretation outlined above. The object model of figure 4.23 captures these relationships :

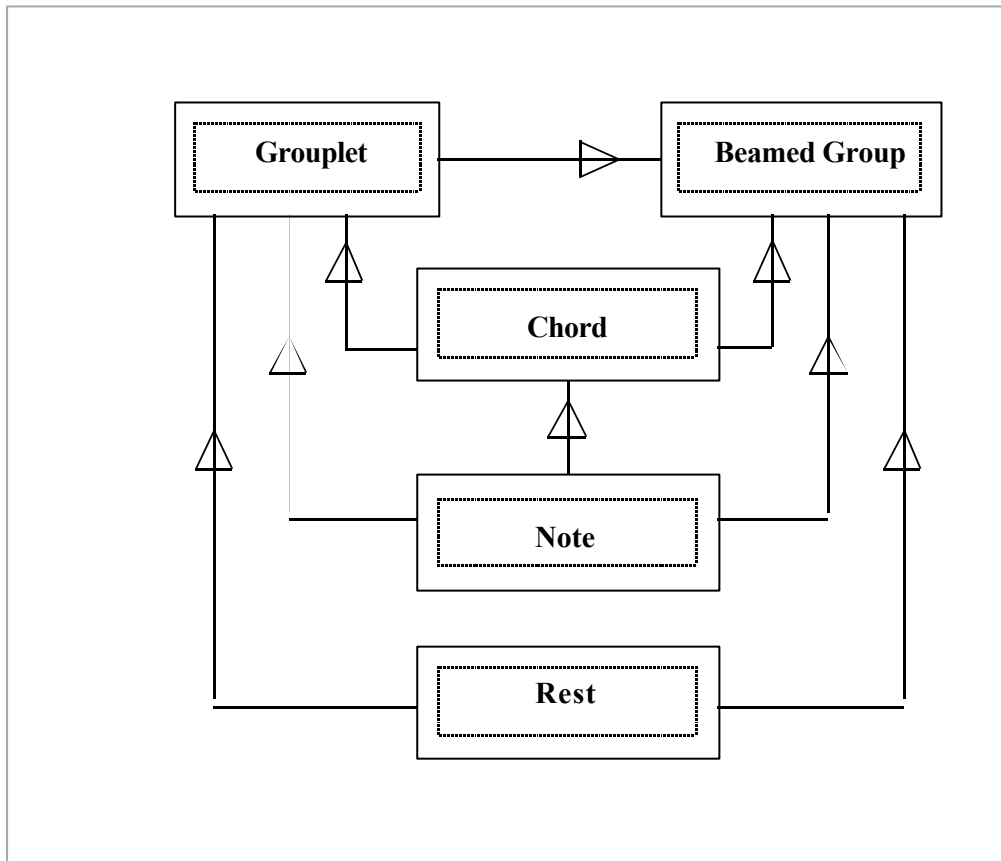


Figure 4.23 Representing notational relationships as classes.

The recursive nature of the relationships (both Grouplets and BeamedGroups contain the same objects, and BeamedGroups contain Grouplets) makes the model more complex than necessary.

A Grouplet is not required to be an explicit part of a beamed group. The Grouplet and BeamedGroup objects provide a visual relationship that implies that the grouplet is part of a BeamedGroup due to their common constituent parts. i.e. the required visual relationship is a side-effect of the computer representation. Figure 4.24 illustrates this relationship by providing an object model that corresponds to the musical fragment of example 4.14 a) :

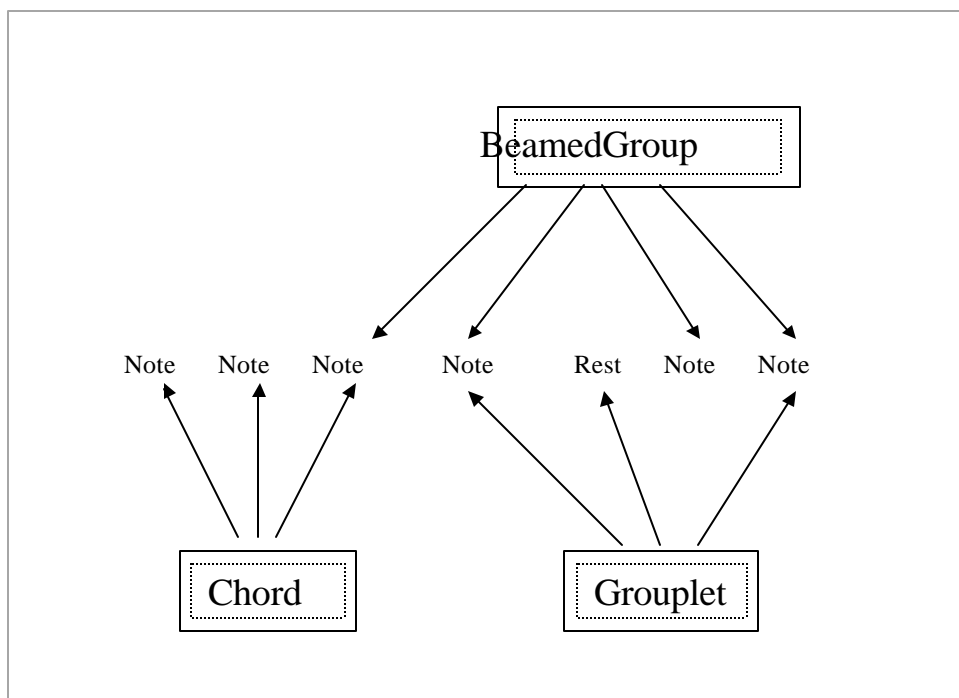
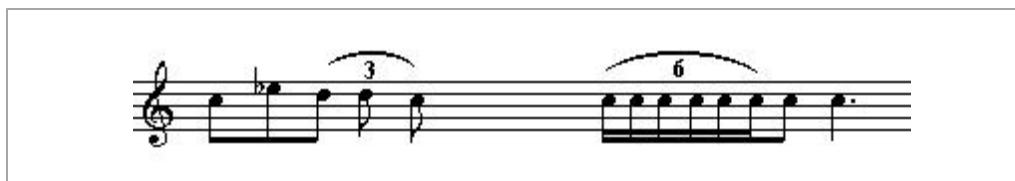


Figure 4.24 Object Relationships between classes BeamedGroup, Grouplet and Chord.

The design philosophy of the OOTMN that attempts to define symbolic relationships using as few constraints as possible allows the existence of relationships that are syntactically correct, but are idiomatically unsatisfactory due to their poor visual grouping. Example 4.15 shows two fragments that each have a grouplet and a beamed group that overlap each other :



Example 4.15 Unidiomatic notational relationships.

As beamed groups usually imply a metric grouping, beamed notes that form a unit greater (as in example 4.15) or less than the prevailing metric unit are awkward to comprehend.

Figure 4.25 shows an object model that is derived from the objects and their relationships discussed in this chapter :

Object Model p1

Figure 4.25 Complete OOTMN Object Model.





Figure 4.26 Message passing in the OOTMN.

Figure 4.26 shows the flow of messages between objects. Individual messages are not shown, the model summarises the relationship between two objects where one sends messages to another.

#### **4.11 Comparing MUSICA and the OOTMN**

The MUSICA notation program which was introduced in chapter two included documentation and the source code of the software. This documentation allows an understanding of the high-level

Figure 4.26 shows the flow of messages between objects. Individual messages are not shown, the model summarises the relationship between two objects where one sends messages to another.

#### 4.11 Comparing MUSICA and the OOTMN

The MUSICA notation program which was introduced in chapter two included documentation and the source code of the software. This documentation allows an understanding of the high-level structure of the software which is not easily obtained by studying the source code on its own. An object model of the MUSICA project, constructed from the available information, is shown in figure 4.27 and figure 4.28 :

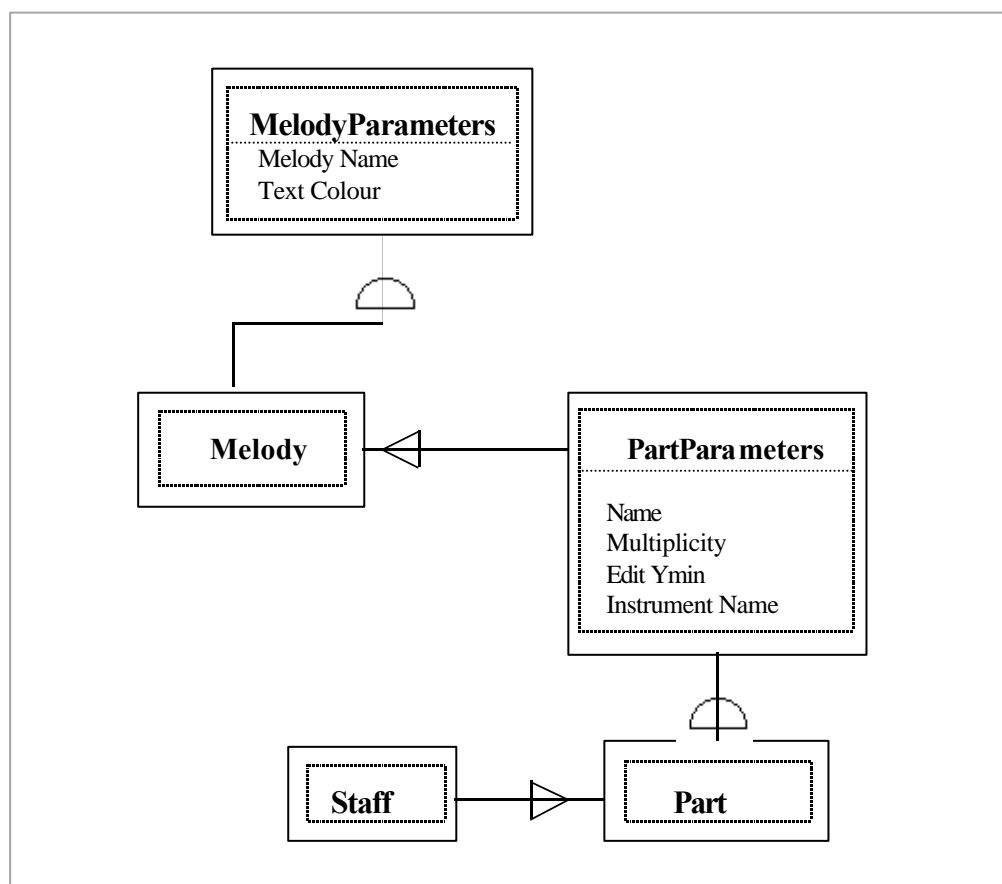


Figure 4.27 The class hierarchy of MUSICA.

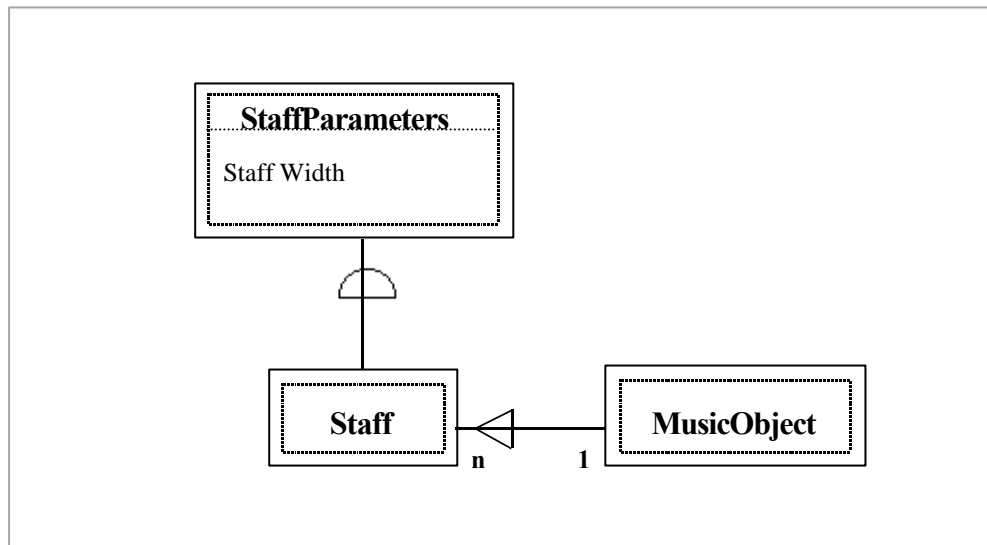


Figure 4.28 MUSICA Staff and MusicObject objects.

#### 4.11.1 Abstract Classes in MUSICA

Most of the objects used in the project are derived from a template (having the suffix ‘parameter’ appended to the class name) that is implemented as an abstract class. Abstract classes define templates by allowing derived classes to inherit attributes common to the category of classes defined by the abstract class. Abstract classes may not be instantiated, i.e. they cannot be created as objects in their own right as they function purely as templates. In figure 4.27 class Melody is derived from the abstract class MelodyParameters. Part and Staff classes are also derived from their respective abstract base classes. Figure 4.28 illustrates a staff object and its abstract base class.

Abstract classes thus provide a higher level of abstraction by factoring attributes common to all possible derived classes. They also support polymorphism which allows base class objects to act in accordance with the derived classes that they represent. Polymorphism is treated in more detail in the next chapter where the implementation of the OOTMN is discussed.

#### 4.11.2 The MUSICA Class Hierarchy

The highest-level class is the Melody class which represents the musical events occurring on a specific staff. Class Part defines a logical grouping of staves that defines the requirements of a particular instrument. Attribute multiplicity defines the number of staves required to notate the

instrument. For example, a piano would usually have a multiplicity of two, an organ would have a multiplicity of three.

Striking similarities in design to the OOTMN are apparent at certain levels of the object hierarchy. Class MusicSymbol corresponds to the OOTMN class of the same name, class ContinuousObject is similar to the OOTMN class LineSymbol. These classes shown in the diagram below are discussed in detail in chapter five.

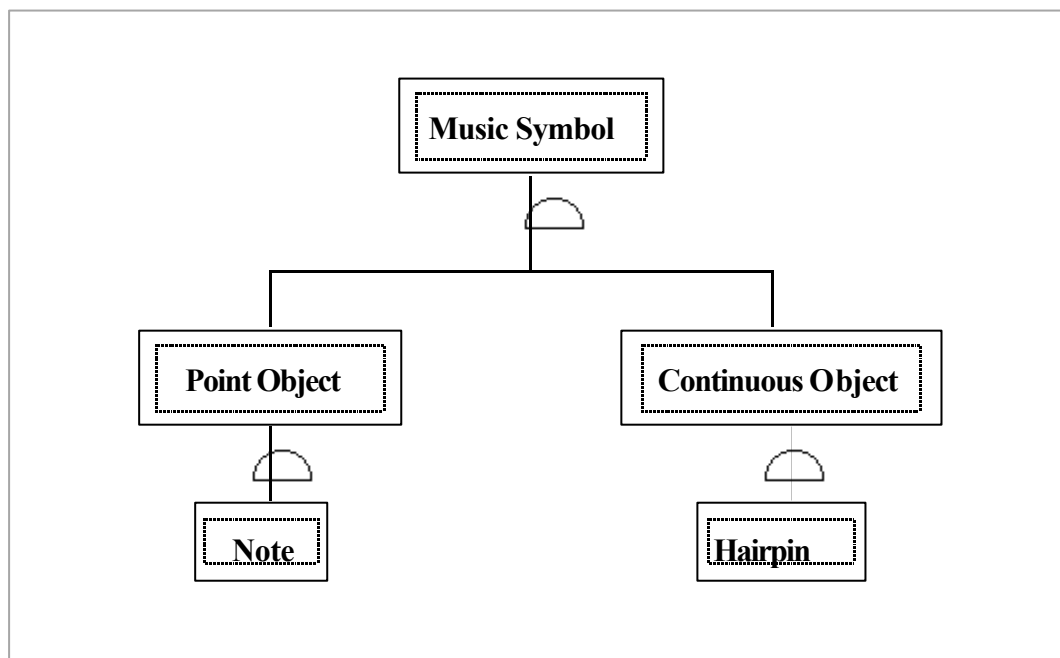


Figure 4.29 MUSICA Class MusicObject and its derived classes.

Considering the abstract classes used in the MUSICA project, the question arises as to whether many of these classes are necessary. For example, do all staves differ sufficiently to warrant representation as an abstract class? The answer is no, as different staves should not have different attributes or functionality. A staff does not represent a class of objects. Any characteristic differences between different types of staves can be adequately captured by attribute values within the class itself. Abstract classes are best used when objects share a few common attributes. Class PointObject is not strictly necessary if one of the points of class ContinuousObject is regarded as a reference point that can be an attribute of class MusicSymbol (i.e. all symbols, whether continuous or not have a primary reference point). This approach is followed in the OOTMN. The overall

impression gained from studying the MUSICA code is that the design process was mainly driven by object-oriented methodology, at the expense of the subject-matter being modelled. The design may have centred around the software technology due to a lack of insight into the subject-matter, or may have been the result of the designer pre-conceiving the design in terms of the capabilities offered by an object-oriented implementation. By contrast, the OOTMN was designed by first deriving an object model of music notation, and then considering the implications of representing the model in a programming language. The design process described in this chapter provides an object model that is elegant and creates a clarity of purpose that is necessary to successfully implement the OOTMN in a programming language.

## Chapter 5

### Implementing the Toolkit for Music Notation

#### 5.1 Selecting a Development Environment

Borland C++ version 4 and Borland's Object Windows Library (OWL) were used to implement the OOTMN. In retrospect, this may have not been the best choice of development environment, but was judged to be a suitable environment at the time that the decision was made. A variety of factors, including hardware platforms, MIDI development and compiler speed influenced the choice of development environment. The project was developed on multiple machines, including Intel 486-66 and Pentium MMX architectures. Borland C++ ver. 5 was not used as it does not perform well on a 486 processor. Previous experience with the Microsoft Visual C++ compiler proved this product to be significantly slower than the Borland Compiler. The OOTMN is implemented as sixteen-bit Windows code, due to the availability of a sixteen-bit MIDI library. Although a thirty-two bit development environment would have been preferable, the overhead associated with implementing MIDI functionality ruled out a thirty-two bit implementation. Relationships between the different components of the development environment are shown in figure 5.1 :

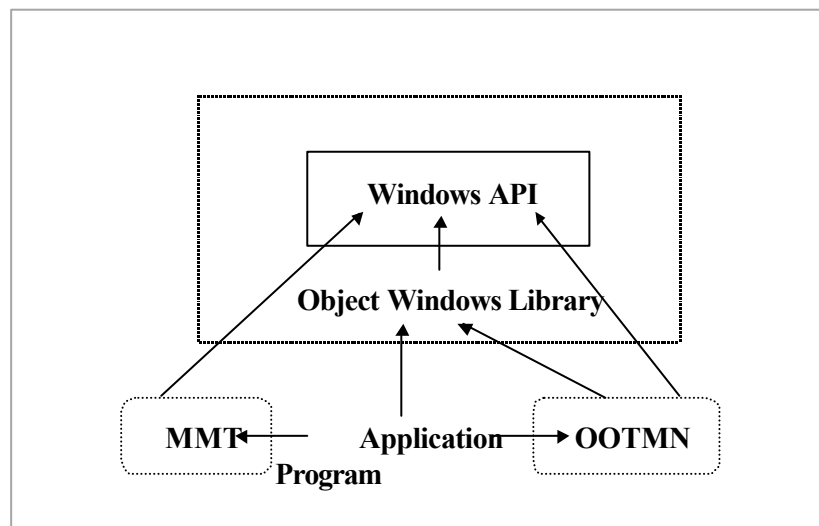


Figure 5.1 Relationships between the Windows API, the OWL, the MMT and the OOTMN.

Borland's OWL provides an object-oriented encapsulation of the Microsoft Windows Application Programmer's Interface (API). Windows development occurs as a mixture of application code in

the chosen development language and Windows API functions that form an interface between the application and the Windows environment. The OOTMN and MMT are implemented as libraries that are used by application programs. Use of the Borland OWL does not prevent an application from bypassing the Borland libraries and directly using API function calls. The OOTMN occasionally makes direct use of certain Windows API functions if these functions are easier to use than their OWL equivalents. Borland's container class library arrays are used to store sorted collections of objects such as MusicEvent and MidiMessage objects. These arrays automatically sort notes, rests and chords according to horizontal position, and MidiMessage objects according to timestamps.

## 5.2 Real-world models and machine models

The previous chapter examined the development of a model of music notation that only considered the development process in terms of the subject matter itself. Computer software and especially object-oriented software, attempts to model the real-world. It can never recreate or be the real world. Models derived from analysis of the subject matter often require further refinement to be elegantly implemented in a programming language.

Real-world objects (and object models) derive structural relationships from whole-part relationships (aggregation), and generalisation-specialisation relationships. Instance relationships are relationships or interactions between objects that are not of a structural nature. The model developed in the previous chapter is almost entirely based on whole-part relationships, containing very few generalisation-specialisation relationships. The gen-spec relationships that do occur, do not exist within the most important relationships such as those between systems, staves and barlines that occur at the centre of the object hierarchy. Whole-part relationships are implemented in a programming language by means of pointers that relate whole to part (and if necessary vice-versa) or, by nesting declarations of the parts within the whole. The structure and implementation of a whole-part relationship using pointers is no different from the structure and implementation of traditional data structures such as linked lists. Object-oriented technology provides no specific means of accessing and manipulating aggregate relationships. Maintenance of these relationships is left to the programmer. Gen-spec relationships provide powerful object-oriented functionality that is



directly supported by object-oriented programming languages such as the C++ language. A real-world model that is to be successfully implemented in a software environment should be adapted to conform to the characteristics and idiosyncrasies of that environment. Such adaptation does not imply that object relationships be altered in any way. Careful thought and further analysis of the problem domain can result in enhancements to the object model that simplify implementation, while retaining relationships that accurately model the subject matter.

