Table of Contents

CLOSe to the edge?
Advanced object-oriented techniques in the representation of musical knowledge[*]

Peter Desain & Henkjan Honing

NICI
Nijmegen University
P.O. Box 9104
NL-6500 HE Nijmegen

Alfa informatica, ILLC
University of Amsterdam
Spuistraat 134
NL-1012 VB Amsterdam

Abstract

The modeling of knowledge about musical expression asks for quite some flexibility during the design process and the availability of high-level abstractions to represent the complex concepts and their interactions in this domain successfully. One would expect, because of the enthusiastic claims made in the literature on object-oriented programming, that such an approach would be ideal for this task. This paper describes some aspects of CLOS, an modern object-oriented language that indeed provides some advanced construct that proved useful in the design and maintenance of a complex system for the manipulation of expression in music. However, some of the mechanisms should be used with care to stay far from the edge beyond which programs become too complex to grasp.

Keywords

Object-oriented programming, knowledge representation, AI and music, expression in music

Background

Since the appearance of the first experimental versions, object-oriented languages have found widespread application in computer music and music representation systems (see Pope, 1991). They have matured and grown into standardized systems (for instance, Smalltalk, C++ and CLOS), and seem to support sophisticated mechanisms for abstraction. But still it is questionable whether these constructs proved all that is necessary for dealing with difficult problems of knowledge representation.

> *"It is certain that programming languages of the distant future will not resemble those in use today. And almost certainly, the people who use abstractions in those*

---

[*] A preliminary report on this work was presented at the IAKTA/LIST International Workshop on Knowledge Technology in the Arts, Osaka, Japan and published as Desain & Honing (1993).

> *languages will have difficulty categorizing as abstractions the mechanisms we use today, so primitive and crude will these mechanisms seem."* (Gabriel & Steele, 1990)

In this paper we will discuss some abstraction mechanisms of a present object-oriented programming language (CLOS, an extension of Common Lisp) and try to indicate the strengths and shortcomings of some the available facilities with respect to their use in the representation of musical knowledge. We will investigate in how far some mechanisms must be used with care to stay far from the edge beyond which programs become too complex to grasp.

The actual modeling of a particular aspect of musical knowledge asks for quite some flexibility during the design process (for example, ease of changing the type relations and modifying the control structure). We need facilities that allow for an explorative construction of modular microworlds that concentrate on one aspect of the musical knowledge, and for a smooth combination of these microworlds into a complete system (Honing, 1993). One would expect, because of the enthusiastic claims made in the literature on object-oriented programming, that such an approach is ideal for the task. But although some modern mechanisms of these languages support this flexibility and contribute to the flexibility and expressive power, some problems remain.

We will describe these issues using an example of an object-oriented system in the music domain: a calculus for expression in music performance. First, the basic problems in the study and modeling of this domain will be presented.

Expression in Music

The expressive aspects of music are those measurable quantities in a performance that cannot be deduced from the information in a musical score - it is the performer's interpretation of that raw material. The term comprises subtle variations in tempo (the rubato that is most obvious in the slowing down at the end of a phrase), the use of timing (the way a note can be accented by delaying its onset by a small amount of time), small accents in loudness, asynchronies in the onsets of the notes that form a chord (e.g. by making the melodic voice stand out more clearly by playing it slightly ahead of the accompaniment). We will not use the term here in the sense that music is played with expression to induce an affect in the listener, an emotion or feeling. These feelings are considered to be communicated somehow by the purely syntactic notion of expression, which is much easier to open up to scientific investigation.

In most of the research on music cognition, this syntactic notion of expression is defined as the deviations of a performance with respect to the score or a mechanical performance. This numerical material, e.g. tempo profiles, can be analyzed and compared over different performances (see, e.g., Clarke, 1988). In most current music software expressive timing is defined as well as the note-to-note tempo or timing deviation from the score, either in the form of a separate stream of tempo information (as used, for instance, in commercial software for controlling synthesizers), or as a set of interpretation algorithms acting on a score (see, for example, Anderson & Kuivila, 1991). However, from a perceptual point of view, there is something awkward about this definition since a listener can perceive and appreciate expression in a performance without knowing the score.

