Proceedings of the NACCQ 2000
Wellington NZ www.naccq.ac.nz

The Music Notation Toolkit: A Study in Object-Oriented Development

Andrew Eales

Central Institute of Technology
Upper Hutt
New Zealand
andrew.eales@cit.ac.nz

ABSTRACT

The Music Notation Toolkit (MNT) is an object-oriented C++ library for music notation that simplifies the development of music software in a Microsoft Windows environment. Semantic relationships between different music symbols are accurately represented by the toolkit, allowing the creation of a complete musical score. User interaction with a musical score is directly supported by the MNT.

This paper examines the object-oriented design of the toolkit, as well as the implementation of the toolkit in the C++ programming language. Design and implementation decisions are discussed in relationship to accepted object-oriented design and implementation practices.

1. INTRODUCTION

The Music Notation Toolkit (MNT) simplifies the development of applications that use traditional staff notation to represent music. The library consists of thirty-eight classes that represent structural and semantic relationships within a musical score. A wide range of functionality including searching, adding or removing context-sensitive relationships and hit testing is provided by the toolkit. In addition, formatting of symbols at local and global levels is also possible.

Implemented in the C++ programming language using Borland C++ and the Borland Object Windows Library (OWL), the toolkit supports application development within a Windows environment. Music notation symbols are represented by a true-type music font, bitmapped graphics and Windows GDI (Graphics Device Interface) graphics primitives provided by the Windows API. MIDI input and playback is implemented using the Maximum MIDI Toolkit (Messick, 1997). The development environment is shown in figure 1.

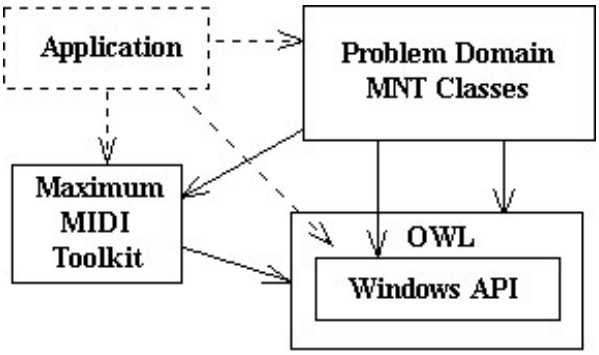

Figure 1. The Music Notation Toolkit Development Environment.

The MNT was created to find solutions to problems encountered while using commercial music notation programs. Steep learning curves, which may be partly attributed to poor user-interface designs, as well as

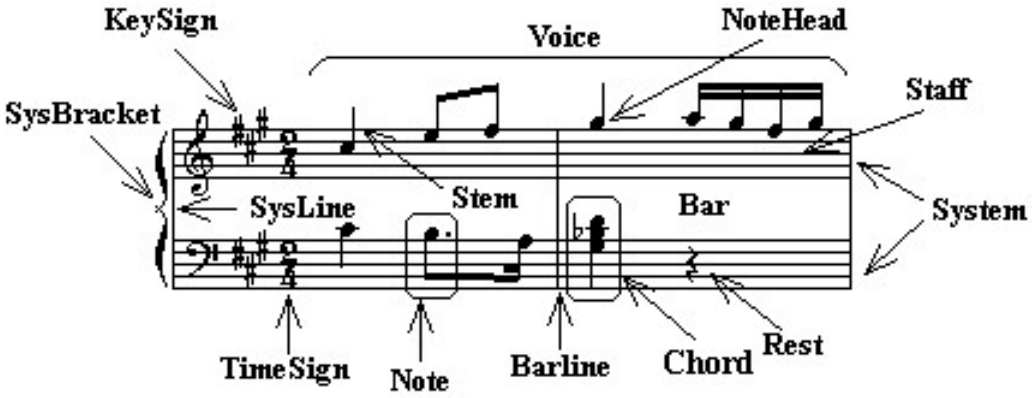

Figure 2. Music Notation Symbols

functional inflexibility characterize many commercially available systems. It was intuitively felt that a well-designed OO representation of music notation could represent traditional staff notation as well as allowing modern extensions to traditional practices to be accommodated.

1.1 Music Notation

Music notation is a highly context-sensitive representational system that consists of a hierarchical arrangement of different symbols. The semantic interpretation of individual symbols usually occurs within the context of other symbols.

Figure 2 provides an example of various music notation symbols.