### 5.3 An Object-Oriented Implementation of the Model

Considering the toolkit for music notation as a whole, it is desirable to identify operations that would be useful to applications built using the toolkit. Three such operations were identified, searching for objects, moving objects, and the playback of objects that can be translated into sound. Most of the objects identified in the object model directly correspond to music symbols. Hairpins, notes, rests, barlines and many other objects have direct graphic counterparts of the same name. Common characteristics of these objects can be factored out to produce an abstract base class MusicSymbol shown in figure 5.2 :

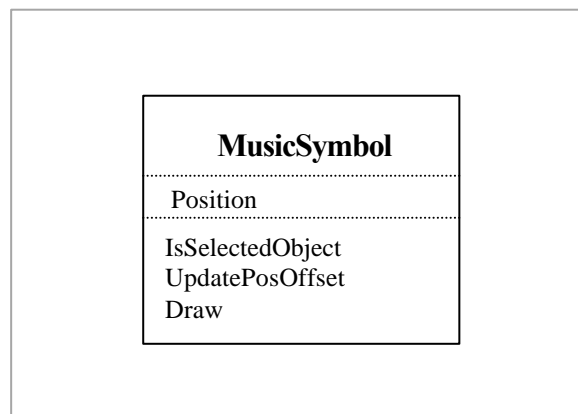


Figure 5.2 Structure of a MusicSymbol object.

Abstract classes cannot be directly implemented (i.e. instantiated), they serve as templates for the creation of derived classes. Class MusicSymbol implements searching for objects and the updating of object positions using the object-oriented runtime functionality provided by gen-spec relationships.

Classes that are derived from the abstract base class MusicSymbol are able to implement methods appropriate for their particular context. For example, updating the position of a Note requires that the position of its Stem object also be updated. Polymorphism (discussed in section 5.3.3) and the ability to override methods inherited from a base class provide this functionality. The Attribute Position and method Draw are self-explanatory. The other methods are explained in section 5.3.3.

### 5.3.1 Creating Objects

Barlines were initially modelled as forming part of a bar, and this relationship was implemented by means of pointers from a Bar object to its left and right-hand Barlines. When this approach was abandoned in favour of barlines not being part of any object, but reflecting multiple relationships to bars and staves, a problem was encountered in determining which object should be responsible for creating barlines.

By default, the OOTMN creates initial barlines that span an entire system. Creation of the object hierarchy proceeds in such a way that each object creates its constituent parts. A System object creates its first Staff, this Staff in turn creates its first Bar. The newly created Bar should be able to have pointers to the Barlines which determine its extent. These instance relationships should be defined by parameters that are passed to the Barline objects' constructor. Unfortunately, at the time of the first Bar's creation the entire System has not been created. The last Staff of a System determines the initial ending position of Barline objects. Partial construction of objects i.e. leaving object attributes undefined to be updated at a later stage by a different section of code is a poor object-oriented programming practice. When two objects are dependent on each other and the one cannot create the other, they should both be created within the function which implements their dependent relationship, or, by a higher-level object where feasible. A solution was devised where System objects create their immediate constituents (i.e. Staff objects), as well as the parts of these constituents (i.e. Bars), as well as all Barlines. It must be emphasised that Barline objects do not have any continuous relationship to Staff objects. Barlines require the position of Staff objects only when being constructed, after which they have no contextual dependencies. The relevant portion of the object hierarchy is shown in figure 5.3 :

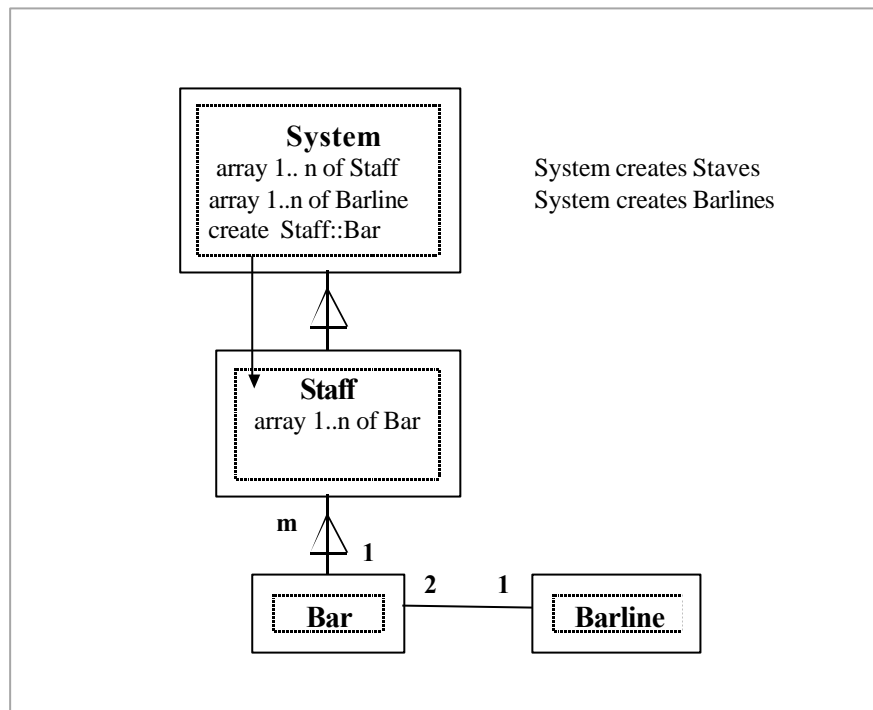


Figure 5.3 Hierarchical relationships that determine object creation and storage.

After all objects have been created, a **System** updates all **Staves**, **Bars** and **Barlines** to capture the relationships that exist between them. **Bar** objects still form part of a **Staff**, but because of the complex relationships involving **Barlines**, a **Staff** does not create its own **Bars**.

### 5.3.2 Storing Objects

Relationships usually determine where an object is stored in a program. Aggregation implies that the parts of the whole are stored in the whole, either directly using nested objects, or in the case of the OOTMN, indirectly using pointers to objects.

**Barlines** determine the extent of one or more bars, starting (initially) from the top of one staff and ending at the bottom of another. They thus have a relationship to **Bars**, determining the starting and ending positions of **Bars** on the *x*-axis. The left-hand and right-hand barlines of adjacent bars share the same horizontal position. Operations such as breaking a barline between staves requires that a single **Barline** object be replaced by two **Barline** objects. Moving **Barlines** requires that overlapping **Barlines** be synchronised. To facilitate these operations it is desirable that all **Barlines** be stored in a

single data structure where they are easily accessible. If Barlines are not stored together and are only accessible by means of the pointers determining object relationships (i.e. via Bar objects), operations on Barlines become difficult to implement. Barlines are stored within the highest level object where a significant relationship exists. As outlined above, Barlines have significant relationships to Bars but are created by System objects. As a barline exists within a single system (a barline denotes a temporal division and thus cannot traverse multiple systems), Barlines are stored at the System level as illustrated by figure 5.3.

### 5.3.3 Searching for Objects

Each class derived from class MusicSymbol has a non-virtual method FindSymbol that directs the search towards a specific object's constituent parts. The method is non-virtual as each object is responsible for searching its constituent parts. When method FindSymbol finds an object, the object is cast to its base class (MusicSymbol) and returned all the way up the object hierarchy via the successive FindSymbol() calls that found it. This process is illustrated using Coad notation in figure 5.4 and psuedocode in figure 5.5

Two other non-virtual methods that return boolean values, IsSelectedObject and ContainsObject assist the search. IsSelectedObject determines whether an object is selected or not by performing hit-testing on each MusicSymbol object examined during the search process. ContainsObject directs the search to the appropriate System, Staff, Bar or MusicEvent where the object is to be found.

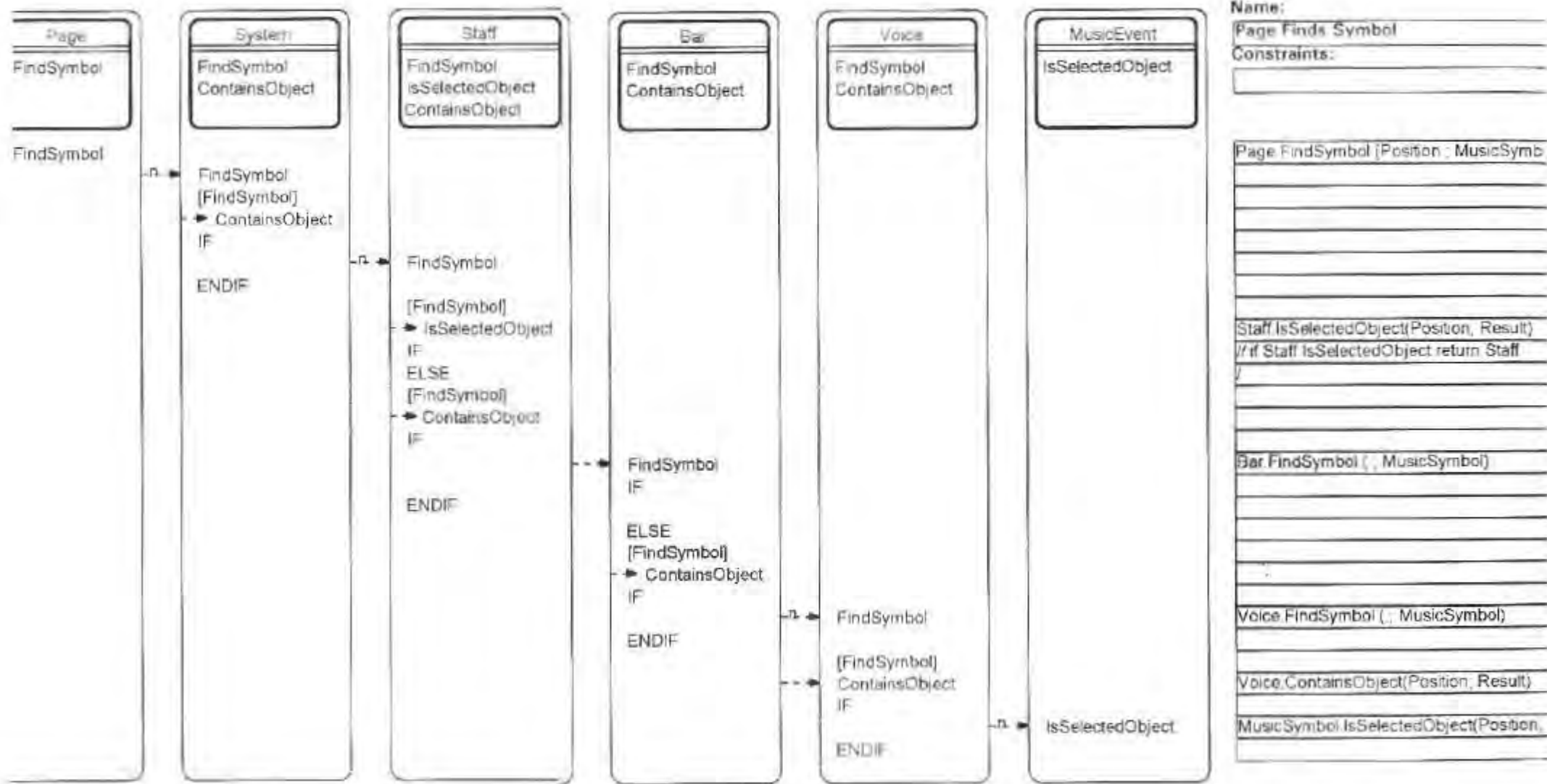


Figure 5.4 Searching process modelled by Coad notation.

```

MusicSymbol* ms = ptrPage->FindSymbol(point)

Page::FindSymbol ( point)
  for each System
    for each Staff {
      if ( SelectedObject (Staff))
        return (MusicSymbol) Staff;
      else if (Staff.ContainsObject ())
        return Staff.FindObject (point);
      else return NULL;
    } //for

Staff::FindSymbol (point)
  for each Bar {
    if Bar.ContainsObject(point)
      for each Voice
        if Voice.ContainsObject(point)
          for each MusicEvent
            if ( SelectedObject (MusicEvent))
              return (MusicSymbol) MusicEvent;
            else if (ContainsObject (MusicEvent))
              return MusicEvent.FindObject (point);
            else return NULL;
  }

```

Figure 5.5 Searching process modelled in psuedocode.

Staves have a bounding rectangle shown in example 5.1 that determines how far notes and rests may exist above or below the staff. As a staff is conceptually unbounded, the bounding rectangle removes conflicts of association occurring between MusicEvent objects and adjacent staves.



Example 5.1 Staves and their bounding rectangles.

Bars also contain bounding rectangles determined by the x-axis of the staff bounding rectangle and their left and right-hand barlines. These rectangles are used by the ContainsObject method to constrain the search process within a specific Staff or Bar.

### 5.3.4 Moving Objects

Object-oriented systems derive their expressive power from polymorphism, the ability to bind an appropriate derived class method to a base-class pointer at run-time. A pointer to any class derived from a common base class can be stored as a base class pointer. At run-time the method from the appropriate class will be bound to the pointer depending on the actual class of object pointed to. Such methods are known as virtual methods. Consider the following simple pseudocode example that implements a search using polymorphism and virtual methods in a mouse-driven environment :

```

class Base {
    virtual void Print(""); //virtual method
};
class D1 : public Base {
    void Print { >>"Apples class D1; } //virtual method
};
class D2 : public Base {
    void Print() { >>"Pears class D2"; } //virtual method
};
main() {
    Base* ptrBase = NULL;
    if (Click) ptrBase = FindObject(position);
    if (NULL != ptrBase) ptrBase->Print();
}

FindObject (position) {
    for all Pear objects if (found at position) return (ptrBase ) Pear;
    for all Apple objects if (found at position) return (ptrBase ) Apple;
}

```

Figure 5.6 Searching using Polymorphism.

The output depends on whether the user clicks on an object of type D1 or D2. The base class pointer `ptrBase` invokes the `Print` method of either D1 or D2 depending on the actual derived object pointed to by `ptrBase`. Method `FindObject` searches for both Pear and Apple objects.

An identical process may be used to update the positions of music notation objects that are moved, as well as their spatially dependent objects, given the existence of a meaningful base class. Objects that can be moved are all derived from a base class `MusicSymbol`. As mentioned in section 5.3 not all objects are of class `Music Symbol`. Only objects that are notation symbols which that can be selected and moved are music symbols. `Score`, `Page`, `Voice`, `System`, `Beamed Group`, `Irregular Group` and `Bar` are not derived from class `MusicSymbol`.

Class `MusicSymbol` has a data member `Position` which holds the x and y co-ordinates of each `MusicSymbol` object, and a virtual method `UpdatePosition`. This method updates the values stored in `MusicSymbol::Position`, and updates all symbols that are spatially dependant on the symbol being updated. `UpdatePosition` polymorphically invokes the appropriate method to update a symbol's position from a pointer to a `MusicSymbol` object returned from the search process. `Position` stores actual screen co-ordinates and is the primary reference point of the symbol. NIFF refers to such a point as an anchor. Dependent symbols are updated by virtual or non-virtual functions of the same name (`UpdatePosition`). This process is illustrated in figure 5.5 using a scenario view for the process involved in updating a system's position. A second example is given in figure 5.6 which illustrates updating a `Note`'s position in psuedocode. The function `Note::UpdatePosition` is implemented as a virtual function that updates the note's position. `Note::UpdatePosition` also contains non-virtual functions that update its dependencies by means of an offset calculated as the difference between the `Note`'s old and new positions. This offset is passed as a parameter to `UpdatePosition` and is not part of the OOTMN, being calculated by the user interface of the application program.



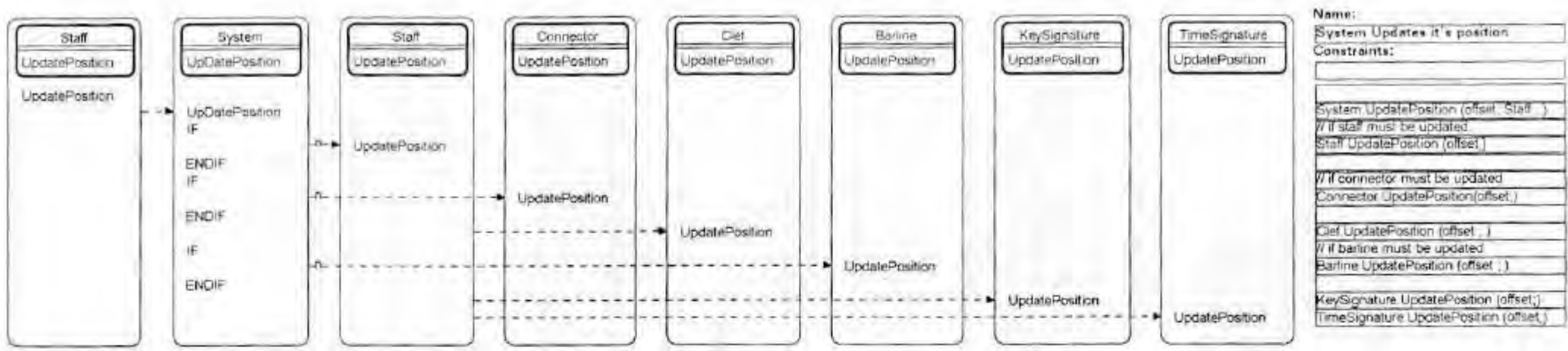


Figure 5.7 Updating a symbol and it's dependent symbols modelled using Coad notation.

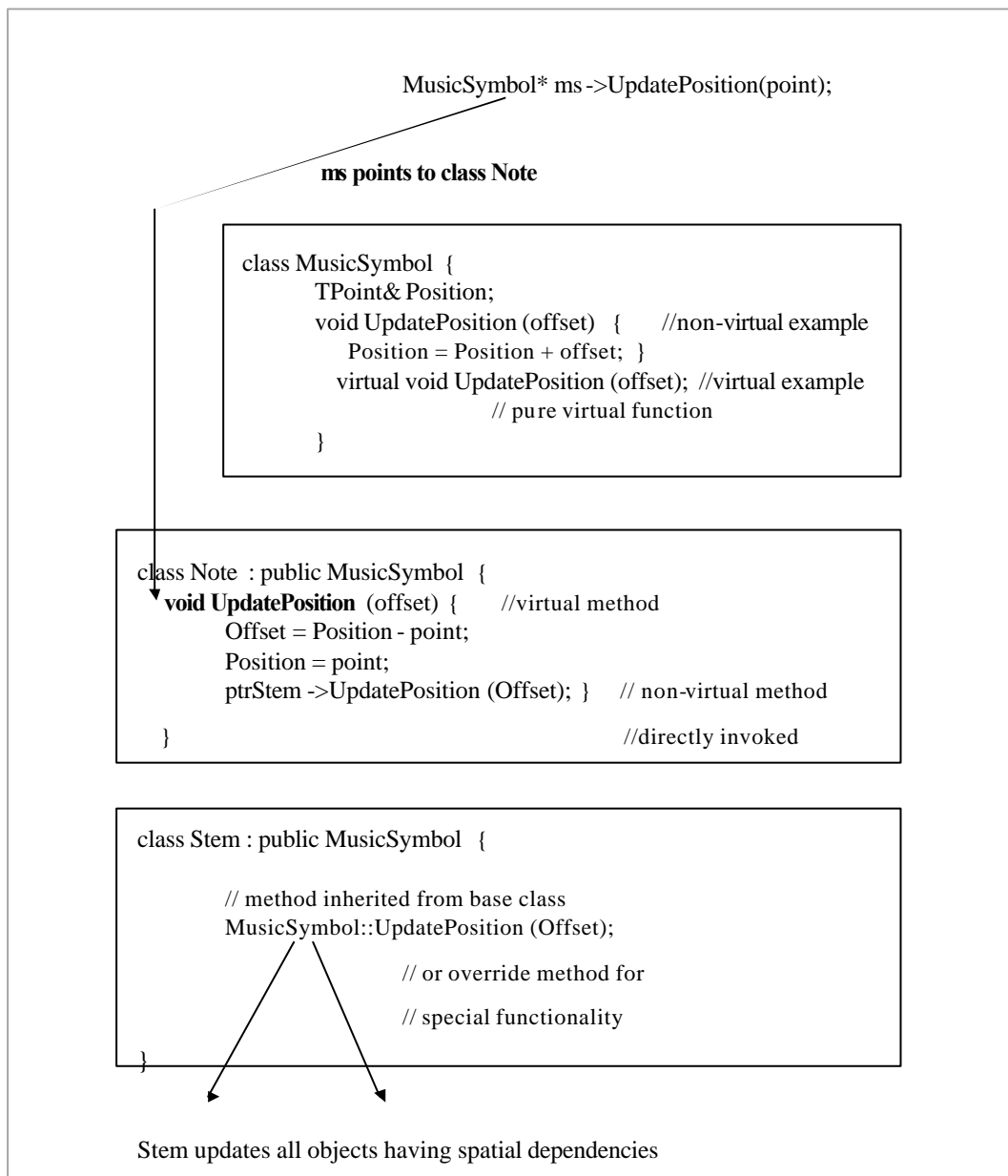


Figure 5.8 Updating a Note and it's dependencies positions in pseudocode.

Figure 5.8 assumes that the base class pointer `ms` points to a valid derived object. As the pointer `ms` points to a `Note` object, polymorphism ensures that the method `Note::UpdatePosition` is invoked. The `Note` object is then able to update all symbols that have a context-sensitive relationship to it. In figure 5.8 it updates its `Stem`, `Stem` in turn can update all objects spatially dependent on it such as `Flags` or `Beams`.