Imagine you are listening to a radio program playing some new music for piano. You have never heard the piece before. Suppose you have very little experience in music making yourself, and do not know any music theory. Maybe to your own surprise, you are capable of deriving some of the musical structure. You are able to distinguish how many voices are played at the same time and where the musical phrases end. You can hear an error in the performance and distinguish it from deviations that were made on purpose. You can identify what is expressive timing (rubato, swing, phrasing) and what is rhythmical structure (the note values that might be notated in a score). You may be able to judge whether it is an amateur or professional pianist, and some might even be able to recognize the pianist by his/her way of playing. Clearly, a lot is communicated, only by means of the performance attributes of each piano note: its time of onset, its time of offset, and its loudness. And it is all communicated effectively, without the listener needing a score to pick-up the information.

In our work we therefore looked for an alternative description of the notion of expression, based on performance information and a structural description of the music performed. We define *expression within a unit as the deviations of its parts with respect to the norm set by the unit itself* (Desain & Honing, 1991). Using this intrinsic definition, expression can be extracted from the performance data, taking more global measurements as a reference for local ones, ignoring a possible score. An example might make this more clear. Lets take, for instance, a metrical hierarchy of bars and beats; the expressive tempo within a bar can be defined as the pattern of deviations from the global bar tempo generated by the tempo of each beat. Or, take the loudness of the individual notes of a chord; the dynamic expression within a chord can be defined as the set of deviations from the chord's mean loudness by the individual notes. Thus the structural description of the piece of music becomes central; it establishes the units which will act as a reference and determines its sub-units that will act as atomic parts whose internal detail will be ignored.

Next to its use in the study of expression itself, an important motivation for this work is the practical applicability of it in systems for computer music. Especially the music editors and sequencer programs that are commercially available nowadays are in need of better ways to treat musical information in musical ways. It is illustrative that structural and expressive notions from the everyday vocabulary of composers and musicians (like phrasing, ornamentation, agogic accents) have no corresponding constructs in current software, except maybe in notation programs that can handle them as annotations of the score. Expressive timing should not be considered a nasty feature of performed music, as it is in current multi-track recording techniques where tempo, timing and synchronization are treated as technical problems. Given a richer and more structured (knowledge) representations of music than the ones in use today (e.g., IMA, 1983), expression can again be regarded as an integral quality of performed music.

Expresso

The Expresso system -a calculus for expressive timing-, will serve as a concrete example of an object-oriented system in the music domain. Expresso is a representational system to describe music performances not only statically, but also in a transformable way. In this system it is possible for instance to modify the phrasing or articulation of a represented performance in musical and perceptually plausible ways. Basic distinctions in musical knowledge and perceptual processing are represented by a prototypical set of types of temporal structure and of expression, and their behavior under transformation; a specific transformation on a representation of a performance should be close to a real performance that underwent a similar transformation. For example, a transformation that changes the overall tempo should not just scale all note durations, but it should reflect what a human performer would do when given the instruction to play at a higher tempo, i.e. adapt the depth of rubato a different levels, leave some ornaments invariant, adjust the articulation. An I/O diagram of Expresso is shown in Figure 1. The input consists of a musical performance in MIDI format, a stream of note-on and note-off messages, and one or more structural descriptions (quantized durations are also needed as input, but they can be derived directly from the performance). The output is a modified performance given a certain transformation. A more complete description of the system is given in Honing (1992) and Desain & Honing (1991) which also includes a full code listing.