While many symbols can be directly mapped to abstract data types, abstractions representing concepts from music theory that indirectly relate to music notation are required for a complete model of the problem domain. For example, the upper melody of the notation example shown in figure 2 forms a voice which is an independent melodic line similar to a software thread. A voice is not a notational symbol, but rather a concept that is required to represent separate melodic strands. The set of symbols used in music notation does not provide the semantic means to create a meaningful model of the subject matter. Discussion of the music notation problems addressed by the toolkit, and the strategies used to solve these problems are beyond the scope of this paper which focuses on the OO design of the toolkit.

2. OO ANALYSIS AND DESIGN

2.1 Selecting a Development Methodology

Development of a library required an approach that emphasized data abstraction, as development did not focus on any specific application. Previous experience included the use of the development method proposed by Coad and Yourdon (1991) and some familiarity with the methods developed by Booch (1994) and Rumbaugh (1991).

The use case driven approach to OO software development first advocated by Jacobson, (1992) and adopted by Rational Software Corporation's CASE tools appeared inappropriate in the absence of a specific application. Furthermore, the adoption of use cases remains a controversial issue within the OO community. Meyer (Meyer, 1998 p 738), discusses the pitfalls of relying on use cases as a driving force in object-oriented software development. Korson (Korson, 1998) dismisses use cases as an excuse to practice functional decomposition within an OO framework. Identification of a set of use cases does not guarantee the discovery of a useful set of abstract data types that exist at an appropriate level of abstraction. Use cases also imply a sequential ordering of functionality that should be avoided in OO development.

Peter Coad (1997) building on his earlier work (Coad and Yourdon, 1992) advocates a development method guided by heuristics which he refers to as strategies, combined with a study of design patterns. This approach does not emphasize functional requirements or the identification of abstract data types as a point of departure. Design procedure attempts to take a holistic view of the system and oscillates between class design and the identification of functional requirements referred to as system features.

Dynamic interactions between objects are modeled using scenario diagrams that correspond to UML sequence diagrams. This allows the designer to freely adopt a spiral analysis, design and implementation process that encourages the development of both logical and intuitive insights into the workings of the problem domain.

2.2 Design of the Music Notation Toolkit

The functional requirements for an OO library that is application independent must consider the set of possible operations that can be associated with different abstract data types. This set of operations subsumes the set of possible use cases, which will be implemented from the operations defined on the identified abstract data types. Such an approach to design is interesting in that a bottom-up process is used where high-level application functionality emerges from combinations of the different class methods. Put another way, operations that naturally occur within the problem domain provide applications with the means to implement higher-level application specific functionality.

Figure 3 shows a simplified UML version of a portion of the object model derived from analysis of the problem domain, which was the result of more than a year of part-time development. The model is interesting in that it consists almost entirely of aggregate and instance relationships, having only a single generalisation-specialisation relationship. Most of the instance relationships model context-sensitive relationships between different music symbols.

The designer, who has experience lecturing music theory at undergraduate and graduate levels assumed the role of domain expert. Despite the background of the designer, the model underwent approximately nine