## 5.4 Implementing Objects

Significant differences exist between representing music symbols in a graphic context (i.e. for display on a computer terminal) and representing symbols in a manner that is useful within music theory. It is essential that objects be represented in terms of the subject matter. This is good programming practice and avoids additional application code that must convert representations to a suitable format.

### 5.4.1 Encoding Music Symbols

Computer storage requires numeric representations of symbols. Many schemes exist including the popular DARMS code described by Brinkman [Brinkman, 1990], and the NIFF encoding format. Integer encodings used in the OOTMN attempt to follow the NIFF format to facilitate the future import and export of NIFF files.

### 5.4.2 Representing Notes

Encoding musical pitch presents several implementation problems resulting from the octave repetition of the seven alphabetic pitch names and the enharmonic equivalents found in the chromatic system. Unambiguous representation of a note within the context of music theory requires that a note be able to determine its name class, pitch class and octave. Staff steps (and thus note names) do not provide a representation of all possible pitches, being limited to the seven diatonic pitches of C major. Unambiguous representation of musical pitch is described by Brinkman [Brinkman, 1990] and Dyer [Dyer, 1986].

#### 5.4.2.1 Name Class

The alphabetic characters A to G form a name class (i.e. a set of names) that map onto NIFF staff steps according to the clef used. Staff steps are constants with zero always referring to the lowest line of the staff. The musical name of a staff step is contextually dependent on the clef associated with a staff. A name class is a mapping of alphabetic note names onto the integers 0 . .6. The following diagram illustrates the relationship between note names, name classes and staff steps within the context of the treble clef :

Name class :	2	3	3	4	4	4	5	5	5	6	6	0	0	1	1	1	2	2	3	3
Musical name :	E	F	F#	Gb	G	G#	Ab	A	A#	Bb	B	C	C#	Db	D	D#	Eb	E	F	F#
Staff step :	0	1	1	2	2	2	3	3	3	4	4	5	5	6	6	6	7	7	8	8

Example 5.2 Name class and staff step relationships.

All notes having the same name map onto the same name class irrespective of whether they have accidentals or not. Thus the notes C, Cb, Cbb, C# and Cx all have a name class representation of zero.

#### 5.4.2.2 Pitch Class

A Pitch class represents all twelve chromatic pitches by indicating the distance of a pitch from the name class 0 (i.e. the note name 'C') in semitones. Figure 5.9 shows pitch class representations related to note letter names on the keyboard. Pitch class representation does not distinguish between enharmonic equivalents, both F# and Gb are PC 6 as they are both six semitones from PC zero.

C#	D#			F#	G#	A#		C#	D#			F#	G#	A#										
Db	Eb			Gb	Ab	Bb		Db	Eb			Gb	Ab	Bb										
C	D	E	F	G	A	B	C	D	E	F	G	A	B											
Pitch classes	0	1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6	7	8	9	10	11

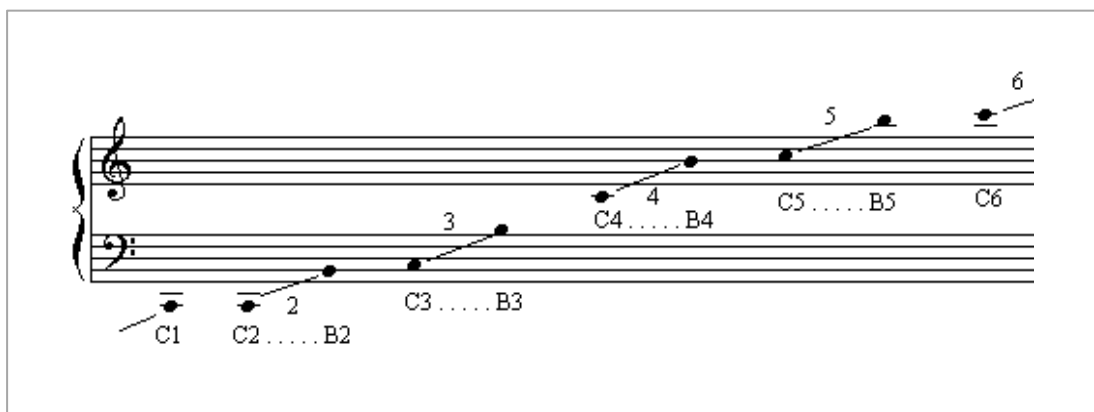
Figure 5.9 Pitch classes and note letter names related to the keyboard.

Representation of music notes requires careful consideration, as a simple numeric encoding of a musical note has little usefulness within the domain of music theory. The name class, pitch class and octave are required to provide an unambiguous representation of a particular note.

Class MusicSymbol stores the screen co-ordinates of a Note object. A Note has a staff step attribute and an instance connection to an Accidental object. The staff step allows computation of a name class, while a pitch class can be computed from the staff step and accidental value. Name and pitch class values are used in transposition (i.e. the calculation of intervals) within the OOTMN and can be used by application programs built using the OOTMN. The harmony tutor presented in chapter six makes use of name and pitch class representations to generate appropriate exercises, as well as to evaluate completed exercises.

### 5.4.2.3 Octave Representation

Octave placement can be indicated by the accepted standard notation that assigns a unique integer to each octave range between consecutive occurrences of the note names 'B' and 'C' :



Example 5.3 Octave representation.

Octave placement is not used internally by the OOTMN. Octave encoding is usually only required by purely numeric encoding schemes that do not make use of staff steps. The OOTMN note class does have a method that returns the octave number of a Note should an application require it. Name class, pitch class and octave representations can be packed into a single numeric representation creating what Brinkman [Brinkman, 1990] refers to as a Binomial Pitch Representation. This

representational scheme was considered for representing the pitch of Note objects, but was rejected in favour of a representation that emphasises the position of a Note by means of staff step and position attributes.

### 5.4.3 Line Symbols

As discussed in section 5.3, all available symbols have a reference position stored in the base class `MusicSymbol`. This position forms a reference point around which a hit spot is created for hit-testing. Certain Symbols such as hairpins contain more than one hit spot. Such symbols are usually created from a set of line segments which can be stored in an array as a set of points. Class `LineStyle` is the base class for all objects that consist of a set of points. `Hairpin`, `Beam`, `Stem` and `Bracket` objects are all derived from class `LineStyle`.

The hairpin symbol shown in figure 5.10 consists of two line segments between points one and two, and points two and three. It is desirable to be able to stretch the symbol horizontally, adjust the vertical position of the whole symbol and to open or close the hairpin. A hitspot at each of the three points defining the placement of the symbol accomplishes these tasks. Hitspot two adjusts the entire

symbol along the *y*-axis or stretches the symbol to the left. Hitspots one and three stretch to the right and open or close the hairpin around point two.

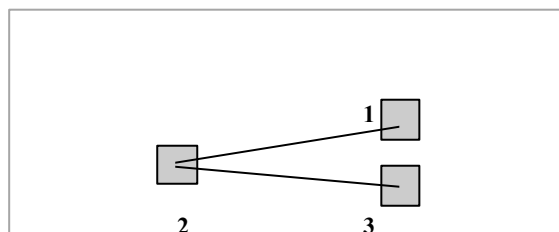


Figure 5.10 Hairpin object hitspots.

Line Symbols are derived from class `MusicSymbol`, retaining a primary reference point (given by `MusicSymbol::Position`), while having an array of points that specify secondary reference points used for hit-testing. Member function `IsSelectedObject`, which returns a boolean value for music symbols, is implemented to return an index identifying the hitspot of a line symbol. This arrangement exploits a convention of the C programming language which regards zero as representing boolean false and any positive integer as boolean true. A `LineStyle`'s array of points also simplifies drawing the symbol by storing data used by the `Draw` method which is implemented as a virtual method.

## 5.4.4 Special Symbols

Most music symbols are atomic entities obtained from the available symbols of the TrueType font used by the OOTMN. Certain symbols are more complex, requiring the use of an algorithm to create the symbol, or requiring symbols that cannot be created from graphics primitives supported by the Windows API. Symbols that cannot be created algorithmically and implemented using Windows graphic functions must be loaded as bitmap resources.

### 5.4.4.1 Slurs, Ties, and Phrase Markings

Slurs, ties and phrase markings are implemented as cubic Bézier splines (curves) which are commonly used in computer graphics applications. Bézier curves are described by Taylor [Taylor, 1994] and Glassner [Glassner, 1990]. The structure of a cubic Bézier curve is shown in figure 5.11 :

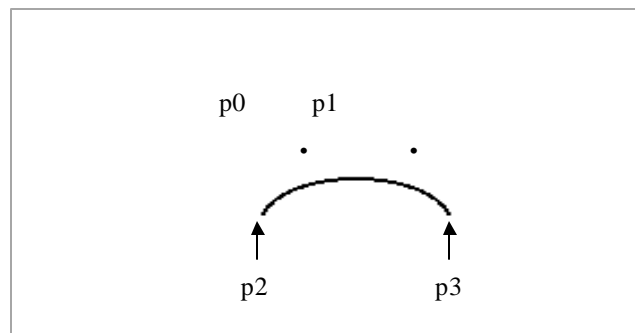


Figure 5.11 A cubic Bézier curve.

Cubic Bézier curves consist of four points. Two points determine the starting and ending positions of the curve, the other two points are control points that determine the shape of the curve. In figure 5.11 p2 and p3 are the starting and ending points, p0 and p1 form the control points.

The algorithm used to compute the set of Bézier points was obtained on the Internet from a C language program written by Dominic Giampaolo [Giampaolo, 1996], who has placed the code in the public domain. In order to create a curve that closely approximates an engraver's slur, a double curve is used by the OOTMN where the starting and ending points are the same. The control points are separated by a single screen pixel on the y-axis, creating a curve that is thicker towards its midpoint.

Slurs, Ties and phrase markings are all implemented by means of the same double Bézier curve. Slurs and Ties are semantically different, depending on the context within which the same visual symbol is used. Slurs indicate a group of notes that must be articulated together (each note as long as possible) and form instance connections to two adjacent notes of differing pitch, or two non-adjacent notes. Ties which add the durations of the two notes that they tie together must have instance connections to two adjacent notes of the same pitch. Phrase markings which indicate the start and end of a musical phrase must be implemented separately to distinguish them from slurs.

#### 5.4.4.2 Beams

A beam is implemented as a LineSymbol and constructed by using a Windows graphic function that draws a polygon from a set of points. Two left-hand and two right-hand points always share the same position on the x-axis, ensuring that the vertical sides of a beam are aligned to the two stems at each end of the beam. Beams are therefore always rectangles or parallelograms, as shown in figure 5.12 :

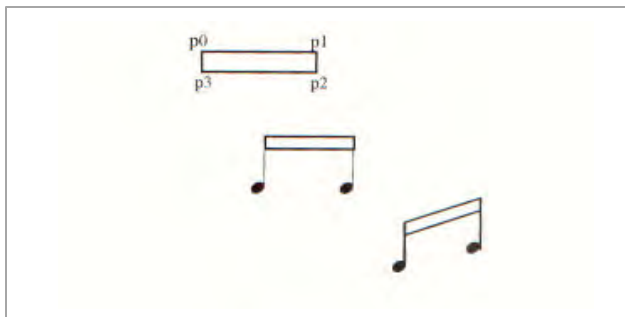


Figure 5.12 Beams implemented as rectangles or parallelograms.



### 5.4.4.3 Bitmaps

Certain symbols such as the bracket and brace shown in figure 5.13 cannot be created from graphics functions that are part of the Windows API and are not provided by TrueType fonts. These symbols must be drawn into a score using bitmaps that are loaded into memory by the OOTMN. Use of bitmaps allows the symbol to be stretched or shrunk to the size required by the application.

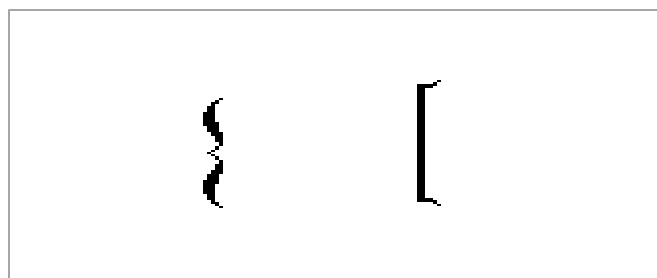


Figure 5.13 Brace and bracket symbols implemented as bitmaps.

## 5.5 MIDI Playback

MIDI output is accomplished by making use of the Maximum MIDI Programmers' Toolkit Lite. This toolkit provides a variety of functions for the control of a MIDI interface and its associated data stream. A detailed description of the Maximum MIDI Toolkit (MMT) is provided by the designer Paul Messick [Messick, 1997]. Example code provided with the toolkit that makes use of buffers to store MIDI data was altered to make use of the Borland C++ container classes. Allocating memory buffers within Windows is a much more complicated and error prone process than using predefined container classes. The Borland container classes also have the advantage of automatically sorting data. This ensures that events sorted by their associated timestamps are output to the MIDI port at the correct time.

Buffering of MIDI data is automatically provided by the MMT using a fixed-size internal buffer. It is the application programmer's responsibility to ensure that the buffer does not overflow. The MMT sends a message to the application's window when the internal buffer has been filled by MIDI event structures. The application must stop filling the MMT buffer and wait for a second message indicating that the buffer is ready to receive data. The OOTMN facilitates MIDI playback by

converting Note objects to MidiMessage objects. The MIDIMessage class is shown in figure 5.14, which encapsulate a MMT MidiEvent structure. MidiEvent structures encapsulated within each Note object are output to a selected MIDI port.

```

class MidiMessage {
    public : MidiEvent me;      // MMT MidiEvent structure
            MusicEvent* ptrMusicEvent;
            MidiMessage() { me.status=0; me.data1=0;
                          me.data2=0; me.time = 0; }
            MidiMessage(MusicEvent*, BYTE, BYTE, BYTE, DWORD);
            ~MidiMessage() { }

            DWORD operator == ( const MidiMessage& mm) const
                          { return me.time == mm.me.time; }
            DWORD operator < ( const MidiMessage& mm) const
                          { return me.time < mm.me.time; }
};

```

Figure 5.14 Implementation of Class MidiMessage.

Data member me is a MMT MidiEvent structure shown in figure 5.15. Overloading the = and < operators facilitates sorting of MidiMessage objects according to the starting time specified by the time field of the MidiEvent structure. A MidiMessage object also stores a pointer to its associated MusicEvent object which allows greater flexibility to be used in the algorithm that implements MIDI playback.

### 5.5.1 MIDI Synchronisation

MIDI output ports as implemented by the MMT are asynchronous receivers that output data as soon as it is received. or output the data at a specified time, offset from the previous received message. The serial MIDI data stream consists of messages where each message has a time tag specifying the time offset from the preceding message. The MMT buffer stores MidiEvents and ensures that they are output to the MIDI interface at the time specified by the time field. Musical events that

occur simultaneously have a time tag of zero and are processed sequentially at the hardware speed of 31.25 Kbaud. The human ear cannot separate musical notes that are very close together, perceiving a closely separated sequential stream as simultaneous notes [Pierce, 1983].

MusicEvent objects (Notes, Rests and Chords) are used to create corresponding MidiMessage objects. Each MidiMessage object translates into two MMT MidiEvent structures, a NoteON message at the start of a note and a NoteOFF at the end of a note. Rests do not translate into a MidiMessage, but are used to increment the elapsed time before the occurrence of the next NoteON message. Chords, consisting of multiple notes are processed by providing equal starting and ending times for each Note object.

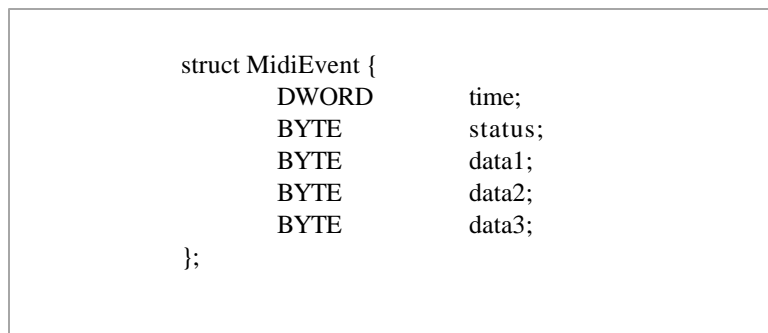
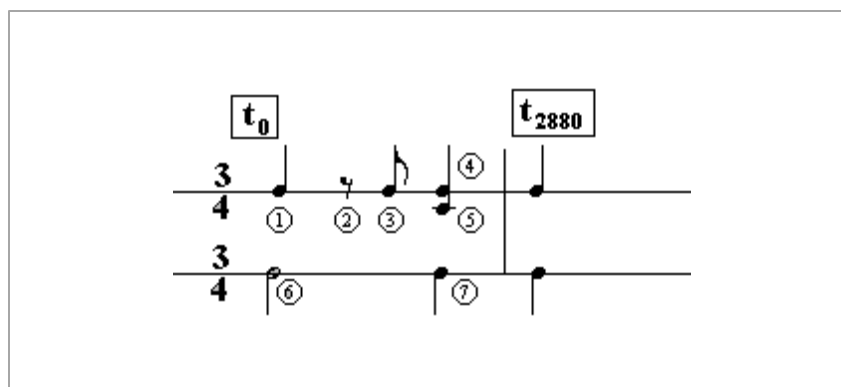


Figure 5.15 MMT MidiEvent structure.

The data1 field is the pitch defined as a keyboard key number. Data2 is the MIDI velocity used by the OOTMN to implement Dynamic objects. The time associated with each event is used to specify a MIDI timestamp offset. The process of time conversion is a two-step process illustrated using the rhythmic fragment of example 5.4 :



Example 5.4 Implementing MIDI timestamps.

Timestamp values are calculated by using the number of clock pulses per beat. Clock pulses are obtained by the MMT from a Windows timer. The maximum resolution is 960 clock pulses per beat. If the example above uses 960 pulses for each crotchet, the number of clock pulses per bar is  $3 * 960 = 2880$ . Thus the first bar starts at  $t_0$  and the second bar starts at  $t_{2880}$ . The first step in time calculation determines starting times for each event in a bar. The starting times for each event (numbered 1 to 7) offset from the start of the bar are as follows :

<u>Event</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>
Start t	0	960	1440	1920	1920	0	1920

A MIDI NoteON message occurs for each note at its corresponding starting time, with the exception of the rest at event two which only serves to increment the bar's counter by the duration of a quaver or 480 clock pulses. Each event's ending time is given by its start time + its duration. The quaver rest has no ending time as illustrated below :

<u>Event</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>
End t	0+960	-	1440+480	1920+960	1920+960	0+1920	1920+960

A MIDI noteOFF message occurs for each note at its ending time. By combining the noteON and noteOFF messages the table in figure 5.16 indicating the starting times of all MIDI messages is constructed :

<u>MessageTime</u>	<u>0</u>	<u>960</u>	<u>1440</u>	<u>1920</u>	<u>2880</u>
MIDI Message	ON 1	OFF 1			
		ON 6			OFF 6
			ON 3	OFF 3	
				ON 4	OFF 4
				ON 5	OFF 5
				ON 7	OFF 7

Figure 5.16 Calculating MIDI starting times.

The second step converts the starting times into offsets representing the difference between adjacent starting times. These offsets form the time tags of MIDI messages. The offset of an event  $n$  is :

$$\text{starting } t(n) - \text{starting } t(n-1)$$

Only the first event of a group of events occurring at the same time is tagged with an offset, all other events have a time tag of 0. Referring to the table above : ON 6, OFF 3, ON 4, ON 5 and ON 7 have time tags of 0. By storing MidiMessage objects ordered by increasing starting times, corresponding MIDI time tags are easily calculated giving the stream of MIDI messages of figure 5.17:

<b><u>MIDI Time tag</u></b>	<b>0</b>	<b>0</b>	<b>960</b>	<b>480</b>	<b>480</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
Event	ON 1	ON 6	OFF 1	ON 3	OFF 6	OFF 3	ON 4	ON 5	ON 7
							<u>2880</u>	<u>0</u>	<u>0</u>
							OFF 4	OFF 5	OFF 7

Figure 5.17 Calculating timestamps as MIDI offsets.

## 5.6 Design of Algorithms for the OOTMN

Certain toolkit operations require the use of fairly complex algorithms. Beaming of notes and MIDI playback presented challenging problems that required an algorithmic solution. All algorithms used by the toolkit, with the exception of the calculation of Bézier curves, are original algorithms.

### 5.6.1 Creating Beamed Groups

BeamedGroup objects consist of all beams used to beam a group of notes. An elementary formatting scheme places the beam from the first note's stem to the last note's stem. Designing an algorithm that attempts to format different groups of notes correctly would be difficult and unnecessary, given that beams can be manually adjusted to conform to any visual requirements. Output results from a formatting algorithm described by Assayag [Assayag, 1988] are not always

satisfactory, and influenced the decision not to include an elaborate formatting algorithm. Maximum flexibility is achieved by use of an algorithm that handles common cases, coupled with the ability to manually configure beams for a variety of less trivial beam configurations.