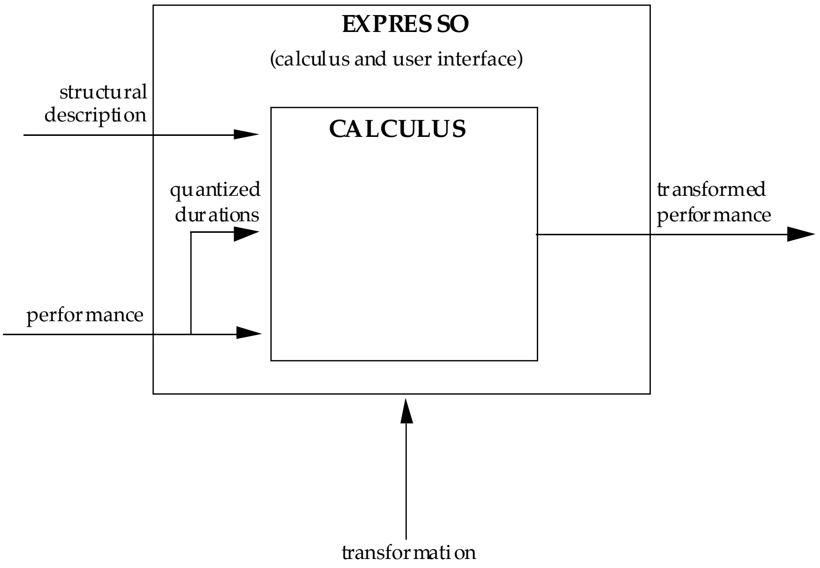

Figure 1. Input/output diagram of Expresso.

The underlying data representation makes a distinction between basic musical objects and structured musical objects. *Basic* musical objects are events that ‘carry’ expression (for instance, notes), with attributes that should be directly measurable from the recorded performance data, like a note's onset or loudness. *Structured* musical objects assign a particular kind of temporal structure to a group or collection of musical objects.

This set of structural concepts mirrors some basic distinctions in the perception of temporal structure, for instance, between successive temporal processes that deal with events occurring one after another, and simultaneous temporal processes, that handle events occurring around the same time. With respect to expressive timing, events of the first type might use rubato as expressive means- the change of tempo over the sequence. Events of the second type might use "chord-spread" or asynchrony between voices as expressive means, both of a more timbral nature. By assigning a structural type to a collection of musical objects, their behavior under transformation is uniquely determined. For example, when a collection of notes is described as a *Sequential* structure it will be associated with tempo, scaling their timing in an logarithmic way. Although the precise timing may change, the sequential order will be fixed and cannot be changed as a result of a transformation. When the same collection of notes is described as a *Parallel* structure, a chord, they will be associated with asynchrony, scaling their timing in a linear way. In this case the order of their onsets may be changed.

Another distinction that was made is an ornamental quality, another possible relation between two musical objects. Ornaments in music, like a *grace note* (a short note played just before a more important one) behave differently. For example, in playing a piece in a higher overall tempo it may well be that the duration of a grace note is not affected, it keeps its original duration (Desain & Honing, 1994). Ornaments can be divided in *acciaccatura*, so called timeless ornaments, and *appoggiatura*, ornaments that take time and can have a relatively important role in e.g.. he structure of melody. This specific behavior under transformation can be compared to a PostScript description of a picture. In this format some graphical data scales proportionally with the size of the final output, and others, like the width of hairlines, does not.

In Figure 2 a score in common music notation is given. In Figure 3 two graphical representations of the structure of the same piece, a metrical and a voice analysis, are given in the form of boxes for each of the notes and their structural groupings. Enclosing boxes represent the *part-of* relation between part and whole, where compound objects can have different temporal (successive or simultaneous) and ornamental qualities (multilateral or collateral). The possible values of these two basic qualities can be combined orthogonal to give four kinds of compound structural units (**S**equential, **P**arallel, **ACCIA**ccatura and **APPOG**giatura) which deal with representing a large class of musical concepts naturally.

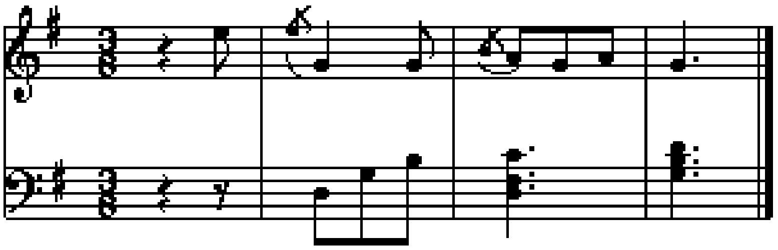

Figure 2. Score of the last bars of the theme of six variations over the duet Nel cor più non mi sento, *by Ludwig van Beethoven.*