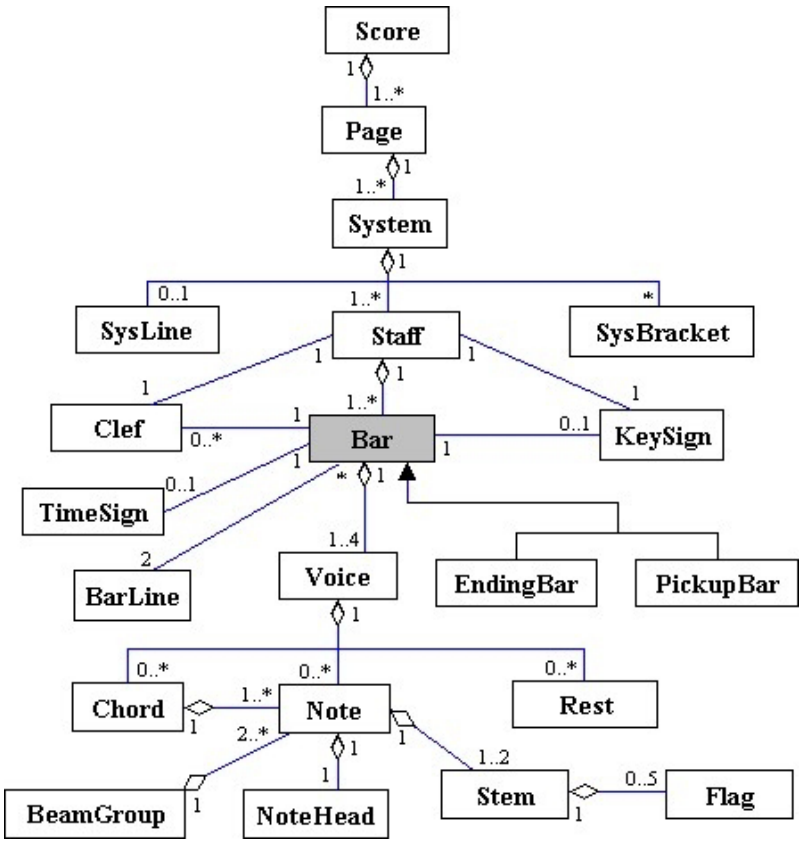

Figure 3 Object Model of the Problem Domain

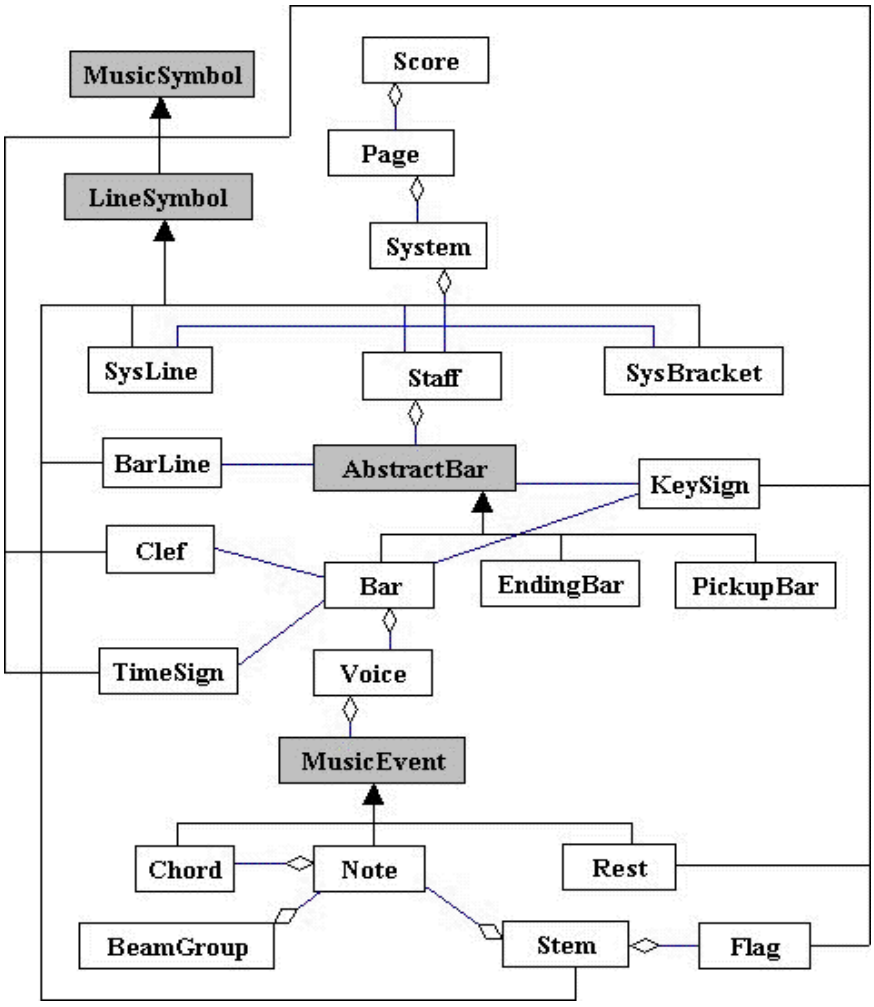

Figure 4 Object Model showing extensions to the Problem Domain.

revisions. Two factors necessitated these revisions. Firstly, the realization that a design decision was flawed and had to be revised and secondly, as a result of discovering implicit domain knowledge. The discovery of implicit conventions within the subject domain formed an interesting experience in that the designer found it extremely difficult to unambiguously codify conventions used on a daily basis.

3. AN OO IMPLEMENTATION IN C++

Attempting to implement an OO design that is devoid of inheritance relationships is absurd. Inheritance relationships drive OO programs by providing required functionality via polymorphism. Object technology as currently implemented by programming languages provides no specific means to implement aggregate and instance relationships. These relationships are implemented either as nested declarations or, as references or pointers to objects. Aggregate and instance relationships on their own offer little advantage over structured development methods.