Beams are created from the quaver level to the level of the shortest note duration, the endpoints of the beam are determined by the endpoints of the stems of the first and last note of the beamed group. Flag objects that form part of stem objects are deleted and stems belonging to notes falling between the first and last notes are shortened or lengthened to terminate within the beam. This process is illustrated in figure 5.18 :

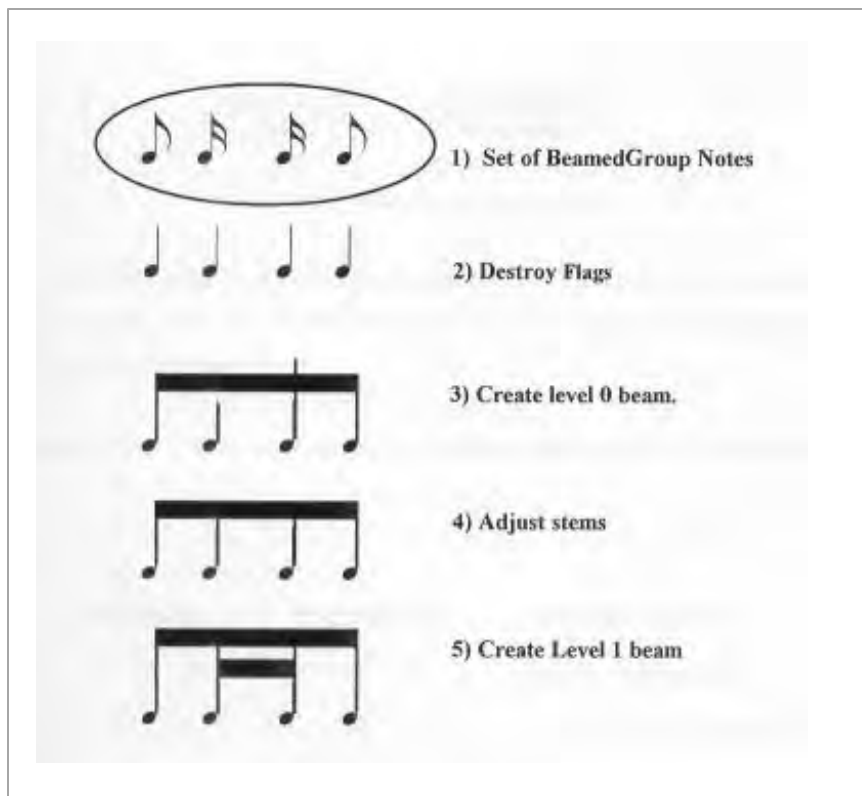
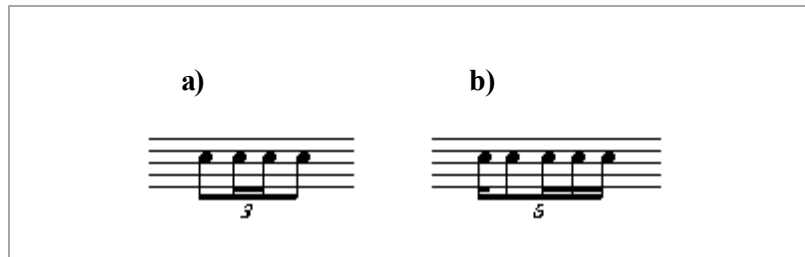


Figure 5.18 Creating BeamedGroup objects.

### 5.6.2 Calculating Grouplet Durations

The structure of grouplet objects was presented in section 4.6. Durations of notes that form part of a grouplet are calculated using the face durational value of the note and the four values that define a grouplet. The grouplet in example 5.5 a) consists of three  $\text{♪}$  face durations performed in the time of two  $\text{♪}$  actual durations. The second quaver face duration is further sub-divided into two semi-quavers. Example 5. b) is a quintuplet of five  $\text{♪}$  face durations performed in the time of four  $\text{♪}$  actual durations, with the second note having a larger face value than the grouplet face value :









Example 5.5 Durations of irregular groups.

Performed durations are always calculated as a portion of the actual duration of the entire grouplet. Shorter or longer values are calculated according to general rhythmic relationships that increase or decrease values by powers of two .

In example 5.5 a) Total actual duration = grouplet actual number \* grouplet actual duration  
 $= 2 * \text{♪}$

<u>Note Number</u>	<u>Face Duration</u>	<u>Performed Duration</u>
1	$\text{♪}$	Total actual duration / 3
2	$\text{♪}$	1/2 of Total actual duration / 3
3	$\text{♪}$	1/2 of Total actual duration / 3
4	$\text{♪}$	1/3 of total

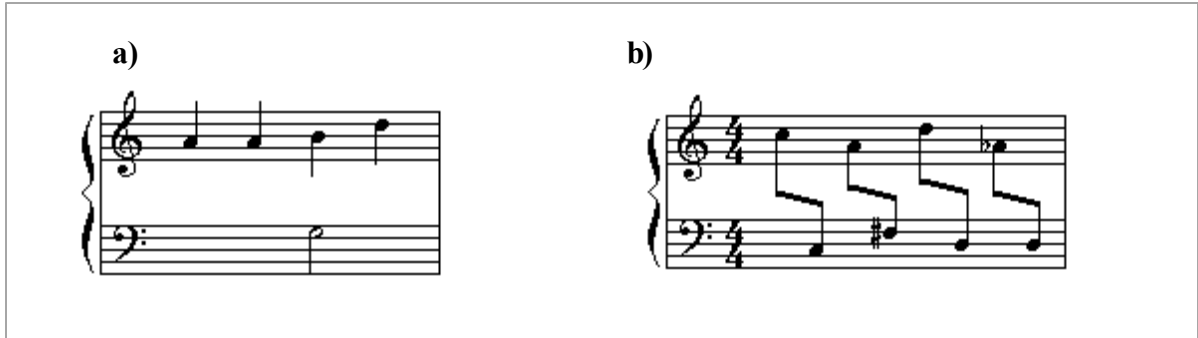
In example 5.5 b) Total actual duration = grouplet actual number \* grouplet actual duration  
 = 4 \* 

<u>Note Number</u>	<u>Face Duration</u>	<u>Performed Duration</u>
1		Total actual duration / 5
2		Total actual duration / 5 * 2
3		Total actual duration / 5
4		Total actual duration / 5
5		Total actual duration / 5

### 5.6.3 Implementing MIDI Playback

Calculation of MIDI timestamps was introduced in section 5.5.1. The algorithm that facilitates MIDI playback proceeds bar by bar processing each voice within a bar. Voices that are complete i.e. they contain MusicEvents that fill the entire voice according to the time-signature, are processed immediately with the associated MidiEvents being placed in the MMT buffer. Incomplete voices such as the voice in the lower staff of example 5.6 a) require additional processing. Each MidiMessage object has an instance relationship to its associated Note object. Notes that occur in an incomplete voice are aligned to Note objects in a complete voice that share the same x coordinate. In example 5.6 a) the note on the lower staff is aligned to the third crotchet of the upper staff. The OOTMN always attempts to assign timings to events using known event timings as a reference point.





Example 5.6 Incomplete bar and cross-staff beams.

Cross-staff beams such as those in example 5.6 b) are processed by attempting to process all the notes within each BeamedGroup object. Should all the voices in a bar be incomplete or the parsing process fail, the MusicEvents of an incomplete Voice are assumed to start at the beginning of the bar.

The playback algorithm processes a single bar at a time using the above procedures providing a powerful and flexible playback environment. Initial attempts to develop a playback algorithm followed the NIFF specification by associating a timestamp with each bar. Each bar contained a timestamp attribute that indicated the total elapsed time. This attribute was found to be redundant, starting playback at a bar other than the first bar requires the timestamp to be ignored as MIDI playback always starts from time zero. It is unclear as to why the NIFF standard associates a timestamp with each bar.

Chapter five examined the C++ implementation of the OOTMN, as well as the algorithms required by specialised notation functions and MIDI playback. The next chapter describes the practical use of the OOTMN by examining the development of applications built using the toolkit for music notation.

## Chapter 6

### Examples of Toolkit-Created Applications

#### 6.1 Designing Applications using the OOTMN

All software design methodologies emphasise a separation of the user-interface from the actual application program. This approach requires that the user interact with the program by means of well-defined functions that link user commands to the application code. As an example of this separation consider the use of a mouse to interact with a program. All code that manipulates, or pertains to the mouse in any way resides in the user-interface of the software. The user-interface determines the information required by the application (usually the state of the mouse buttons or the mouse position) and provides this information to the application. The design of the OOTMN emphasises this separation by providing methods that allow an application program to manipulate the objects provided by the OOTMN. Figure 6.1 shows this interface which was partially included in the complete object model presented at the end of chapter four :

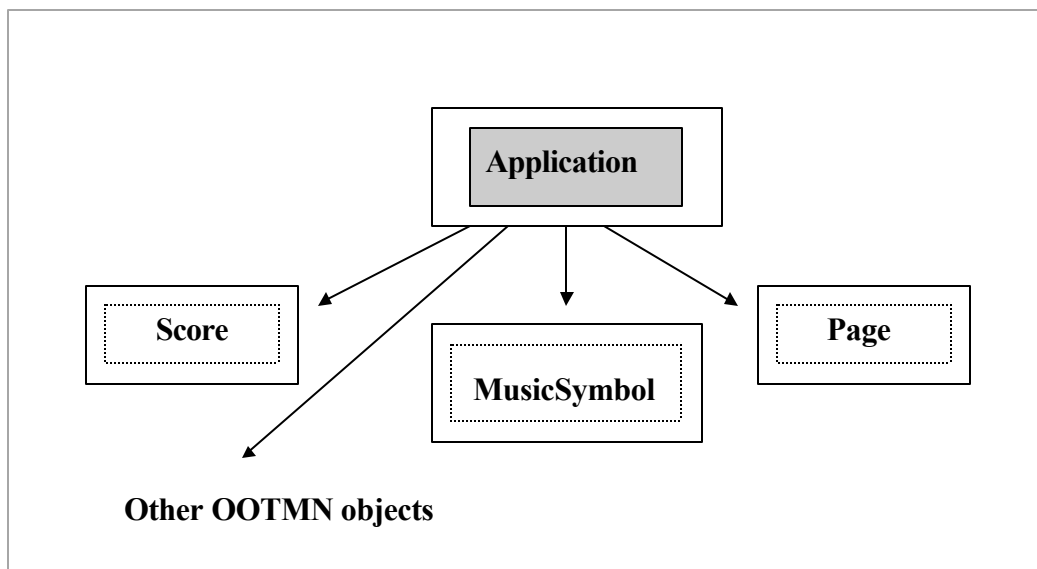


Figure 6.1 Application interface to the OOTMN.

Much of the functionality provided by the toolkit is made available by methods in the Score, Page and MusicSymbol objects. Global settings are stored in the Score object, searching and hit-testing starts from the Page level. The updating of an object's position, as well as the positions of all other symbols that are spatially dependent on the object are performed by MusicSymbol objects. More specialised operations are made available by user interaction with specific OOTMN objects. Examples of how a Microsoft Windows user-interface interacts with the toolkit can be found by studying the 'pagewin.cpp' files which form part of the notation and harmony tutor example applications presented in this chapter. Especially interesting are the OWL functions within the user-interface that process mouse and keyboard input. These functions invoke the OOTMN methods that perform searching and the updating of object positions.

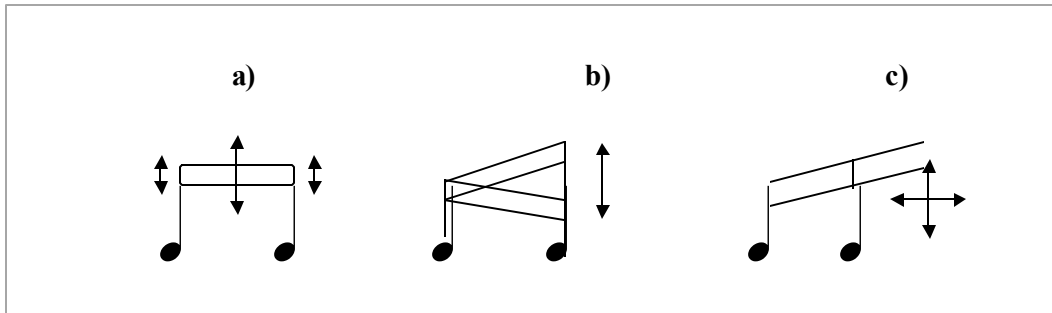
## **6.2 Notation and Composition Software**

The notation and composition software application provides a rigorous testing environment for the OOTMN. Apart from testing the general correctness of the various methods associated with objects in the OOTMN, this application also tests whether the individual objects are able to provide the levels of functionality required to successfully implement complex music notation software. Not every aspect of the application is discussed in detail. The following sections provide a summary of the available commands and menu options. Although the program does not attempt to provide the functionality of a commercial application, it has exceeded expectations in some of the tasks that it performs.

### **6.2.1 User Interface Design**

Given a pencil and a piece of paper, the elementary school music student and the accomplished composer perform many identical, elementary tasks. These tasks should exist at a high level in the user interface, complex operations that are infrequently used are hidden at lower levels. All operations are implemented as simple click-and-drag operations within a WYSIWYG (What you see is what you get) environment. The user-interface is similar to the interface employed by 'Encore' which was discussed in chapter three. As pointed out in chapter three, notation programs that use complex menu operations to perform tasks are not conforming to the visually-oriented design of the

Windows environment. The possible operations on Beam objects shown in example 6.1 serve as an example of this design philosophy :



Example 6.1 Dragging Beam objects.

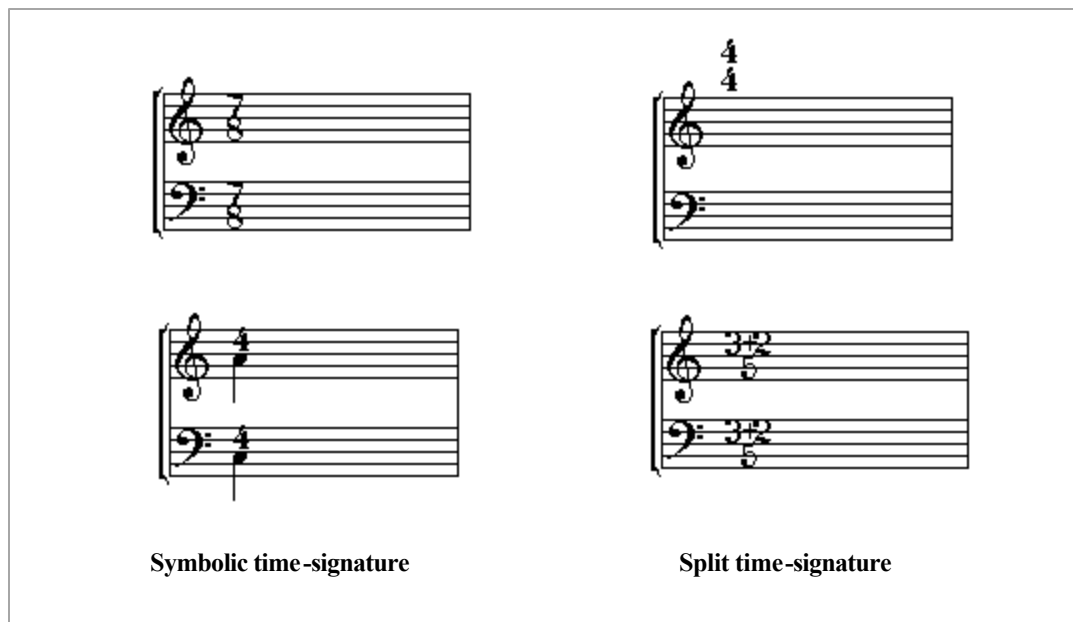
Beam positions are adjusted by using the left-hand mouse button to click on a beam and drag it to the required position while holding the button down. By default, beams are adjusted vertically by dragging the middle or either end as shown in example 6.1a). All beams belonging to a beamed group are automatically adjusted. Use of the <shift> key adjusts individual beams vertically as shown in example 6.1 b). Pressing the <ctrl> key allows individual beams to be extended both vertically and horizontally as illustrated in example 6.1 c). These operations allow modern conventions such as the extended beams and fan beams of example 6.2 to be created by simply extending the basic operations that adjust conventional beams. The design of the OOTMN does not prevent this logical extension of functionality.



Example 6.2 OOTMN examples of extended and fan beams.

Time-signatures are modified in a similar way. A complete time-signature may be dragged to any position, while use of the <shift> key adjusts the position of only the upper or lower value of a time-signature. The symbolic and split time-signatures of example 6.3 must be implemented by selecting

menu options that set the display method (numeric or symbolic) for a time-signature, or obtain the required values for a split time-signature.



Example 6.3 OOTMN created examples of time-signatures.

### 6.2.2 Using the Notation and Composition Program

Objects are inserted by selecting them from the different palettes. Chords are created by entering a note at the same x co-ordinate as an existing note. All objects may be selected by clicking on the object or one of its hitspots using the left-hand mouse button as outlined in the previous section. Figure 6.1 provides a summary of the hitspots for each symbol. MusicSymbol objects (that are not LineSymbols) have a single hitspot located at the centre of the symbol. LineSymbols listed in figure 6.2 can have their hitspots made visible by pressing the <Tab> key. Pressing <Tab> a second time hides the hitspots.

<u>Symbol</u>	<u>Hitspot/s</u>
<b>Staff</b>	Top left-hand corner.
<b>Barline</b>	Top or bottom of barline.
<b>Note</b>	Centre of notehead.
<b>Rest</b>	Centre of rest symbol.
<b>Chord</b>	Any Note belonging to the Chord.
<b>Dynamic</b>	Centre of dynamic symbol.
<b>Clef</b>	Centre of clef symbol.
<b>Time-signature</b>	Centre of top or bottom symbol.
<b>Key-Signature</b>	Centre of key-signature.

Figure 6.2 MusicSymbol objects and their associated hitspots.

<u>LineSymbol</u>	<u>Hitspot/s</u>
<b>Beam</b>	Centre, left, or right of beam.
<b>Hairpin</b>	Three points forming the endpoints of the two lines forming the hairpin.
<b>Slur, Tie, Phrase</b>	Endpoints and two internal Bézier control points.
<b>Stem</b>	Endpoints of the line forming the stem.
<b>Barline</b>	Endpoints of the line forming the barline.

Figure 6.3 LineSymbol objects and their associated hitspots.

Symbols can also be selected by double-clicking the symbol with the left-hand mouse button. Symbols selected in this manner are highlighted. An appropriate symbol must be selected in this way before invoking a menu corresponding to the selected symbol. All symbols are moved by dragging them with the left-hand mouse button held down. Figure 6.3 provides a summary of the

default actions, as well as the command keys that provide enhanced capabilities when dragging symbols with the mouse :

<u>Symbol</u>	<u>Command Key</u>	<u>Description</u>
<b>Staff</b>	none	Moves the staff, automatically aligning all system staves on the x-axis. The y-axis positions of all staves below the staff being moved are automatically adjusted. All bar sizes, as well as MusicEvent positions within a bar are automatically adjusted..
	<shift>	
	<ctrl>	
<b>Barline</b>	none	Sets a barline's position on the x-axis.
	<ctrl>	Sets top or bottom y-axis position, as well as x-axis position of a barline
<b>Beam</b>	none	Adjusts all beams of a beamed group on the y-axis.
	<shift>	Sets right or left-hand y-axis positions of individual beams.
	<ctrl>	Sets right or left-hand x and y-axis positions of individual beams.
<b>Chord</b>	none	Moves the entire chord on the x-axis.
	<shift>	Moves individual chord notes on the x and y axes.

Figure 6.4 Options controlling the movement of symbols.

Details of the methods that implement these options can be found in the OOTMN reference guide given in Appendix A. Additional commands recognised by the OOTMN include the following :

<u>Command</u>	<u>Action</u>
<esc>	Moves to select mode after insertion or deletion of a symbol, or cancels the selection of a symbol.
<del>	Deletes the selected symbol that is highlighted.
<b>Group selection</b>	Hold the right mouse button down and drag across a group of notes.
<B>	Beams a group of selected notes (see group selection above).
<G>	Creates a grouplet from a group of selected notes. (see group selection above).

Figure 6.5 OOTMN command summary.

Clicking the left-hand mouse button anywhere on a page cancels the selection of a symbol.

### 6.2.3 Secondary Bars, Secondary Stems and Split-Stem Chords

A secondary bar must first be created by double-clicking the bar that is to have a secondary bar with the left-hand mouse button, and selecting the secondary bar option from the bar menu. To make the secondary bar visible, the right-hand mouse button is double-clicked in the bar containing a secondary bar. Double-clicking the right-hand mouse button within the bar toggles the view from the parent bar to the secondary bar and vice-versa.

Secondary stems (discussed in section 2.2.7.1), as well as note's primary stem can be created or deleted by double-clicking a note to select it, activating the Note menu and setting the stem attributes as desired. Split-stem chords (illustrated in example 2.16) are created by graphically altering an existing Chord object. A split-stem chord can be created by the following sequence of events :



- a) creating a Chord by inserting Notes at the same x co-ordinate.
- b) dislocating the desired Note or Notes from the Chord by dragging them with the <shift> key depressed.
- c) adding a Stem and / or secondary stem as required.
- d) dragging the Stem or Stems to the required position.

### 6.2.4 Toolbar and Menu Options

Toolbar options are shown in figure 6.6. Voice buttons select voices for both insertion of notes and rests, as well as playback. A (All) plays all voices but inserts music events into voice one. Auto stem direction sets the appropriate stem direction according to the position of the note on the staff. A specific stem direction can be set using the stem-up or stem-down buttons. The arrow button (corresponding to the <esc> key) is used to cancel the insertion or deletion of objects, allowing objects to be selected and/or dragged.