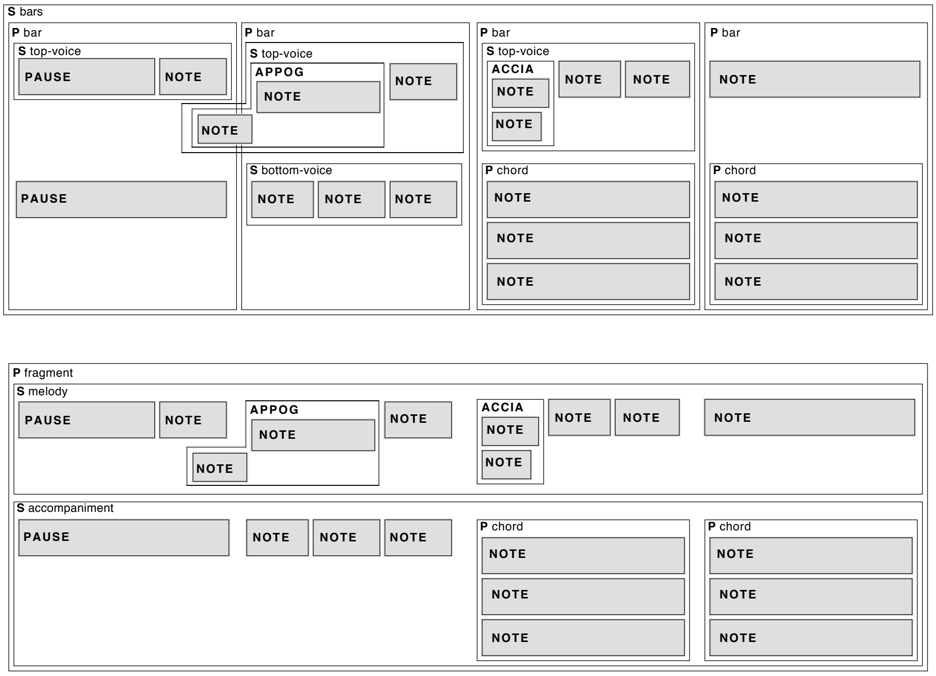

Figure 3. a) Structural description of the metrical hierarchy of the score in Figure 2, and b) Structural description of the voices in that piece.

Unfortunately, expressive parameters in music (like timing or dynamics) cannot be divided into small and elegant sets of orthogonal types. There is always some kind of interaction - one type of expression in a representation of a performance can not be changed without influencing the other. For example, shifting the onsets of a collection of notes will interact with the expression through the articulation (i.e. the interval between the offset of one note, and the onset of another). One of the problems in the design of the Calculus was to factorize the knowledge involved and to make these kind of interactions explicit and controllable. Below we will describe some of the mechanisms that the Common Lisp Object System provides, and that turned out to be useful in the development of the Expresso system. (We assume that the reader has some familiarity with object-oriented languages and related terminology.)

Common Lisp Object System (CLOS)

The central concepts of CLOS are classes, instances, generic functions and methods (see Steele, 1990). The language elegantly supports the use of an integrated functional- and object-oriented programming style in one language (see Gabriel, White & Bobrow, 1991), and advanced object-oriented constructs like multiple and mixin inheritance, multi methods and method combination. Its definition includes a metaobject protocol (Kiczales, Rivières & Bobrow, 1991), that defines the full semantics of CLOS in CLOS itself. This is not just a theoretical exercise, but enables the programmer to elegantly extend or change the language itself by modifying the metaobjects that implement the concepts like classes, slots and generic functions.

Using an object oriented style in representing musical knowledge

Multiple inheritance, as used for modeling musical objects

In general, it is claimed that simple inheritance schemes are too rigid for a complex representational problem (like the design of user-interfaces or operating systems). In *single inheritance* a class can only inherit behavior from one super class.

In *multiple inheritance*. it is not only allowed for a class to inherit from a set of direct super classes (a perfectly neat way to factor knowledge), but they may in turn inherit (possibly indirectly) from the same class. This supports different kinds of class organizations in the form of a general directed a-cyclic graph whereas simple inheritance is restricted to tree-like hierarchies only.

While single inheritance sufficed to model the basic musical objects in the Expresso system multiple inheritance was used in the first design to organize the structured (compound) musical objects (see Figure 4).