Object models must first strive to accurately model the problem domain. A modeling process that attempts to enforce inheritance relationships by identifying abstractions that are not readily obvious during early stages of the process may be misled by inappropriate abstractions. An OO design for the music notation application MusicA (Lishik, 1994) developed at the Haifa Institute of Technology focused on discovering high-level abstractions. Abstract data types such as MelodyPart which is a musical abstraction resulting from a combination of notational elements were identified. Preconceptions concerning how the system should be designed to maximize the use of programming language features resulted in abstract data types that did not directly form a part of the subject matter.

3.1 Extending the Problem Domain

Extensions to the problem domain provide the object model with two root classes, class Score forming a non-base class root, while class MusicSymbol provides a root class that is an abstract base class. Class MusicSymbol represents all classes that have direct symbolic counterparts, class LineSymbol (derived from MusicSymbol) consists of continuous symbols having more than one anchor point associated with them. These additions are illustrated in figure4:

The classes MusicSymbol and LineSymbol extend the problem domain and at the same time provide a level of abstraction that is suitable for implementation in a

programming language. In addition to providing polymorphic behavior an abstract base class that exists as the root of an OO class hierarchy provides a common type among derived classes. This base type allows a search procedure that has a search space consisting of different types to return a common type.

Class MusicEvent is an abstract base class representing notational elements that can be translated into sound. This class allows polymorphic behavior to drive the MIDI playback process. Inheritance relationships imply that a base class and all classes derived from it contain common features. Meyers (1996) recommends that these common features be utilised to create a higher-level abstraction implemented as an abstract base class. He suggests that all non-leaf base classes within an object hierarchy are potential abstract base classes. Class Bar is a non-leaf base class, which becomes a dereived class through the addition of an abstract base class AbstractBar. This transformation is shown in Figure 5 and Figure 6.

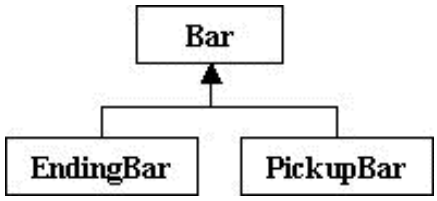

Figure 5: Concrete inheritance

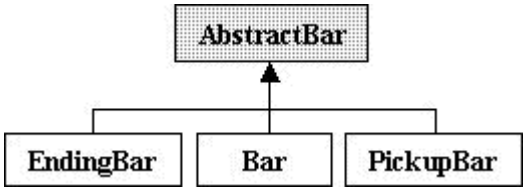

Figure 6. Abstract Inheritance

This transformation provides many benefits to the implementation. Polymorphic behavior is encouraged and the system is easy to extend. In addition, less pleasant features of OO programming languages such as partial assignment between derived classes are avoided.

Addition of class MusicSymbol influences the Note class in an interesting way, causing one of the classes forming a part of a Note to be redundant. Class NoteHead shown in figure 3 becomes redundant in figure 4 as the process of moving data members and methods to class MusicSymbol allows the remaining data to be moved to Class Note.

4. TESTING THE MUSIC NOTATION TOOLKIT

Testing a library can be achieved in two ways. By systematic testing of different methods and their dependent methods using scenarios or, by taking a more interesting approach and building an application that uses the library. Two applications, a notation editor and an educational application that teaches elementary harmony (an area of music theory) were built to test the capabilities of the MNT.

By compiling the MNT into a static library which was linked into the applications a rigorous separation of the user-interface and the problem domain was achieved.

4.1 User-Interface Design

User-Interfaces designed for modern GUI's quickly become unmanageable as global variables are added to keep track of the events occurring within the user interface. The state of a user-interface refers to information such as the current menu selection, the previously selected symbol or last mouse position. A solution to this problem is to create an abstract data type that stores the user-interface state. Thus, the state of an object (represented by the current values of its data members) mirrors the state of the user-interface. This approach provides two benefits. Data representing the state of a user interface is protected by being encapsulated inside a class and state transitions are formally controlled by class member functions.

5. CONCLUSIONS

The Music Notation Toolkit has allowed applications to successfully implemented music notation and suggested solutions to a wide variety of problems encountered in commercially available notation programs. The following insights gleaned from the design of the MNT can be valuable to OO developers and educators teaching OO analysis and design:

1. OO designers should resist implementing inheritance relationships for the run-time benefits of polymorphism. Enforced inheritance may create preconceptions regarding the abstract data types that exist within the problem domain.
2. Good OO designs attempt to discover abstractions within the problem domain that provide for an elegant implementation in a programming language. This process should only occur after a subject-centered model has been created.