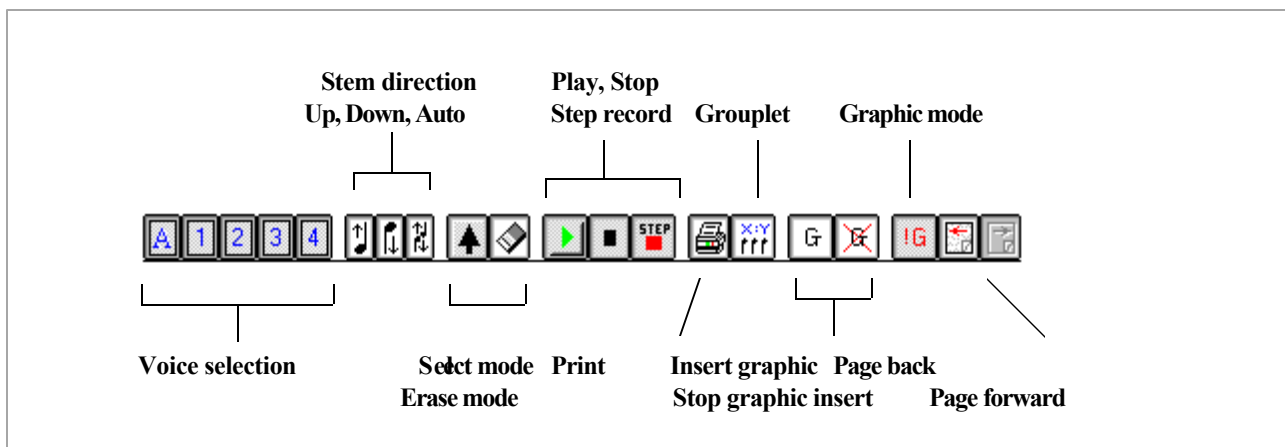


Figure 6.6 The Score Application toolbar.

Step record allows step-recording from a selected MIDI input device. Note durations must be selected from the note palette before playing the required pitch on a MIDI instrument. The stop button ends both playback and recording. The grouplet button opens a dialog box that allows specification of the desired grouplet. Insert graphic / stop insert graphic allows context-sensitive

symbols such as accidentals and clefs to be inserted as graphics. Graphics mode destroys context-sensitive relationships and treats each symbol as a graphic object.

Staff, Bar and Note menus require that appropriate object be selected by double-clicking with the left mouse button before activating the appropriate menu. The palette menus shown in figure 6.7 activates palettes containing different categories of symbols :

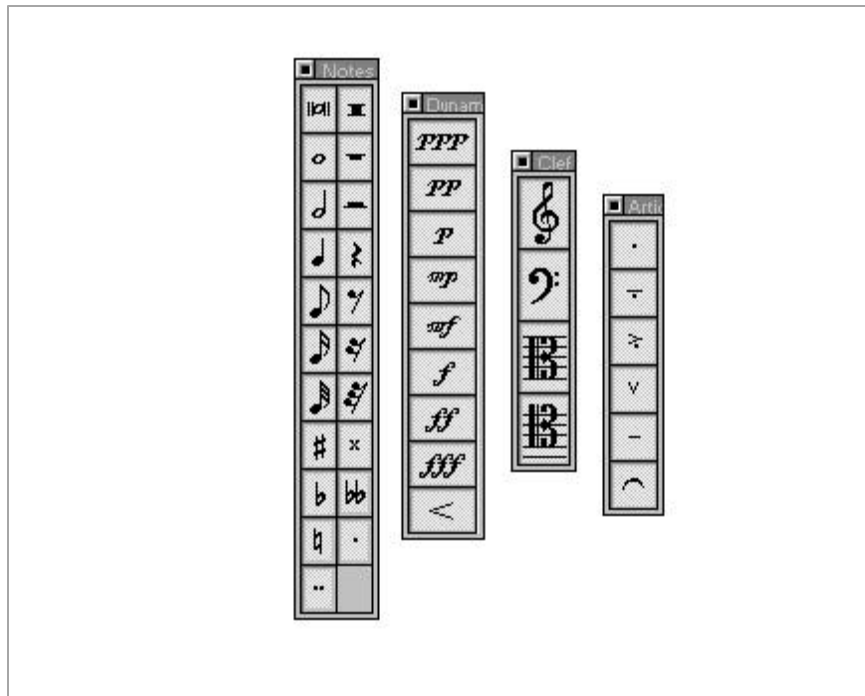


Figure 6.7 Palette menus for different music symbols.

Hairpin and slur symbols are the last items within the dynamic and articulation palettes. Some of these symbols can be inserted to form context-sensitive relationships as described in the next section.

### 6.2.5 Context-sensitive Symbolic Relationships

Articulation and dynamic symbols are inserted as context-sensitive symbols by adding the symbol to an existing note by selecting the required articulation or dynamic and inserting it directly onto the

note's notehead. Hairpins and slurs require a relationship with two Note objects. A hairpin or slur is inserted by selecting the symbol from the appropriate palette and then clicking on two note objects.

This process creates instance connections between the two Note objects and the Hairpin or Slur object where the notes define the starting and ending positions of the hairpin or slur. Moving either of the Note objects automatically adjusts the position of the dependent object. The dependent object can be moved on its own, creating a spatial offset from the object on which it is dependent. This offset relationship is maintained by the OOTMN.

### **6.2.6 MIDI Playback and Recording**

A MIDI output port must be opened using the MIDI menu before starting playback. All available MIDI ports are detected by the application and made available to the user. Clicking on a bar with the left-hand mouse button places the caret within the selected bar. Playback begins from the currently selected bar, with only the selected voice or voices being played. Similarly, a MIDI input port must be opened using the MIDI menu before starting to record. Input begins from the currently selected bar as indicated by the position of the caret. Duration is selected from the note palette, pitch being determined by the keyboard key number of the MIDI ON message received by the OOTMN.

## **6.3 A Harmony Tutor**

Existing commercial applications for theoretical music education concentrate on elementary music skills. Skills such as note recognition, durational comprehension, and the identification of intervals and chords form a relatively simple problem domain. Mapping of the subject domain to computer software is often further simplified by using exercises having a true/false or multiple choice format. Such exercises have little value where reasoning skills and conceptual insight are required from the student. Higher-level skills such as harmony, melody writing and counterpoint require evaluation mechanisms that are difficult to implement in software. The harmony tutor application is limited in

scope and uses simple evaluation functions. Evaluation of extended exercises having a richer harmonic vocabulary would require the use of an expert system.

Harmony is the study of chords, including their construction and their relationships to other chords. This sub-discipline of music theory extends from elementary principles that may be taught in the secondary school to non-trivial research topics. The tutorial provides introductory-level tuition in the completion of four-part perfect cadences where the soprano and bass voices are given. Use of the tutor is covered in section 6.3.4.

### **6.3.1 Selecting a Development Environment for a Harmony Tutor**

In addition to the interactive exercises created by the OOTMN, it is desirable to be able to include tutorial matter consisting of text and musical examples that are only graphical images. The Microsoft Windows Help system was investigated as a possible development environment. Help files for Windows applications are created separately from the application using a compiler placed in the public domain by Microsoft. In addition to the familiar hypertext links, extended functionality within help files can be achieved using third-party products that exist as dynamic link libraries. These libraries create additional windows within help files known as embedded windows. The envisaged application would make use of an embedded window containing the interactive component of the harmony tutor. Text-based tutorial matter would surround the embedded window. After examining more than fifteen packages that provide development environments for the creation of help files, and studying the Microsoft Help Development Kit (freely available as part of the compiler), the idea was reluctantly abandoned. According to Arnote [Arnote, 1996], embedded windows are never able to receive keyboard or mouse focus as the main help window always has the input focus. Mouse input can only be achieved by an indirect process that transfers information from the main window to the embedded window. Cessna [Cessna, 1996] advised that an alternative solution whereby the help window is embedded in an application window would not be feasible.

Java is an interpreted, object-oriented programming language specifically developed for writing applications that execute within a web browser. The Java Development Kit is freely available from the developers, Sun Microsystems Inc. [Sun Microsystems, 1995] Java also allows functions written in other programming languages, known as 'native methods', to be embedded in Java applications.

Such an environment has the potential to provide an ideal development environment for educational music software. The text capabilities of HTML, coupled with Java's ability to integrate programming code such as the OOTMN could allow a music education application to execute within a web browser. Educational software that is compatible with world-wide-web browsers has many advantages. Web browsers are freely available, and allow a separation between the development and execution environments of the software product. Distributed control, and updating of software is also possible via the Internet. According to Wentworth [Wentworth, 1997], the use of native methods within Java is an aspect of the language that has not been fully and freely implemented, as the unrestricted use of native methods poses serious security risks on the Internet. The possibility of a Java environment not being able to provide the required level of functionality resulted in this idea not being seriously investigated.

### **6.3.2 Authoring Packages**

Software packages that facilitate the creation of software for interactive education or presentation purposes are referred to as authoring software. An initial Internet search for suitable authoring software produced disappointing results; available packages were either of poor quality, or demonstration versions having extremely limited functionality. Three packages that were potentially useful were eventually discovered and evaluated.

Everest is a sophisticated, object-oriented authoring package from InterSystem Concepts Inc. [InterSystem Concepts, 1998]. Formula Graphics developed by Formula Software [Formula Software, 1988] emphasises graphic operations. MultiMedia Builder from MediaChance Software [MediaChance Software, 1988] is the least sophisticated of the three packages, but is easy to use and provides a solid set of core functions that are adequate for most applications. Problems encountered while using Everest and Formula Graphics resulted in the decision to make use of MultiMedia Builder for the harmony tutor. These problems included erratic z-order (z-axis from the monitor screen to the viewer) placement of graphics by Everest and the hiding of the harmony tutor window by Formula Graphics. Example 6.1 illustrates tutorial matter created with Multimedia Builder and the Harmony Tutor executing simultaneously, providing the illusion that the application is part of the tutorial :

Chapter 2      **Chords in Root Position**

The most common chord progressions have roots of the chords that are a perfect fourth apart.

Figure 6.8 Multimedia Builder and the Harmony Tutor executing simultaneously.

The harmony tutor is written as a separate Windows application that executes in its own window after being activated from within MultiMedia Builder using the software's ability to launch external applications.

### 6.3.3 Using the Harmony Tutor

Exercises are completed by selecting a note from the note palette and adding alto and tenor voices to the two given chords. A new exercise is selected by clicking the <Next> button. Soprano and alto voices share the top staff, tenor and bass voices share the bottom staff. The appropriate voice must be selected from the voice buttons on the left before inserting a note. Evaluation functions make use of note representations using the name and pitch-classes discussed in section 5.4.2. Errors that are detected and indicated by the evaluation functions of the program include :

- a) incorrect notes i.e. notes that do not belong to the chord.
- b) doubling of the major third in a chord.
- a) parallel fifths and octaves between any voices of adjacent chords.
- b) a falling leading-tone within the dominant - tonic chord progression.

The user is able to clear all error messages, correct the errors by moving the incorrect note (or notes) and allow the program to evaluate the exercise again. Program instability often arises from attempting to correct notes - the reasons for this are not clear as the application is built from the same library as the notation program.

An interesting aspect of the harmony tutor is that the default spacing between staves was found to be insufficient. Alto or tenor voices (especially the tenor) may require up to three leger lines. To accommodate leger lines, four-part harmony requires that the two staves forming a system should be spaced further apart than is the norm for notating most instrumental music. Additional functions that set the spacing between staves within a system had to be added to the toolkit to allow the programmer to control the spacing between adjacent staves. Unlike the notation software, the Harmony Tutor required that an additional class be created to represent the subject matter in a meaningful way. Class `HarmonyChord` encapsulates the four notes that form a four-part chord. Evaluation functions are able to evaluate the structure of a `HarmonyChord` object as well as the progression that exists between two `HarmonyChord` objects.

## 6.4 An Embedded Notation Example

This example illustrates the use of a subset of the toolkit's features to quickly and easily create music notation in a visual form only. Figure 6.2 illustrates the simplicity of the code below taken from the example application. Because of its simplicity, this score has certain restrictions. It cannot be played and does not support any form of user interaction.

```

ScorePtr = new Score(this, 1, 44, 1, 3, "");           //create a Score
ptrPage = new Page(this, 1);                          //create a Page
ScorePtr->ptrPageArray->Add(ptrPage);                 //add the Page to the Score
Note* ptrNote = new Note(4, 0, 0);                   //create a Note
ScorePtr->InsertNote(ptrNote, 1, 1, 1);               //add the Note to the first voice
                                                    //of the first bar of the first staff.

ptrNote = new Note(4, 6, 0);
ScorePtr->InsertNote(ptrNote, 1, 1, 1);

```

Figure 6.9 Code example of embedded notation.

Information concerning the creation of applications using features of the ‘basic’ mode of the OOTMN can be found in appendix A. This example illustrates the ease with which simple melodies can be created using the OOTMN. In practice, it is probably easier to paste a graphic image from a notation program into an application for illustrative purposes than to create the fragment using the toolkit.

This chapter introduced three applications created with the OOTMN. Chapter seven evaluates the OOTMN by examining the strengths and weaknesses of the toolkit, as well as examining the project as a whole.



## Chapter 7

# Conclusion

A comprehensive, objective evaluation of the toolkit for music notation is a difficult task from the perspective of the designer. Faced with a large number of features, some fully implemented, others capable of further extension, it is difficult to place the toolkit as a whole in perspective. Many design aspects of the toolkit can be further refined, and undergo further testing. No apology is made for these limitations, as the development of a commercial strength library or application is not the aim of this study. The following sections examine the functionality, design and limitations of the toolkit.

A discussion of the software engineering insights gained from the project and future development possibilities are also included in this chapter.

### 7.1 Functionality of the OOTMN

The OOTMN attempts to illustrate that careful design using object-oriented technology produces a powerful, yet flexible library that offers natural solutions to many music notation problems. Flexibility, as well as a wide range of functionality (illustrated in the previous chapter) are characteristics provided for by the design. Extensions to the capabilities of the toolkit are achieved within the basic design, which elegantly captures the structural and semantic relationships which exist within music notation. The strength of the toolkit lies in the rigour of its design, which always centres around the subject matter. Functional aspects of the toolkit, such as hit-testing, searching and context-sensitive positional updates discussed in chapter five, are built on top of the hierarchy of musical objects by extending the hierarchy to accommodate these features. These enhancements do not detract from the integrity of the representation of traditional music notation. MIDI realisation is also implemented within the model of music notation. Aspects of the OOTMN not directly concerned with representing music notation never influenced, nor preceded the design, they were always accommodated within the design.

### **7.1.1 Simplifying Application Development**

As a library, the OOTMN is intended to simplify the development of any application that makes use of music notation, including applications that require interaction with the user. The score processing application discussed in chapter six is difficult to implement, as it requires a complex user interface, and a large functional capability from the toolkit. The Harmony Tutor is an application that requires less complex processing as far as notation is concerned. Implementing the notational requirements of the harmony tutor was very easy. Complexity resides in the evaluation of the exercises, which form part of the subject matter and are not related to the toolkit's implementation of a representation of music notation.

In considering the effectiveness of the OOTMN it is encouraging to note that most of the time spent during this study was concerned with problems within the Windows environment, especially the Borland development environment. Relatively little time was spent solving problems within the toolkit itself or use of the MMT to implement MIDI playback. The OOTMN greatly simplifies application development in that only the user interface and any specialised functionality required by the application need be coded. Although the toolkit provides invaluable functionality, it cannot hide the inherent complexity of the subject matter.

### **7.1.2 Manipulating Notation Symbols**

Chapters four and five illustrated that a structural hierarchy of objects and their relationships is only the first step in creating a useful object-oriented representation. Successful integration of functional requirements within the structural hierarchy provides expressive power to the system as a whole. Hit-testing, searching and updating positional relationships between symbols require complex operations that are implemented in a high-level manner. The implementation hides complexity and emphasises the clear separation of an application's user interface from the internal functions of the toolkit. Applications interact with the internal representation of a musical score by means of clearly defined functions that attempt to operate at the highest level of abstraction possible. The user need not be aware of the internal functionality of the OOTMN, nor the structure of the underlying object hierarchy.

### 7.1.3 Extensions to Traditional Notation

One of the most satisfying features of the toolkit is the ability to graphically manipulate barlines, time-signatures, staves, beams and stems to achieve modern notational conventions. These conventions are not obtained at the expense of destroying underlying context-sensitive relationships, allowing MIDI playback to be independently developed to accurately interpret the visual score.

Integrating traditional context-sensitive relationships with a partial graphical freedom, provides for a flexible environment. Features characteristic of modern notation do not have to be separately implemented, as they are built on the conventions of traditional notation. The example score by Lutoslawski shown in example 4.1, as well as the beam, time-signature and split-stem chord examples discussed in section 6.2 are easily created (with one exception mentioned in section 7.2) using the example notation program. Most commercial notation programs do not cater for modern notational practices; those that do (such as Finale discussed in chapter three) require complex operations to achieve the desired notational constructs.

## 7.2 Design of the OOTMN

The design of the current version of the OOTMN can be improved. Flaws in the design, as well as aspects of the design that can be enhanced became apparent during implementation of the toolkit. The use of bounding rectangles associated with staves discussed in section 5.3.3 has a side-effect. MusicEvent objects belonging to a particular staff that are moved outside of the staff's bounding rectangle are lost. Searches are directed by whether an object falls within or outside of a particular bounding rectangle. There are two possible solutions to this problem. The searching process can perform an exhaustive search or, the storage relationships of objects can be revised. Revising the object hierarchy is preferable as it enhances the toolkit as a whole. A clear separation of the visual and functional roles of objects would eliminate the need for staves to have bounding rectangles. MusicEvent objects (having a functional, non-visual role) are derived from the base class MusicSymbol and stored in the Voice objects to which they belong. MusicSymbol objects (having a functional and visual role) representing all the symbols contained within a Page should be stored within the Page object. Each Note object would have two representations, a symbolic representation at the Page level and a sonic representation at the Voice level.

The relationships between Bars, Voices, and MusicEvents used in the present design can possibly be improved. As outlined in chapter four, Bar objects consist of multiple Voices. Voice objects should possibly be implemented as higher-level abstractions that are not constrained to exist within Bars, as musical voices exist outside of, and across bar divisions. Bars would then consist of MusicEvents that would have instance connections to Voices. Although this representation is intuitively more satisfying, it does not alter, nor enhance the functionality of the OOTMN in any way.

As discussed in section 5.4.3, class LineSymbol is derived from class MusicSymbol. This derivation implies that a LineSymbol object has a reference position (given by the Position attribute of class MusicSymbol), as well as an array of points defining the symbol in class LineSymbol. The first point in the array contains the same x and y co-ordinates as the position attribute inherited from LineSymbol. It may be preferable to avoid this duplication by not deriving class LineSymbol from class MusicSymbol. Considering the four points determining a slur as an example, there is no reason why any single point should form a primary reference point. This redundancy is the result of a design decision that derives all symbols from class MusicSymbol. Staff objects should possibly be derived from class LineSymbol, providing hotspots at the top left-hand and top right-hand points of a staff. Staves can then be dragged from the left or right-hand sides allowing the notation of scores such as the Lutoslawski example of figure 4.1 that require staves that are shorter than the right-hand page margin.

### **7.3 Limitations of the OOTMN**

MIDI functionality is limited to correct playback of notes, rests and dynamic symbols. Support for different MIDI channel and patch configurations are provided through the channel and patch attributes of Staff and Voice objects which presently make use of default settings. Hairpin and Articulation objects do not currently influence the MIDI data stream during playback. They can easily be extended to provide MIDI capabilities in the same way that MIDI functionality was added to Dynamic objects after they were implemented as visual symbols. MIDI capabilities should ideally form part of the toolkit. Limitations regarding the use of the MMT within a segmented memory model discussed in chapter five prevented a MIDI implementation from within the toolkit.

## 7.4 The Borland Development Environment

The Object Windows Library is fairly intuitive to use and provides a robust object-oriented development environment. Unfortunately, a steep learning curve is encountered that is the result of poor documentation rather than the OWL environment itself. Poor documentation encourages an undesirable trial-and-error approach to application development. Despite these shortcomings, Borland provides window management functions and a mechanism to respond to Windows messages that allows for a compact and clear layout of application code.

Use of the Borland container classes created serious development problems that have never been satisfactorily resolved. Container classes used in the OOTMN are defined as follows :

```
typedef TContainerClass <object> Container    or,
typedef TIContainerClass <object> Container
```

The first definition defines a direct container that stores objects using the storage scheme defined by TContainerClass. These storage schemes are modelled on classical data structures. The second definition defines an indirect container that stores pointers to objects indicated by the 'I' appended to the name of the storage scheme. In attempting to circumvent container class problems the following definition was also used :

```
typedef ContainerClass <object*> Container
```

This definition explicitly forces the storage of pointers to objects and requires that the programmer manage memory allocation by deleting the objects referenced by the pointers stored in the container as well as removing the pointers from the container. Despite attempting to use these three container

definitions with a variety of different methods provided for the management of container classes, problems occurred that were not evident when testing smaller programs that prototyped OOTMN object storage. It is probable that Borland containers are not always able to accurately determine memory requirements resulting in the corruption of memory when deleting objects from containers or removing the container object from memory.

## 7.5 A Software Engineering Perspective

It is most valuable to consider some of the insights gained from the development of the OOTMN.

In particular, what aspects of the design and implementation processes would be approached differently ? The following is a summary of some of these aspects :

### 7.5.1 Prototyping

Prototypes are partial implementations that provide an evaluation of a design. However, partial solutions may not reflect the true characteristics of the system which may only become apparent with a more detailed implementation. A critical level of implementation which is not easy to determine is required before any worthwhile appraisal can be made. A lack of familiarity with the functioning of the OWL resulted in early prototypes that did not have sufficient detail to rigorously test the design. The complex development environment provided by Microsoft Windows creates an implementation overhead that focuses the developer on the environment rather than the design. Prototyping should make use of a skeleton user interface and attempt to partition the object model as discussed in the next section. Because of the complex relationships that exists between many of the objects, a detailed implementation of a portion of the object model is preferable to a partial implementation of the entire object model.

### 7.5.2 Partitioning the Object Model

Portions of the object model that exhibit a relatively high-degree of independence can be developed as an independent functional unit. MIDI implementation was greatly simplified due to the simple relationship between Note objects and MidiMessage objects that occurred naturally in the object model. The only other objects within the OOTMN that directly influence MIDI playback are Dynamic objects which only have an indirect influence via Note objects played at a given dynamic level.

Further examination of the object model reveals a natural partition between Voice and MusicEvent objects. A Voice is only related to two other objects, it is part of a Bar and Consists of MusicEvents. By contrast, a Bar is related to eight other objects. Development of the object model from the Voice level downwards (with simple staff and bar objects required by MusicEvents) could

have proceeded independently of the portion of the object model from the Bar level upwards. These partitions could be combined by simply adding the spatial dependencies between MusicEvent and Staff objects.

### **7.5.3 Simplifying Software Development**

Aspects of the development environment that create problems that may not be satisfactorily resolved should be abandoned at an early stage if at all possible. Unstable software tools result in an enormous loss of time and productivity. The Borland container classes should have been replaced by a different container class library or a container class specifically written for the OOTMN. The time required to develop a container class may have been significantly less than the time spent attempting to utilise the Borland classes.

Windows95, being a multi-threaded rather than a multi-tasking operating system provides a poor development environment. Errors during program development freeze or crash the machine, requiring that the machine be rebooted. WindowsNT or IBM's OS/2 are true multi-tasking operating systems that do not allow applications to corrupt the operating system. Such operating systems would provide a more productive development environment.

Use of memory should be closely monitored, especially when developing Windows applications, as memory leaks (failure to de-allocate heap memory) and memory corruption can have catastrophic consequences as mentioned above. Use of a freely available memory monitor, 'MemWatch' developed by Terry Richards [Richards, 1995] indicated some memory leaks that could be rectified. Unfortunately, the utility also indicated memory leaks in Borland example applications. Documentation provided with the utility indicated that memory allocation and de-allocation performed by the OWL could not always be accurately determined.

## **7.6 Future Directions**

The OOTMN can be extended to include a number of features that are not currently implemented. Support for storage formats, the addition of editing functions and the real-time transcription of MIDI input are the major extensions that are necessary to support commercial score processing applications. Development of a music font that aligns symbols according to reference points

contained in the NIFF specification and the use of a memory device context to increase the speed of display updates would also be useful extensions.

### **7.6.1 File Formats**

Translation of Score objects into different file formats would allow the storage of data, as well as further processing of the data representing a musical work. Useful storage formats are the NIFF and MIDI formats, which have a standard representation. Although MIDI does not provide for an adequate representation of music notation, Score objects are easily translatable into MIDI files. Commercial notation packages offer real-time conversion of MIDI input to music notation, although the results are often unsatisfactory. Durational quantisation of MIDI input (i.e. determining rhythmic values according to a reference duration) is an algorithmic process that is independent of symbolic conventions. Translation into other music representational formats such as DARMS [Brinkman, 1990] allows musicological research into the structure and stylistic features of works. This is an extremely important area of theoretical research that has not been sufficiently explored due to the absence of support for standard representations. NIFF can provide exciting opportunities for theoretical music research.

### **7.6.2 Algorithmic Enhancements**

Conversion of real-time MIDI input, and automatic graphical formatting capabilities are desirable features, unfortunately, both of these are non-trivial tasks. The OOTMN performs automatic formatting of the contents of a bar using a simple spacing algorithm based on the rhythmic duration of MusicEvent objects. Correspondence with Walsen [Walsen, 1998] revealed that these are two features of the Notation Engine discussed in chapter three that are still under development.

Score structure and the structure of individual pages present interesting challenges. Pages may be incomplete (i.e. contain fewer staves than the first page of the score) or, have additional staves added to them within a single score. Such a score must keep track of the staff names and MIDI channel assignments across pages having different structures. The unique identification of each staff, as well as the ability to dynamically assign MIDI channels for the staves of each page during playback can allow different pages to have different formats.



### 7.6.3 Developing a Music Font

The OOTMN requires a music font that is tailored to the requirements of the toolkit. Such a font should attempt to adhere to the NIFF standard regarding the reference positions of different music symbols. Positional idiosyncrasies that exist in the music font used for the OOTMN had to be dealt with on a trial and error basis as they were encountered. Use of a font where positional attributes, spacing characteristics and reference points are known will greatly simplify future development. a study of existing music fonts can provide valuable information for the design of a font.

### 7.6.4 Optimising Display Updates

Optimising the drawing of a page using a memory device context as described by Schildt [Schildt, 1995] is necessary to prevent unnecessary screen flickering during the drawing of a page. The screen device context is updated by copying the relevant area of the screen from a memory device context rather than redrawing the entire display area. To support such a scheme, the OOTMN must draw symbols to both a memory device context and a screen device context as shown in figure 7.1. Updating of the display is accomplished by means of Windows forcing the display to be repainted or the OOTMN explicitly invalidating a portion of the screen.

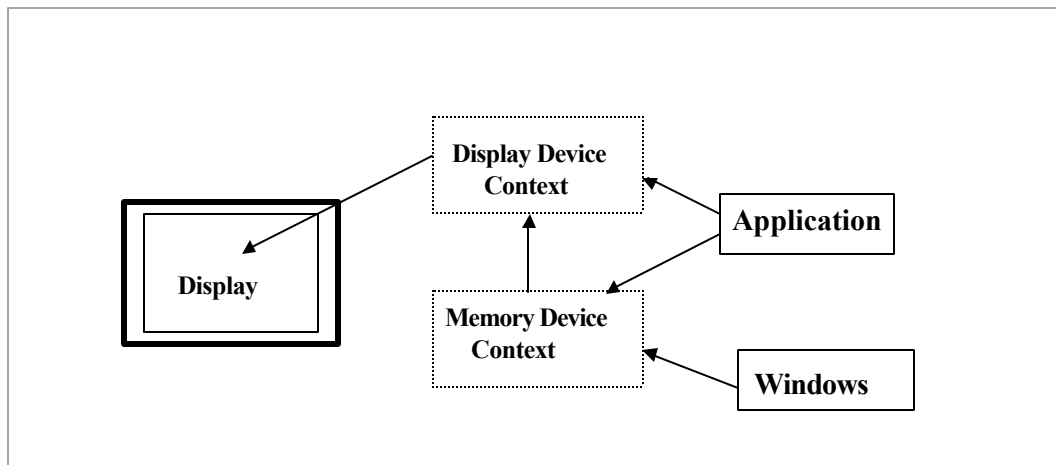


Figure 7.1 Display updates using screen and memory device contexts.

### 7.6.5 Porting the OOTMN

The negative impact of the Borland Object Windows environment on the development of the toolkit was mentioned in chapter five. The fact that the MMT is not able to support 32 bit Windows development under Borland C++ was also discussed. Considering these factors, it is not viable to proceed with further development using Borland C++ and Object Windows. The entire toolkit should be ported to Microsoft Visual C++ using the Microsoft Foundation Classes. These classes provide an object-oriented encapsulation of the windows API in much the same way as the Object Windows Library and are directly supported by the thirty-two bit Maximum MIDI Toolkit. Porting the toolkit to a UNIX environment under XWindows would also be an interesting possibility.

## 7.7 Summary of the Functionality of the OOTMN

The following is a summary of the features of the toolkit for music notation :

- i) Supported symbols are notes, rests, accidentals, dynamics, hairpins, slurs, ties, clefs (treble, bass, alto, tenor) and articulation symbols.
- ii) MIDI playback of scores.
- iii) Automatic chord creation. Notes can be dragged out of a chord and stems added to individual notes to form split-stem chords.
- iv) Automatic beam creation. Beams can be adjusted vertically and horizontally. Beams can be adjusted to represent extended and fan beams. Cross-staff beaming with limited MIDI playback is also supported.
- v) Grouplets of any duration can be represented and are correctly performed via MIDI.
- vi) Intelligent playback of incomplete bars.
- vii) Automatic hit-testing of all symbols.
- viii) Automatic updating of context-sensitive relationships between symbols.
- ix) Symbols can be placed anywhere on a page.
- x) Symbols may be treated as context-sensitive, purely graphic symbols or, a mixture of both types of symbols.

## Appendix A

# The OOTMN User Guide

## 1. Introduction

The toolkit consists of a single library file (toolkit.lib) that is linked into an application.

The toolkit may be used in three different modes, each mode behaves differently and is suited to different applications.

### 1.1 Basic Mode

Requires the least coding effort but has the following restrictions :

- 1.1.1 Score and Page Objects are not supported, systems and staves form the highest level of the object hierarchy.
- 1.1.2 Only single-line staves or a grand (piano) staff are supported. A staff or system that uses playback may only be one line long. As many staves or systems as needed may be created in a window but they will not provide continuous MIDI playback; playback will stop at the end of a line.
- 1.1.3 No user interaction is supported. i.e. the methods used have a very limited context-sensitivity. For example, a note cannot be dropped onto a staff, it must be explicitly assigned to a staff-step from the code.

### 1.2. Default Mode

Default mode does not place any restrictions on the use of the toolkit, it does however make certain assumptions when creating objects :

- 1.2.1 Staves are created with the standard five lines.
- 1.2.2 The time-signature 4 | 4 is used for all staves.
- 1.2.3 The key-signature of C major / a minor (i.e. no sharps or flats) is used.
- 1.2.4 Treble clefs are used for all staves.

### 1.3. Template Mode

Provides the most flexibility but requires the most coding. Template mode creates a musical score from a template object that must be created before creating the score. Using a template facilitates individual key-signatures, time -signatures, and clefs for each staff, as well as the ability to specify grouping of staves and the breaking of barlines within a system.

## 2. The Score Object and Global Data

A Score object MUST be created when using the toolkit in either simple, default or template mode.

The score object contains important data that is used globally within the toolkit. To make the Score object visible to all source files the variable **ScorePtr** should point to the Score object and be visible to the entire application. The constructor need not be invoked with all parameters, as some of the parameters have default values which are indicated in the documentation.

## 3. Windows and Pages

In order to maintain type compatibility with the internal declarations used by the toolkit, a window that is passed to a page constructor should be of type TWindow. If an application uses a window object derived from TWindow or TFrameWindow, the window pointer should be cast to the base class TWindow before being used as a parameter to the page object's constructor.

## Appendix B

# The Object-Oriented Toolkit Library Reference

## Basic Toolkit Reference

### Class

#### Score

---

Class Score holds global information about the musical score.

#### Constructor

**Score (TWindow\* ptrPageWindow, int KeySign, int TimeSign,  
int NumStaves, int NumBars);**

Creates a Score Object with the specified number of staves, number of bars, time-signature and time signature.

#### Parameters :

**PageWindow\* ptrPageWindow**

Window in which the score is to be displayed. Applications should derive class PageWindow from the Borland base class TWindow.

**int KeySign**

Specifies the number of sharps or flats : 1 to 7 sharps, or 8 to 14 flats.

**int TimeSign**

Specifies the Time signature as a single two digit number. Common time is 44, three minims is 32.

**int NumStaves**

Specifies the number of staves. The score will by default have a single system consisting of the specified number of stave. Range of values : 1..2

**int NumBars**

Determines the number of bars for each staff. Range of values : 1..n

Destructor

**~Score();**

Member functions**BOOL OpenMIDIOut (int MIDIport);**

Opens the MIDI output port specified by MIDIport.

MIDIport may range from 0 (MIDIMapper) to the total number of available MIDI devices.

**void Play ();**

Plays the score using a previously opened MIDI port.

**void Format();**

Adjusts the spacing of all notes and rests in the score.

**SetVoice ( int StaffNumber, int VoiceNumber);**

Sets the currently active voice on the staff specified by StaffNumber to the voice specified by VoiceNumber. VoiceNumber values may be 1 or 2.

Voice number one has upward stems, while voice number two has downward stems.

**InsertNote (Note\* ptrNote, int StaffNumber, int BarNumber, int VoiceNumber);**

Inserts a note belonging to voice VoiceNumber, in the bar specified by BarNumber.

BarNumber is a bar which occurs on the staff specified by StaffNumber. The note will be placed after any other notes or rests.

**InsertRest (Rest\* ptrRest, int StaffNumber, int BarNumber, int VoiceNumber);**

Inserts a rest belonging to voice VoiceNumber, in the bar specified by BarNumber

BarNumber is a bar which occurs on the staff specified by StaffNumber. The rest will be placed after any other notes or rests.

**DeleteMusicEvent (int BarNumber, int VoiceNumber, int EventNumber);**

Deletes a note or rest belonging to voice VoiceNumber, in the bar specified by BarNumber. EventNumber is the number of the event starting from the left-hand side of the specified bar.

**BeamNotes (int StaffNum, int BarNum, int VoiceNum,  
int StartingNote, int EndingNote);**

Beams a group of notes belonging to the specified staff, bar, and voice. The first note is specified by StartingNote, and the second note by EndingNote.

## **Class Note**

---

Class note is derived from class MusicEvent which is derived from the base class MusicSymbol. Note objects automatically construct their stems and flags as required.

### Constructor

**Note (int Duration, int StaffStep, int Accidental);**

Constructs a note specifying its duration, pitch and optional accidental.

Parameters :

**int Duration** Possible values : 1 ..7 where

1 = whole note, 4 = crotchet and 7 = demi-semi-quaver.

**int StaffStep**

The lowest line of the staff is staff step 0, extending upwards using positive integers, and downwards using negative integers.

**int Accidental**

Possible values : 0 = no accidental, 1 = #, 2 = x, 3 = b, 4 = bb.

### Destructor

**~Note();**

### Member Functions

## **Class**

## **Rest**

---

### Constructor

**Rest (int Duration, int StaffStep);**

See class Note for possible values.

### Destructor

**~Rest();**

## Default / Template Toolkit Reference

### Global Data

#### **ScorePtr**

Points to the score object.

#### **Score class** \_\_\_\_\_

A single Score object holds global data essential to the correct functioning of the toolkit. A pointer to the score object must be visible to the entire application. This object should be created before any other toolkit objects, as many toolkit objects require both data and functions provided by the Score object.

### Constructor

```
Score (TWindow* ptrWindow, int SystemsPerPage, int StavesPerSystem,
      int BarsPerStaff, int StaffSize = 2, int StaffLines = 5,
      int KeySign = 0, int TimeSign = 44, int Tempo = 80);
```

Parameters :

#### **TWindow\* ptrWindow**

ptrWindow specifies the main window of the application.

#### **int SystemsPerPage**

The number of systems contained by each page.

#### **int StavesPerSystem**

The number of staves forming each system.

#### **int StaffSize**

The size of the staff Possible values 1 . . 3. Default = 2.

#### **int StaffLines**

Number of lines for each Staff. Possible values 1 . . 5. Default = 5.

#### **int KeySign**

Global key-signature of the score. Possible values : 0 = no key-signature.

1 . . 7 sharps, 8 . .14 (i.e. 1 . . 7) flats. Default = 0.



**int TimeSign**

Specifies the Time signature as a single two digit number. Common time is 44, three quavers is 38.

**int Tempo**

Specifies tempo in beats per minute. Possible values : 1..max.

Destructor**~Score()**Public member functions**void AddPage (Page\* ptrPage);**

Adds a new page to the score.

**void DeletePage (int PageNumber);**

Deletes the page at PageNumber from the score.

**int GetCurrentPageNo ();**

Returns the number of the currently active page..

**void SetCurentPageNo (int PageNumber);**

Sets the current page to the page specified by page number.

**Page\* GetPage();**

Returns a pointer to the currently active page.

**LoadFont (LPSTR FontName );**

Loads the Windows TrueType font specified by FontName returning a handle to the font. Returns NULL if the font cannot be loaded.

**void Draw (TDC& dc, int Flag);**

Draws the currently active score page using the device context specified by dc.

Flag indicates whether hitspot rectangles should be drawn for line symbol objects.

Possible values : 0 = do not draw hitspots, 1 = draw hitspots.

**int GetCurrentVoice ();**

Returns the number of the currently active voice. Possible values 1 . . 4.

**void SetCurre ntVoice (int VoiceNumber);**

Sets the currently active voice to the voice specified by VoiceNumber.

**char\* GetStaffName (int StaffNumber) ;**

Returns the name of this staff..

**int GetGroupStemDir (NoteArray\* ptrNoteArray);**

Returns the 'average' stem direction for the Note objects in ptrNoteArray.

Returned values : 1 = up stem, 0 = down stem.

**NoteArray\* SetGroupStems (NoteArray\* ptrNoteArray, int direction);**

Sets stems of all note objects in ptrNoteArray to the specified direction.

Values for direction : 1 = up stem, 0 = down stem. Returns a NoteArray where all notes have stems set to the specified direction.

**void Transpose (NoteArray\*, int Transposition);**

Transposes the notes in note array by the number of semitones specified by transposition.

Values : positive values transpose upwards, negative values downwards.

**Page\* FindPage (int BarNumber);**

Returns pointer to the page that contains the bar specified by BarNumber.

BarNumber is the global bar number from the start of the score.

**int GetSystPerPage();**

Returns the global default number of systems per page.

**int GetStavesPerSyst ();**

Returns the global default number of staves per system.

**int GetBarsPerStaff ();**

Returns the global default number of bars per staff.

**void SetMIDIvelocity (int DynamicID, int MIDIvelocity);**

Sets the MIDI velocity specified by value for the dynamic symbol specified by dynamic.

DynamicID values : ppp = 1, pp = 2, p = 3, mp = 4, mf = 5, f = 6, ff = 7, fff = 8.

MIDIvelocity : 0 . . 127

**int GetMIDIvelocity (int DynamicID) ;**

Returns the MIDI velocity associated with the dynamic DynamicID.

See SetMIDIvelocity for DynamicID values.

Public Data Members**BOOL AlignBarlines;**

Determines whether barlines are drawn for each staff or across the entire system.

Possible values : TRUE draws barlines across system, FALSE draws barlines for each staff.

**BOOL AutoSpace**

Determines whether notes and rests must be automatically spaced within bars

Possible values : TRUE = auto space on, FALSE = auto space off.

**MusicSymbol\* ptrMusicSymbol**

Hold the currently selected MusicSymbol object. NOTE : this value must be set before attempting operations on a selected object.

**Page class** \_\_\_\_\_

Class Page represents a single page of a score that is displayed in a window. All symbols belonging to a page with the exception of MusicEvent objects should be added to, or deleted from a page using the AddSymbol and DeleteSymbol functions described below.

Constructor**Page (TWindow\* ptrPageWindow, int PageNumber, char\* PageTitle = "");**

Constructs a new page displayed in the window specified by ptrPageWindow having the specified page number and title.

Parameters :

**TWindow\* ptrPageWindow**

The window in which to display the page. This window should be derived from the Borland classes TFrameWindow or TWindow.

Destructor**~Page();**

Public Member Functions**MusicSymbol\* FindSymbol (TPoint& Point);**

Searches for a MusicSymbol object at screen position point. Returns NULL if no symbol exists at that position, or a pointer to the symbol if it is found.

**Bar\* FindBar (TPoint& Point);**

Returns a pointer to the bar that contains the position specified by Point.

**Bar\* GetNextBar (Bar\* ptrBar);**

Returns a pointer to bar following the bar specified by ptrBar.

**System\* GetSystem (int SystemNumber);**

Returns a pointer to the system on this page having the number specified by system number.

**MusicEventArray\* GetMusicEvents (TRect\* Rectangle);**

Returns a pointer to an array of music events contained within the rectangle Rectangle.

**NoteArray\* GetNotes (TRect\* Rectangle);**

Returns a pointer to an array of notes contained within the rectangle Rectangle.

**inline int GetPageNumber() ;**

Returns the PageNumber;

**inline int GetTopMargin();**

Returns the top margin of the page.

**inline int GetBottomMargin();**

Returns the bottom margin of the page.

**inline int GetLeftMargin();**

Returns the left margin of the page.

**inline int GetRightMargin();**

Returns the right margin of the page.

**void AddSymbol (MusicSymbol\* ptrMusicSymbol);**

Add the symbol ptrMusicSymbol to the page.

MusicSymbol can be any object derived from class MusicSymbol.

**void DeleteSymbol (MusicSymbol\* ptrMusicSymbol);**

Deletes the symbol ptrMusicSymbol from the page.

## **MusicSymbol Class**

---

Class MusicSymbol is the base class from which all objects that are music symbols are derived. Note that MusicSymbol is an abstract base class and is invoked by the constructors of classes derived from it.

### Constructor

**MusicSymbol (TPoint& point);**

Constructs a MusicSymbol object at the screen position given by point.

### Destructor

**~MusicSymbol();**

### Public member functions

**virtual void UpdatePosition (TPoint& point);**

Sets the position of MusicSymbol to the screen position specified by point.

**virtual void DrawBoundsRect();**

Draws a rectangle around the music symbol.

**inline TPoint& GetPosition();**

Returns a TPoint object representing the symbol's position.

**inline void SetPosition (TPoint& point);**

Sets the position of the symbol to the position specified by point.

**void SetPosx (int xpos);**

Sets the x co-ordinate of the symbol to the position specified by point.

**void SetPosy (int ypos);**

Sets the y co-ordinate of the symbol to the position specified by point.

**int GetDrawState();**

Returns the visibility state of the MusicSymbol object. 0 = hidden, 1 = visible.

**void SetDrawState (int State);**

Sets the visibility state of the MusicSymbol object. 0 = hidden, 1 = visible.

**inline int GetDrawState();**

Returns the current visibility state of a music symbol, 0 = hidden, 1 = visible.

**inline void SetDrawState (int State);**

Sets the visibility of a symbol, State may be : 0 = hidden, 1 = visible.

## **System class**

---

Class System represents a logical grouping of staves on a page. Systems are numbered from 1 (starting at the top of a page) to the number of systems on a page.