3. The discovery of abstractions at higher levels may make classes at lower levels redundant as the higher-level classes abstract data and functionality away from the lower level classes.
4. Abstractions that are indirectly related to the problem domain are to be avoided in the selection of classes. Such abstractions can be difficult to identify. They can be meta-abstractions of an abstract data type within the problem domain or higher-level abstractions formed from a set of abstractions within the problem domain.
5. Intelligent behavior i.e. high-level functionality should not be distributed among classes within the problem domain. Functionality that is not the responsibility of a class has no claim to existence within the class. For example, a function that formats a page of music notation should use methods provided by the different classes. The function itself should not be spread across different classes.
6. Designers should not rely on use cases to drive an object-modeling process. Identification of elegant abstractions that exist at an appropriate level will enable any required functionality to naturally flow across the system.
7. Designers should develop a thorough understanding of the subject matter forming the problem domain bearing in mind that implicit domain knowledge is difficult to identify and codify.
8. Inheritance relationships should be examined to determine whether the implied commonality determining the relationship can be used to construct an abstract base class.
9. No known automated process guarantees a successful design. It is unlikely that a complete process will ever be discovered.
10. The process of implementing a design exposes weaknesses in a design. An active involvement in all aspects of development develops the necessary intellectual and intuitive skills required to evaluate and improve designs.

6. FUTURE DIRECTIONS

The MNT is currently being ported from a development environment that used the Borland Object Windows Library to Borland C++ Builder. Code that is dependent on the Borland container class library is being rewritten using the C++ Standard Template Library. Standard C++ code that is minimally dependent on the Windows API will simplify porting the MNT to other platforms.

General enhancements that would be valuable to the toolkit include the parsing of MIDI input in real-time, precise formatting of the layout according to engraving rules and support for different file formats. Correctly parsing MIDI input data into music notation in real-time is a non-trivial task. Currently available commercial notation packages process real-time input with varying degrees of success. An investigation of the conventions used by music engravers would provide insights into ways of improving the formatting capabilities of the MNT. Support for the Notation Interchange File Format (NIFF) which provides a standard file format for music notation based on the Microsoft Resource Interchange File Format (RIFF) would allow application that use the toolkit to export NIFF data to other applications.

Direct support for platform and application independent user-interfaces can be developed by considering the Model-View-Controller paradigm originally associated with Smalltalk development environments. A MVC architecture forms one of the object-oriented design patterns documented by Gamma (1995) and Vlissides (1998). Applied to the Music Notation Toolkit, the controller component would handle user input and notify the model consisting of the MNT class hierarchy should any alterations to the model occur. The model responds to changes by updating the view, which consists of the graphic representation of music notation. This arrangement would allow different views of a musical score to be presented to the user. A view of the MIDI data or a graphical controller window representing tempo changes can provide different views of the underlying object model. Future computer music systems will provide different representations of a musical work that will allow the user to control all notational and sonic aspects of a musical composition.

7. ACKNOWLEDGEMENTS

I would like to thank Rhys Owen of the Department of Electronics and Software Engineering, Central Institute of Technology for pointing out that my user-interface design used aspects of the Model-View-Controller pattern.

8. REFERENCES

Booch, G. (1994) Object-Oriented Analysis and Design with Applications (2nd ed) Menlo Park : Benjamin/ Cummings

Coad, P. (1997) Object Models - Strategies, Patterns and Applications. New Jersey : Yourdon Press.

Coad, P. and Yourdon, E. (1991) Object-oriented Analysis (2nd ed) New Jersey : Prentice-Hall.

Eales, A.A. (1999) An Object-Oriented Toolkit for Music Notation. M.Sc. Thesis, Rhodes University, Grahamstown.

Gamma, E. , Helm, R. , Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software.Reading : Addison-Wesley

Jacobson, I., (1994) Object-Oriented Software Engineering: A Use Case Driven Approach. Reading : Addison-Wesley.

Korson, T. (1998) The Misuse of Use Cases. Object Magazine 8(3).

Lishik,D. and Ben-Har,D. (1994) MUSICA3 and MUSICA2 source code and documentation. Haifa Institute of Technology - Dept. of Electrical Engineering.

Messick, P. (1997) Maximum Midi: Music Applications in C++. Greenwich : Manning Publications.

Meyer, B. (1997) Object-Oriented Software Construction, 2nd ed. New Jersey : Prentice Hall

Meyers, S. (1996) More Effective C++ - 35 New Ways to Improve Your Programs and Designs. Reading : Addison-Wesley.

Rumbaugh, J., Blaha M., Premerlani W. Eddy F. and Lorensin, W. (1991) Object-Oriented Modelling and Design, New Jersey : Prentice-Hall.

Vlissides, J. (1998) Pattern Hatching: Design Patterns Applied. Reading : Addison-Wesley