### Constructor

**System (int PageNumber, TPoint& point, int SystemNumber);**

### Parameters :

#### **int PageNumber**

Number of the page containing this system.

#### **TPoint& point**

point is the starting position of the system, system number represents

#### **int SystemNumber**

Number of this system on the page (from the top downwards) starting from 1.

### Destructor

**~System();**

Public Member Functions

**void UpdatePosition (int StaffNumber, TSize& offset);**

Updates the positions of all system staves lower than and including the staff specified by StaffNumber by the x and y offsets specified by offset.cx and offset.cy.

**Staff\* GetStaff (int StaffNumber);**

Returns the staff on this system having a number StaffNumber.

**inline int GetSystemNumber();**

Returns the SystemNumber of this system.

**void SetTime (int Top, int Bottom, int Split1, int Split2);**

Sets the time signature for all bars within this system. Top and Bottom define the top and bottom values of the time-signature. Split1 and Split2 define two values to be used as the top values in a composite time-signature. (Thus  $Top = Split1 + Split2$ ). For a time-signature that is not a composite time-signature these values should be set to zero.

**void SetTime (Bar\* ptrBar, int Top, int Bottom, int Split1, int Split2);**

Sets the time signature for all bars sharing the same vertical position on this system as the bar given by ptrBar. Top and Bottom define the top and bottom values of the time-signature.

Split1 and Split2 define two values to be used as the top values in a composite time-signature. (Thus  $Top = Split1 + Split2$ ). For a time-signature that is not a composite time-signature these values should be set to zero.

## **Connector class**

---

### **Base class : MusicSymbol**

Class connector represents symbols such as brackets and curly braces that are used to group staves.

#### Constructor

**Connector (Staff\* ptrFirstStaff, Staff\* ptrSecondStaff, int Symb) ;**

Creates a connector object that connects the two staves specified by ptrFirstStaff and ptrSecondStaff. Symbol specifies the graphic symbol used as a connector.

Possible values for Symbol : 0 = Bracket, 1 = Brace.

#### Destructor

**~Connector();**

#### Public Member Functions :

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## Staff class

---

**Base class : MusicSymbol**

### Constructor

**Staff (System\* ptrParent, TPoint& StaffPosition, int StaffNumber, int StaffSize = 3,  
int NumLines = 5, char\* StaffName = "");**

### Parameters :

**System\* ptrParent**

Points to the system that contains this staff.

**TPoint& StaffPosition**

The screen position of this staff.

**int StaffNumber**

The number of this staff within it's parent system.

**int StaffSize**

The size of this staff. Possible values : 1 . . 3 (largest). Default = 3.

**int NumLines**

The number of lines of this staff. Possible values 1 . . 5. Default = 5.

**char\* StaffName**

The name of this staff. Default = "".

### Destructor

**~Staff();**

### Public Member Functions

**int GetStaffStepPos (int Ypos);**

Returns the nearest y-axis position (that is a valid staff step) to the y co-ordinate Ypos.

Each line or space on the staff is represented by an integer staff step. Step 0 is the lowest line of the staff. Staff steps extend upwards using positive integers and downwards using negative integers.

**int GetYPos (int StaffStep);**

Returns the y co-ordinate of the staff step StaffStep. See above for an explanation of staff step.

**int GetStaffStep (int ypos);**

Returns the staff step at the y co-ordinate ypos. See above for an explanation of staff step.



**void SetClef (int ClefType);**

Sets the clef of this staff to ClefType. See the clef constructor for possible values.

**inline int GetBottomy();**

Returns the y co-ordinate of the bottom line of this staff.

**Bar\* FindBar (TPoint& point);**

Returns the bar that contains the screen position point. Returns NULL if a bar is not found.

**Bar\* GetBar (int BarNumber);**

Returns the bar on this staff having the bar number specified by BarNumber.

**inline Clef\* GetClef();**

Returns a pointer to this staff's clef.

**void SetClef (Clef\* ptrClef);**

Sets the clef of this staff to the clef specified by ptrClef. If the staff contains notes they are automatically moved to retain their original pitches.

**void DrawLegerLines (TPoint& point);**

Draws leger lines from point to the top or bottom of the staff depending on whether point lies above or below the staff.

**inline int GetStaffNumber();**

Returns this staff's number.

**inline int GetNumLines();**

Returns the number of lines of this staff.

**inline int GetStaffSize();**

Returns the size of this staff.

**inline System\* GetptrParentSystem();**

Returns a pointer to this staff's parent system.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **Bar class**

---

**Base Class : none**

Constructor :

**Bar (int BarNumber, Staff\* ptrParentStaff);**

Parameters :

**int BarNumber**

The number of this bar within the staff starting from 1 - the leftmost bar of the staff.

**Staff\* ptrParentStaff**

Specifies the staff which contains this bar.

Destructor

**~Bar();**

Public Member Functions

**Voice\* GetVoice (int VoiceNumber);**

Returns a pointer to the voice occurring in this bar having a voice number of VoiceNumber.

**inline int GetLxpos();**

Returns the left barline's x co-ordinate.

**inline int GetRxpos();**

Returns the left barline's x co-ordinate.

**Clef\* GetClef();**

Returns the clef at the start of the bar.

**Clef\* GetClef (int xpos);**

Returns the clef that is operative at the horizontal position in the bar given by xpos.

**void DrawBoundsRect();**

Draws this bar's bounding rectangle.

**void AddNote (int StaffStep, int Duration, int Direction, int Accidental);**

Adds a Note object of pitch StaffStep, a duration specified by Duration, a stem direction specified by Direction and an accidental specified by Accidental. The function creates the note and accidental objects. The note is added to the currently selected voice.

See `Score::GetCurrentVoice()`; NOTE : for non-trivial applications it is preferable to use

`Voice::AddEvent(MusicEvent*);`

**inline Bar\* GetBar (int BarNo);**

Returns the bar number of this bar.

**inline int GetBarNumber();**

Returns the bar number of this bar relative to its parent staff.

**inline int GetGlobalNumber();**

Returns the bar number of this bar from the start of the score.

## **Clef class**

---

Class clef represents musical clef symbols and is associated with Staff or Bar objects via a staff or bar ptrClef attribute.

**Base class : MusicSymbol**

### Constructor

**Clef (int ClefType, TPoint& Position);**

Constructs a clef object of type ClefType at the window point Position.

### Parameters :

**int ClefType**

Specifies the symbol for this clef object.

Possible values : 0 = Percussion, 1 = Treble, 2 = Bass, 3 = Tenor, 4 = Alto.

**TPoint& Position**

The screen position of the clef.

### Destructor :

**~Clef();**

### Public Member Functions :

**int GetClefType();**

Returns the type of clef. Possible values returned are :

0 = Percussion, 1 = Treble, 2 = Bass, 3 = Tenor, 4 = Alto.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **KeySign class**

---

**Base Class : MusicSymbol**

### Constructor

**KeySign (Staff\* ptrStaff, int Key, TPoint& point);**

**KeySign (Bar\* ptrBar, int Key, TPoint& point);**

### Parameters :

**Staff\* ptrStaff or, Bar\* ptrBar**

Parent object for this key-signature.

**int Key**

Specified the number of sharps or flats for this key-signature.

Possible values : 0 = no key-signature, 1 . . 7 sharps, 8 . . 14 (i.e. 1 . . 7) flats.

**TPoint& point**

Position of the key-signature.

### Destructor

**~KeySign();**

### Member Functions

**inline int GetKeyValue()**

Returns the number of sharps or flats of this key-signature

Possible values : 0 = no key-signature, 1 . . 7 sharps, 8 . . 14 (i.e. 1 . . 7) flats.

**void SetKeyValue (int key)**

Sets the number of sharps or flats of this key-signature specified by key.

Possible values : 0 = no key-signature, 1 . . 7 sharps, 8 . . 14 (i.e. 1 . . 7) flats.

**inline int GetSpacing();**

Returns an integer representing the spacing between accidentals used for this key-signature

**void SetSpacing (int spacing)**

Sets the spacing used by accidentals for this key-signature.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **TimeSign class**

---

### Constructor

**TimeSign (int meter, int value, TPoint& point);**

### Parameters :

**int meter**

The top integer of the time-signature.

**int value**

The bottom integer of the time-signature.

**TPoint& point**

The position of the time-signature in screen co-ordinates.

### Destructor

**~TimeSign();**

### Member Functions

**TimeSignMeter\* GetTSignMeter();**

Returns a pointer to the TSignMeter (top value) object of this key-signature.

**TimeSignValue\* GetTSignValue();**

Returns a pointer to the TSignValue (bottom value) object of this key-signature.

**void SetSplit (int FirstValue, int SecondValue);**

Converts this time-signature to a composite time-signature where the top value is displayed as : FirstValue + SecondValue.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState (int State);** See MusicSymbol.

### **TimeSignMeter class**

---

**Base class : MusicSymbol**

NOTE : The constructor and destructor are handled by the TimeSign object, they should not be explicitly invoked. See class TimeSign for obtaining a pointer to a TimeSignMeter object.

#### Member Functions :

**void SetMeter (int Meter);**

Sets the value (number of beats) for this TimeSignMeter object.

Possible values : Any integer.

**int GetMeter();**

Returns the metric value for this TimeSignMeter object. See **SetMeter**.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

### **TimeSignValue class**

---

**Base class : MusicSymbol**

Represents the lower value of a time-signature. NOTE : The constructor and destructor are handled by the TimeSign object, they should not be explicitly invoked. See class TimeSign for obtaining a pointer to a TimeSignMeter object.

Member Functions :**void SetDuration (int Duration);**

Sets the lower-value to the value specified by Duration.

Possible values : 6 =  5 =  4 =  3 = 

**int GetDuration();**

Returns the durational value of this time-signature. See **SetDuration** for possible values.

**void SetDisplayMode (int Mode);**

Sets the way in which the lower value is displayed for this time-signature.

Possible values : 0 = numeric, 1 = symbolic (i.e. a note)

**int GetDisplayMode();**

Returns the current display mode for this time-signature's lower value.

See **SetDisplayMode** for possible values.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

**Grouplet class** 

---



**Base Class : none**

Constructor :



**Grouplet (MusicEventArray\* ptrMusicEventArray, int FaceNumber, int FaceDuration,  
int ActualNumber, int ActualDuration);**

Parameters :



**int FaceNumber**

The notated number of durations in the grouplet. For a triplet of 3  in the time of 2   
the FaceNumber is three.



**int FaceDuration**

The notated durational value of the grouplet. For a triplet of 3  in the time of 2   
the FaceDuration is 5 (quaver).

**int ActualNumber**

The actual number of durations in the grouplet. For a triplet of 3  in the time of 2   
the ActualNumber is two.

**int ActualDuration**

The actual durational value of the grouplet. For a triplet of 3  in the time of 2   
the ActualDuration is 5 (quaver).

NOTE : no restrictions exist as to the range of the above values.

Destructor :

**~Grouplet();**

**GroupBracket class** 

---

**Base Class : LineSymbol : MusicSymbol**

Constructor :

**GroupBracket(Grouplet\* ptrParentGrouplet, TPoint& Position, int Symbol);**

Destructor :

**~GroupBracket();**

Member Functions

**TPoint CalcBracketPos();**

Returns the starting position of the bracket. Also calculates the points stored in class LineSymbol determining the position of the bracket.

**int GetBracketDirection();**

Returns the direction of the bracket. Possible values : 1 = up, 0 = down.



## MusicEvent Class

---

### Base Class : MusicSymbol

Class MusicEvent is a base class for note and rest objects.

#### Constructor

**MusicEvent (TPoint& point);**

#### Destructor

**MusicEvent();**

#### Member Functions

**WORD GetEventTime();**







Returns the MIDI duration of this MusicEvent.

**void DrawLegerLines();**

Draws leger lines for this MusicEvent. All leger lines required by the note or rest are drawn above or below the staff as required by the position of this MusicEvent object.

**inline void SetDuration (int duration);**

Sets the duration of this MusicEvent to the value specified by duration.

Possible values : 1 =  2 =  3 =  4 =  5 =  6 =  7 = thirty-

second.

**inline int GetDuration();**

Returns the duration of this MusicEvent. See SetDuration for possible values.

**int GetStaffStep();**

Returns the staff step of this MusicEvent.

Possible values : -n . . 0 (lowest line of the staff) . . n

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **Voice class**

---

### Constructor

**Voice (Bar\* ptrParentBar, int VoiceNumber);**

Constructs a voice object belonging to the bar specified by ptrParentBar having a voice number specified by VoiceNumber.

### Destructor

**~Voice();**

### Member Functions

**MusicSymbol\* GetMusicEvent (TPoint& point);**

Returns the MusicEvent at position point.

**void AddEvent (MusicEvent\* ptrMusicEvent);**

Adds the MusicEvent object specified by ptrMusicEvent to this voice.

NOTE : a Note object added at the same position as an existing Note object automatically creates a new CChord object.

**int GetXPosition();**

Returns the correct x position for automatically spacing events within a voice.

**BOOL IsFull();**

Returns TRUE if this voice contains notes and rests whose total duration equals that of the time-signature for the bar in which this voice occurs. Returns FALSE otherwise.

**void SetStemDirection (int Direction);**

Sets the default stem-direction for this voice. Possible values : 1 = up, 0 = down.

**int GetStemDirection() ;**

Returns the default stem direction for this voice. See **SetStemDirection** for possible values.

**inline int GetVoiceNumber();**

Returns the voice number of this voice.

**inline Voice\* GetVoice (int VoiceNumber);**

Returns the voice having a voice number of VoiceNumber.

**inline Bar\* GetptrParentBar() ;**

Returns the parent bar of this voice.

## **CChord class**

---

**Base class : MusicEvent : MusicSymbol**

### Constructor

**CChord (Note\* ptrFirstNote, Note\* ptrSecondNote);**

Creates a chord from the two notes ptrFirstNote and ptrSecondNote. A single stem is set for the entire chord, existing stems belonging to chord notes are deleted where necessary.

### Destructor

**~CChord();**

### Member Functions

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **Accidental**

### **class**

---

**Base class : MusicSymbol**

### Constructors

**Accidental (int AccidType, TPoint& position);**

Constructs an accidental at screen position Position. Possible values of AccidType :  
2 = double-sharp, 1 = sharp, 0 = natural, -1 = flat, -2 = double flat

**Accidental (Note\* ptrNote, int AccidType);**

Constructs an accidental object of AccidType associated with the note ptrNote.  
See above for AccidType values.

### Destructor

**~Accidental();**

Member Functions**inline int GetValue();**

Returns the value of this accidental. See above for possible values.

**inline void SetValue (int value);**

Sets the value of this accidental. See above for possible values.

**inline int GetOffset();**

Returns the offset of this accidental from it's associated note object.

**inline void SetOffset (int offset);**

Sets the offset of this accidental from its associated note object.

**inline TPoint& GetPosition();** See MusicSymbol.**inline void SetPosition (TPoint& point);** See MusicSymbol.**void SetPosx (int xpos);** See MusicSymbol.**void SetPosy (int ypos);** See MusicSymbol.**int GetDrawState();** See MusicSymbol.**inline void SetDrawState(int state);** See MusicSymbol.**Note class** 

---

Constructors**Note (Voice\* ptrVoice, int Duration, TPoint& Position);**

Constructs a Note object belonging to the voice specified by ptrVoice at screen position Position. See **MusicEvent::SetDuration** for possible durational values.

**[Note (Voice\* ptrVoice, int Duration, int Accidental, int Dot);**

Constructs a Note object belonging to the voice specified by ptrVoice at screen position Position. See **MusicEvent::SetDuration** for possible durational values.

See **Accidental::SetValue** for possible values.

Destructor**~Note();**Member Functions**int GetNameClass();**

Returns the name class of this note. Values : C = 0 . . B = 12

**int GetPitchClass();**

Returns the pitch class of this note. Values : C = 0 . . B = 11

**int GetOctave();**

Returns the octave of this note.

**inline int GetStaffStep();**

Returns the staff step of this note.

**int GetDefaultNoteHead (int NoteHead);**

Returns the symbol currently used as the notehead for this note.

**void SetNoteHead (int NoteHead);**

Sets the note head of this note to the symbol specified by NoteHead.

**Stem\* SetStem (Voice\* ptrVoice);**

Creates a stem object for this note

**virtual MidiMessage\* GetMidi ();**

Returns a MidiMessage object corresponding to this note.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

**WORD GetEventTime();** See MusicEvent.

**void DrawLegerLines();** See MusicEvent.

**inline void SetDuration (int duration);** See MusicEvent.

**inline int GetDuration();** See MusicEvent.

**int GetStaffStep();** See MusicEvent.

## **Rest class**

---

**Base class : MusicEvent : MusicSymbol**

Constructors

**Rest (Voice\* ptrVoice, int Duration, TPoint& Position, Grouplet\* ptrGrouplet);**

**Rest (Voice\* ptrVoice, int Duration, TPoint& Position);**

Destructor

**~Rest();**

Member Functions

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

**WORD GetEventTime();** See MusicEvent.

**void DrawLegerLines();** See MusicEvent.

**inline void SetDuration (int duration);** See MusicEvent.

**inline int GetDuration();** See MusicEvent.

**int GetStaffStep();** See MusicEvent.

## **Stem class**

---

### **Base class : LineSymbol : MusicSymbol**

NOTE : A Stem object need not be explicitly created. Creation of a Note object automatically creates an associated stem.

Member Functions

**void SetDirection (int StemDirection);**

Sets this stem's direction to the direction specified by StemDirection.

0 = down, 1 = up.

**inline int GetDirection();**

Returns the direction of this stem, 0 = down, 1 = up.

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **Flag class** \_\_\_\_\_

**Base class : MusicSymbol**

NOTE : A Stem object need not be explicitly created. Creation of a Note object automatically creates an associated stem.

Functionality for class flag is not currently implemented. All functions are performed by the stem class that is associated with a flag object.

## **MidiMessage class** \_\_\_\_\_

### Constructor

**MidiMessage (BYTE Status, BYTE KeyNum, BYTE Velocity, DWORD Time);**

### Parameters :

**BYTE Status**

**BYTE KeyNum**

**BYTE Velocity**

**DWORD Time**

### Destructor

**~MidiMessage();**

## **BeamedGroup class** \_\_\_\_\_

Base class : none

BeamedGroup objects encapsulate all the notes and beams of a group of notes that are beamed together.

Constructor**BeamedGroup (NoteArray\* ptrNoteArray);**

Creates a BeamedGroup consisting of the note objects contained in the array ptrNoteArray.

Destructor**~BeamedGroup();**Public Member Functions :**void DrawAllBeams (TDC& dc, int Flag);**

Draws all the beams of this beamed group using the device context specified by dc .

**void UpdateAllBeams (TSize& offset, int Flag);**

Updates the positions of all beams according to the value of flag.

Possible flag values : **1** = update left-hand side positions,  
**2** = update right-hand side positions,  
**3** = update left and right-hand side positions.

**void UpdatePosNote (Note\* ptrNote, TSize& Offset);**

UpdatePosNote should be called after moving a note that is part of a beamed group. This function ensures that the stem of a note that is moved is correctly adjusted within the beamed group. ptrNote represents the Note object that has been moved. Offset is the offset on the x and y axes given by offset.cx and offset.cy.

**Dynamic class**

---

**Base class : MusicSymbol**Constructor**Dynamic (int DynamicID, TPoint& point);**

Creates a dynamic of type DynamicID at the screen position point.

**Dynamic (int DynamicID, int yPos, MusicSymbol\* ptrParentMusicSymbol);**

Creates a dynamic of type DynamicID at the vertical screen position ypos.

The dynamic has an instance connection to the music symbol ptrParentMusicSymbol.

The dynamic's x screen co-ordinate is taken from the x co-ordinate of music symbol ptrParentMusicSymbol.



Destructor

**~Dynamic();**

Member Functions

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

**Tie class** 

---

**Base class : LineSymbol : MusicSymbol**

Constructor

**Tie (Note\* ptrNote1, Note\* ptrNote2);**

Member Functions

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosx (int xpos);** See MusicSymbol.

**void SetPosy (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

Destructor

**~Tie();**

**Hairpin class** 

---

**Base class : LineSymbol : MusicSymbol**

Constructor

**Hairpin (TPoint& Position);**

Constructs a hairpin object at position point.

**Hairpin (MusicEvent\* ptrFirstEvent, MusicEvent\* ptrSecondEvent);**

Constructs a hairpin object with instance connections to ptrFirstEvent and ptrSecondEvent.

#### Destructor

**~Hairpin();**

#### Member Functions

**inline TPoint& GetPosition();** See MusicSymbol.

**inline void SetPosition (TPoint& point);** See MusicSymbol.

**void SetPosX (int xpos);** See MusicSymbol.

**void SetPosY (int ypos);** See MusicSymbol.

**int GetDrawState();** See MusicSymbol.

**inline void SetDrawState(int state);** See MusicSymbol.

## **Barline class** \_\_\_\_\_

**Base class : LineSymbol : MusicSymbol**

#### Constructor

**Barline(System\* ptrSystem, TPoint& point, int LowerYpos, int Symbol);**

Constructs a barline within the system ptrSystem, starting from the upper position point and extending to a lower position given by the x co-ordinate of point and y co-ordinate lowerYpos. The barline is drawn according to the symbol given by Symbol. Possible values for symbol : 1 = normal, 2 = thin double-line, 3 = thin / thick line

#### Destructor

**~Barline();**

#### Member Functions

**void SetXPos(int xpos);**

Sets the x co-ordinate of this barline to the value specified by xpos.

**inline int GetBottomY();**

Returns the y co-ordinate of the bottom of the barline.

**inline void SetBottomY(int ypos);**

Sets the y co-ordinate of the bottom of the barline to the value given by ypos.

**inline void SetSymbol(int Symbol) ;**

Sets the barline symbol to the symbol specified by Symbol. See the constructor for possible values.

**inline int GetSymbol();**

Returns the integer representing the symbol used to draw this barline.

**inline void SetSide (int BarSide);**

Sets the side of the bar that this barline belongs to.

Possible values for BarSide : 1 = left-hand side, 0 = right-hand side.

**inline int GetSide() ;**

Returns the side of this barline. See SetSide for possible values.

**Part class**

---

Base class : none

Part objects represent a part extracted from a score.

Constructor

**Part (Staff\* ptrStaff, int VoiceNumber);**

Constructs a part objects consisting of the contents of the voice specified by VoiceNumber occurring on the staff specified by ptrStaff .

**Part (Staff\* ptrStaff1, Staff\* ptrStaff2);****Part (Staff\* ptrStaff1, Staff\* ptrStaff2, Staff\* ptrStaff3);**

Constructs a part object consisting of the contents of the specified staves .

Destructor

**~Part();**

## Glossary

**Abstract base class** A base class that serves as a template from which other classes can be derived. An abstract class cannot be instantiated (i.e. constructed) as it is only a high-level abstraction of a class of objects.

**Accidental** Sign immediately preceding a note that modifies the pitch of a note. A sharp or double-sharp raises the pitch, a flat or double-flat lowers the pitch. A natural cancels the effect of a sharp or flat.

**Aggregate (Whole -part) relationship** Relationships between objects where an object is defined as being 'a part of' another object.

**Articulation Symbol** Notation symbols used to indicate articulation (the manner in which a note is performed.)

**Attribute** A data member encapsulated within an object. Typically a characteristic of the object.

**Auxiliary Symbol** Music notation symbols that are instrument specific.

**Bar** A temporal unit occurring across a system. Bar lines indicate the starting and ending position positions of a bar. The total durational content of a bar is given by the time-signature.

**Barline** Vertical lines drawn across a system that indicate the starting and ending positions of bars.

**Base class** A class forming the highest-level of an object hierarchy from which other more specialised classes are derived.

**Beam** A thick horizontal or sloping line used to group notes having the duration of a quaver or less. A single beam replaces the flags of the beamed notes.

**Beamed group** Set of notes that are visually joined together by means of one or more beams.

**Beat** (Meter) A metric unit within a bar.

**Cardinality** (between objects) The number of objects that exist within a relationship with another object.

**Cent** Measurement of pitch used in physics. Twelve-hundred cents = one octave.

**Changing meter** Application of one of multiple time-signatures to a bar depending on the metrical grouping found within the bar.

**Chord** A group of notes that are performed simultaneously.

**Class** Collection of objects that have a similar description and thus characteristics in common.

**Clef** Symbol associated with a staff that indicates the actual pitches designated by the staff.

**Context-sensitive relationship** Relationship between two entities where modification of one entity implies a modification of the second entity.

**Cross-staff beam** A beam that connects notes notated on more than one staff.

**Dynamic** Alphabetic symbol or symbols denoting loudness. Usually notated below the staff they influence all succeeding notes. f (forte) = loud, p (piano) = soft. Often used together with m (mezzo) = medium.

**Ending bar** A bar that is the last bar of a repeating section of music and is only performed once during either the first or second performance of the passage.

**Enharmonic** A note that has a different syntax to another note but shares the same pitch.

**Extended beam** A beam that is extended to start or end before or after the first or last stem of notes forming a beamed group.

**Flag** Symbol attached to the end of the stem of a note that indicates duration.

**Generalisation-Specialisation relationship** Relationship where one class is a specialisation of another. The specialised class is termed a **derived** class, the general class is termed a **base** class.

**Grouplet** Generic term that refers to any irregular group. A group of notes that divides the meter by an amount that results in durations that cannot be notated by the available durational symbols for notes and rests.

**Group bracket** Symbol (either a slur or a rectangular bracket) that visually groups the notes forming a grouplet.

**Hairpin** Symbol consisting of two sloping lines that converge at one end used to indicate a diminuendo or crescendo.

**Harmony** The study of the construction of chords, as well as the relationships between chords.

**Instance relationship** A relationship where one object is aware of the existence of one or more instances of a different object. Usually a semantic connection exists between the two objects.

**Key-signature** Sharps or flats notated at the beginning of a staff or bar. Usually indicates the key (i.e. tonal centre) of the music.

**Leger line** Lines that extend the staff above or below the usual five lines. Used only for the notation of specific notes that are notated above or below the staff.

**Line-symbol** Class of OOTMN objects that are symbols that are constructed from one or more line segments.

**Method** A member function encapsulated within an object. Usually used to perform operations on the object's attributes or to communicate with other objects.

**Music symbol** Class of OOTMN objects that are graphic symbols used to notate music.

**NIFF (Notation In terchange File Format)** File representation designed specifically for music notation based on the Microsoft RIFF format.

**Notational Convenience** Term developed for the OOTMN denoting a music symbol that has a purely visual, graphic function.

**Ornament** Group of notes conforming to a well-defined pattern that may be notated using an abbreviation represented by a symbol.

**(Operator) Overloading** Assigning a user-defined meaning that differs from the traditional meaning associated with an operator symbols found in a programming language.

**Part** The portion of a score that is intended to be performed on a single instrument.

**Piano-roll notation** Graphic representation of MIDI notes that represents pitch on the y-axis and duration on the x-axis.

**Polymorphism** The ability of a base class to dynamically select an appropriate derived-class method during program execution.

**Real-time System** A software system that must immediately respond to system inputs.

**Rest** Music notation symbol used to denote silence. The length of the silence depends on the particular symbol used.

**Secondary bar** A virtual bar that exists as an alias for a bar. Invented for the OOTMN, only the alias can be performed allowing the MIDI realisation to differ from the written notes.

**Semantic Constituent** A symbol that is required for the semantic interpretation of another symbol.

**Semantic modifier** A symbol that modifies the semantic interpretation of another symbol.

**Sequencer** Software that facilitates the recording, manipulation and playback of MIDI (and possibly audio) data.

**Slur** Symbol that connects two notes that do not have the same pitch. The first note is to be performed for as long as possible - until the start of the second note.

**Split-meter time-signature** A time-signature having two values that represent the meter. Equivalent to the alternation of two separate meters.

**Split-stem chord** A chord having one or more additional stems that allow the notation of chord notes occupying the same staff-step.

**Staff** Horizontal lines used to denote musical pitch. Notes and rests are notated on or between the lines of a staff which traditionally consists of five lines.

**Stem** A vertical line attached to the notehead of a note.

**System** A group of staves that are visually grouped together. All notes and rests occupying the same vertical time-slice are performed at the same time.

**System line / System bracket** The vertical line and bracket or brace that visually connect staves at their starting positions on the left of the page.

**Tempo** The frequency at which metric units occur. Measured in beats per minute.

**Tie** Symbol in the form of an arc that connects two notes of the same pitch that are performed as a single duration representing the sum of the durations of the two notes.

**Timbre** Tone quality of a sound determined by the harmonic spectrum of the sound. MIDI timbres are provided by synthesiser patches.

**Time-signature** Two integer values notated at the beginning of a staff or bar. The top value indicates meter, the bottom value specifies the durational value of each metric unit.

**Tonality** The pitch (key) centre of a musical work or passage.

**Unified Modelling Language (UML)** An object-oriented design notation that attempts to provide a standard notation for object diagrams.

**Voice** A single melodic thread within multiple melodic threads that are notated on a single staff.

## Bibliography

Borland International, Reference Manuals for Borland C++ ver. 4.0 and ver. 5.0

Cantù, M. and Tendon, S. Borland C++ 4.0 Object-Oriented Programming.  
Random House, New York, 1994.

Cilwa, P. Borland C++ Insider : The Guide To Hard-To-Find and Undocumented Features.  
John Wiley, New York, 1994.

Coplien, J.O. Advanced C++ : Programming Styles and Idioms.  
AT&T Bell Telephone Laboratories, 1992.

Johnsonbaugh, R. and Kalin, M. Object-Oriented Programming in C++.  
Prentice Hall, New Jersey, 1995.

Kelley, A. and Pohl, I. C by Dissection : The Essentials of C Programming.  
Benjamin / Cummings, Redwood City, 2nd ed., 1992.

Shammas, N.C. What Every Borland C++4 Programmer Should Know.  
Prentice-Hall,

Stroustrup, B. The C++ Programming Language.  
AT&T Bell Telephone Laboratories, 2nd ed., 1991.

Thayer, R.H. and Dorfman, M. System and Software Requirements Engineering.  
IEEE Computer Society Press, Los Almitos, 1990.



## References

- Anderton, C. MIDI For Musicians.  
Amsco Publications, New York, 1986.
- Arnott, P. Personal electronic mail concerning embedded windows in help files  
pamnote@sky.net, 1996.
- Assayag, G. and Timis, D. A Toolbox for Music Notation.  
Proceeding of the International Computer Music Conference, Montreal, 1986.
- Booch, G. Object Oriented Design with Applications.  
Benjamin/Cummings, Redwood City, 1991.
- Brinkman, A. Pascal Programming for Music Research.  
University of Chicago Press, Chicago, 1990.
- Carmichael, A. Object Development Methods.  
SIGS Books Incorporated, New York, 1994.
- Cannam, C., Green, A. Rosegarden Notation Editor and MIDI Sequencer  
<http://www.maths.bath.ac.uk/~masjpf/rose.html>, 1997
- Cessna, M. Personal electronic mail concerning embedded windows in help files  
mcessna@wordmission.org, 1996.
- Clements, P.J. A System for the Complete Enharmonic Encoding of Musical Pitches and Intervals.  
Proceedings of the International Computer Music Conference,  
Computer Music Association, 1986.
- Coad, P. Object Models - Strategies, Patterns, and Applications.  
Yourdon Press, 2nd ed, 1997.
- Coad, P. and Yourdon, E. Object-Oriented Analysis.  
System Software Requirements and Engineering. ed. Thayer, R.H., Dorfman, M.  
IEEE Computer Society Press, Los Alamitos, 1990.
- Coad, P. and Yourdon, E. Object-Oriented Analysis.  
Yourdon Press, New Jersey, 2nd Ed. 1991.
- Coad, P. and Yourdon, E. Object-Oriented Design.  
Yourdon Press, New Jersey, 1992.

- Cole, H. *Sounds and Signs : Aspects of Musical Notation.*  
Oxford University Press, 1974.
- Computer Music Journal - Review of Rosegarden Notation and Sequencer Software.  
CMJ vol. 21(2), MIT Press, 1997.
- Dyer, L.M. *MUSE : An Integrated Software Environment for Computer Music Applications.*  
Proceedings of the International Computer Music Conference,  
Computer Music Association, 1986.
- Formula Software Ltd. *The Formula Graphics Multimedia Authoring System.*  
[www.FormulaGraphics.com](http://www.FormulaGraphics.com)
- Forte, A. *The Structure of Atonal Music.*  
Yale University Press, New Haven, 1973.
- Giampolo, D. 'C' source code to compute Bézier curves.  
[dbg@sgi.com](mailto:dbg@sgi.com), 1996.
- Glassner, A.S. *Graphics Gems.*  
Academic Press, Cambridge, 1990.
- Grande, C. *Notation Interchange File Format Specification version 6a.*  
<http://mistral.ere.umontral.ca:80/~belkina/NIFF.doc.html>, 1995.
- Huron, D. *Design Principles in Computer-Based Music Representation.*  
in Marsden, A. , Pople, A. eds. *Computer Representations and Models in Music.*  
Academic Press, London, 1992.
- InterSystem Concepts. *The Everest authoring System.*  
[www.insystem.com](http://www.insystem.com), 1998.
- Lande, T.S. and Vollsnes, A.O. *Object Oriented Music Analysis.*  
*Computers and the Humanities*, vol. 28, 1995.
- Lehrman, P.D. and Tully, T. *MIDI for the Professional.*  
Amsco Publications, New York, 1993.
- McGowan, A. *Encore 3.0 Installation Guide.*  
Passport Designs Inc, Half Moon Bay, California, 1993.
- MediaChance Software. *The MultimediaBuilder Authoring System.*  
[www.mediachance.com](http://www.mediachance.com), 1988.
- MIDI 1.0 Specification.  
International MIDI Association, California, 1983.

- Messick, P. Maximum MIDI : Music Applications in C++.  
Manning Publications, 1997.
- Messick, P. Posting regarding use of the MMT with the 32 bit Borland Compiler.  
MMT user's forum, [www.maxmidi.com](http://www.maxmidi.com)
- Music Quest Inc. and Messick, P.A. MIDI Programmer's ToolKit for Windows.  
Music Quest Inc., Plano, 1994.
- Richards, T. 'Memwatch' Memory Monitor.  
OWL Archives [www.ioc.ee/owl](http://www.ioc.ee/owl), 1985.
- Petzold, C. Programming Windows 3.1.  
Microsoft Press, 1983.
- Pierce, J.R. The Science of Musical Sound.  
Scientific American Library, 1983.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W.  
Object-Oriented Modelling and Design.  
Prentice-Hall, 1991.
- Schildt, H. Windows 95 Programming : Nuts and Bolts for Experienced Programmers.  
McGraw-Hill, Berkeley, California, 1995.
- Sibelius-Software  
[www.sibelius-software.com](http://www.sibelius-software.com), 1998.
- Stroustrup, B. A Better C ?  
Byte - the small systems Journal, August 1988,
- Salzer, F. Structural Hearing.  
Dover, 1962.
- Straus, J.N. Introduction to Post-Tonal Theory.  
Prentice-Hall, London, 1990.
- Sun Microsystems. Java Development Kit ver. 1.1  
Sun Microsystems, 1996.
- Taylor, W.F. The Geometry of Computer Graphics.  
Wadsworth and Brooks, Belmont, 1992.
- Technoligor Software  
[www.igortech.pi.se](http://www.igortech.pi.se), 1998.

Walsen, M. Description of the Notation Engine.  
www.notation.com, 1998.

Walsen, M. Personal electronic mail concerning features of the Notation Engine.  
markwa@notation.com, 1998.

Ward, P., Mellor, S. Structured Development for Real-Time Systems.  
Prentice-Hall, New Jersey, 1985.

Wentworth, P. Discussion regarding Java native code.  
pw@cs.ru.ca.za, 1997.

# Index

## A

Abstract class, 78-9, 83  
 Accidental, 6, 15-16, 18, 23, 39-41, 94  
 Accidental object, 61, 95, 153  
 Acoustics, 10  
 Aggregation, 62, 82, 85-6  
 Algorithm, 97, 103-4, 106-7, 130  
 Anderton, C., 3  
 Apple Macintosh, 1  
 Articulation, 66  
 Articulation symbol, 19, 23  
 ASCII characters, 35  
 Attribute, 40, 163  
 Auxiliary symbol, 20, 165

## C

C programming language, 96  
 Canon, 602  
 Clef, 10-11, 51, 56  
 Clef object, 57  
 Cent, 10  
 Changing meter, 12  
 Chord, 17-18  
 Chord (split-stem) 18  
 CChord object, 62-4  
 Coad, P., 38-9  
 Container classes, 127  
 Context-sensitive relationship, 22, 116-17  
 COPYIST, 29-30  
 Cross-staff beam, 107

## B

Bar, 12-14, 50-522  
 Bar (Secondary), 55  
 Bar object, 48-9, 55  
 Barline, 12-14, 50-55, 84  
 Barline object, 48-9, 84-6, 164  
 Beam, 16, 18-9, 23, 28, 98, 110  
 Beam (extended), 19, 110  
 Beam (fan), 19, 110  
 Beam (cross-staff), 107  
 Beam object, 65-6  
 Beam symbol, 93  
 Beamed group 64, 103-10  
 BeamedGroup object, 63  
 Beaming algorithm, 103  
 Berio, L., 54  
 Bernstein, L., 12  
 Bézier curve, 97  
 Binomial pitch representation, 95  
 Bitmap, 99  
 Booch, G., 37-8  
 Borland C++, 27, 75, 81, 94, 127  
 Brace, 99  
 Bracket, 99  
 Brahms, J., 66  
 Brinkman, A., 93

## D

DARMS code, 33, 93  
 Dynamic symbols, 20  
 Durations, 16  
 Dyer, L.M., 93

**E**

Eales, A.A., 29  
 Embedded notation example, 121-122  
 Encoding music symbols, 93-4  
 ENCORE, 27-28, 109  
 EVEREST authoring software, 119

**G**

Generalisation-specialisation, 40, 77-78  
 Giampaolo, 97  
 Grouplet, 65, 100  
 Grouplet object, 68, 153  
 GroupBracket object, 68

**I**

IGOR, 30  
 Inheritance, 78-79, 84, 92  
 Initialism, 36  
 Instance relationship, 40-41, 49, 56, 61, 68,  
 82-4, 95, 98  
 IrregularGroup (see grouplet)

**K**

Key-Signature, 5-6, 10-11, 28, 56  
 Key-Signature object, 150

**M**

Macintosh (Apple), 1  
 NIFF, 33, 52, 103  
 Maximum MIDI toolkit (MMT), 94-95  
 Mellor, S., 37  
 Notation program, 105, 107  
 Messick, P., 94  
 Method, 40  
 Microsoft Foundation Classes, 94  
 MIDI, 2-3, 31, 94-96  
 MIDI implementation, 99-103  
 MIDI playback, 99, 106-7, 117  
 MIDI recording 117  
 MIDI resolution, 98  
 MIDI synchronisation, 97-99  
 MidiEvent structure 96-97  
 MidiMessage object, 66, 76, 96, 98-99  
 Model, 77

**F**

FINALE, 25-27  
 Flag, 59  
 Flag object, 60  
 Font (Music), 34-35, 131  
 FORMULA graphics, 119

**H**

Hairpin, 96  
 Hairpin class, 163  
 Harmony tutor, 117-121  
 Hierarchy (notation), 6  
 Hitspot, 111-12  
 HTML, 118

**J**

Jacobson, I., 37  
 Java, 118

**L**

Leger line, 10, 121, 147  
 Lehrman, T., 3  
 LineSymbol class, 96  
 Lutoslawski, W., 46, 54

**N**

Name class, 93  
 NOTATION ENGINE, 34  
 Notation program - commands, 112-14  
 Notation program - MIDI usage, 109  
 Notational convenience, 23  
 Note, 15-16  
 Note object, 60-61, 63-4, 68, 71,  
 Note name, 94  
 Note representation, 93-5  
 Note Structure, 59  
 Notehead, 59

Moving objects, 89-92  
 Mozart, W.A., 13  
 MULTIMEDIA BUILDER, 119-20  
 MUSICA, 29, 78-80  
 MusicEvent object, 62-3, 155  
 Music font, 34-5, 138  
 Music Notation, 1-2  
 MusicSymbol object, 83-4, 143  
 MUSIKUS, 36

## O

OOTMN code example, 122  
 OOTMN (functionality of), 132  
 OOTMN (limitations of), 126  
 Object, 40  
 Object model, 73-76, 82-3  
 Object notation, 40  
 Object-Oriented analysis, 39, 77  
 Object-Oriented design, 1, 39  
 Object-Oriented development methods, 37  
 Object-Oriented music representation, 36  
 Object-Oriented programming, 1  
 Object-Oriented representation of music, 36  
 Object relationships, 70-72  
 Object storage, 85  
 Object Windows library (OWL), 81  
 Octave, 10, 90  
 Operator overloading, 100  
 Ornaments, 20

## R

Reciprocal duration code, 33  
 Rest, 16-17  
 Rest object, 61  
 RoseGarden, 30  
 Rumbaugh, J., 37

## P

Pachelbel, J., 14  
 Page, 7  
 Page object, 42-3  
 Palette (notation application), 116  
 Part, 21-22  
 Part object, 69, 165  
 Phrase symbol, 92-93  
 Piano roll notation, 31  
 Pitch-Class, 94  
 Pictorial mnemonic, 36  
 Pitch, 6, 9, 16  
 Postscript font, 35  
 Prototyping, 128

## S

Salzer, F., 5  
 Schenkerian notation, 5  
 Score, 7  
 Score object, 42-3  
 ScoreTemplate object, 43-44  
 Secondary bar, 55-6, 114  
 Secondary stem, 110  
 Semantic constituent, 22-3  
 Semantic modifier, 23  
 Scope (of symbols), 5  
 Score, 7  
 Searching, 86-88  
 SIBELIUS, 30  
 Slur object, 61  
 Slur symbol, 97  
 Software Design, 2  
 Software Engineering, 128  
 Software tools, 33  
 Spatial dependency, 24

**T**

Tie object, 60  
 Tie symbol, 97  
 Time-signature, 11-12, 56  
 Time-signature (split), 111  
 Time-signature (symbolic), 111  
 TimeSign object, 57, 151  
 TimeSignMeter object, 57, 152  
 TimeSignValue object, 57, 152  
 True-Type font, 35  
 Toolbar (Notation application), 115

**V**

Visibility (of symbols) 5  
 Visual C++, 81  
 Voice, 14-15, 103  
 Voice object, 58

**X**

Xwindows, 132

SplitTimeSignMeter object, 56  
 Split-stem chord, 18, 114  
 Stroustrup, B., 1  
 Staff, 7-11, 45-6  
 Staff object, 47  
 Staff step, 88-90  
 Stem, 59  
 Stem object, 60  
 Stravinsky, I., 53  
 Symbolic time-signature, 107  
 System, 7-9, 45-6  
 System object, 47  
 SystemBracket object, 47

**U**

Unified Modelling Language (UML), 37  
 UNIX, 30, 132  
 User interface design, 107

**W**

Ward, P., 38  
 Whole-part structure, 40, 77  
 Windows API, 75

**Y**

Yourdon, E., 38-9