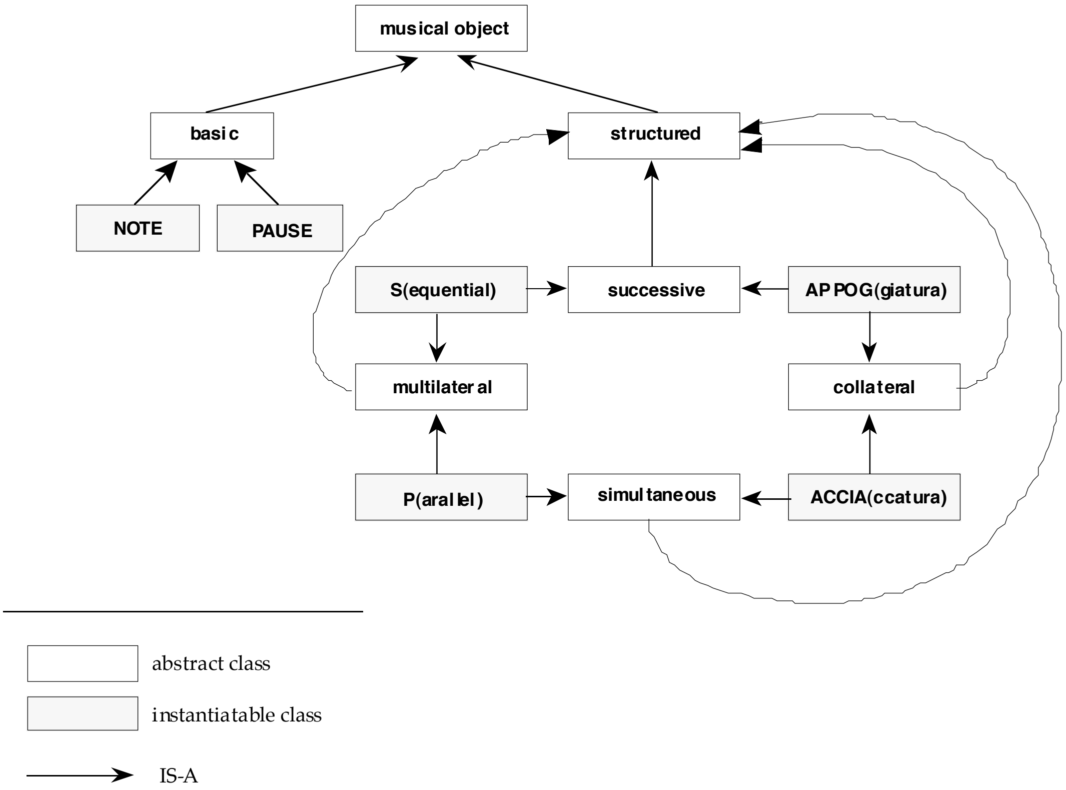

Figure 4. Classes of musical objects and their interrelations.

For these musical objects there were two orthogonal aspects to model. The first is an ornamental relation between parts that may or may not exist (collateral vs. multilateral). The second is the time order between the parts (successive vs. simultaneous). Concrete (instantiable) classes were defined for each of the four combinations. So a sequential structure inherits behavior from the successive and from the multilateral class, which both are structured musical objects. One can see the complication that may arise when the same method is defined in the different classes: inheritance may work via different paths and this freedom calls for the need of a consistent algorithm to calculate the superclass that a specific method is inherited from. Although this algorithm is well defined in CLOS, and the factoring of behavior into classes as presented for Expresso works well, it is in general not wise to program too complex class relations. This is because different methods all have to make use of the same structure of classes for specifying their inheritance. Furthermore, when one relies too heavy on the inheritance mechanism, for example, to decide which method shadows which, the order in which direct super classes are specified in the class definition becomes very important (and may need to be different for different methods too). So in general, it is wise to put some extra effort in deciding to which class a particular behavior belongs. In other words, when knowledge is properly factored there is less need for complex inheritance schemes. For our domain this resulted in revising the inheritance scheme of Figure 4 such that the instantiable classes are now the only classes that inherit from the class of structured objects.

Mixin inheritance, as used in modeling types of expression

The types of expression available to a performer constitute a rich world – even when the basic note objects only have a few parameters like in keyboard music (e.g. onset, offset and loudness). Based on these few note attributes, many types of expression can be distinguished: articulation (the amount of sounding overlap between notes in a sequence), chord asynchrony (the spread of the actual onset times of notes that belong to the same chord), voice asynchrony (e.g. the way in which melody and accompaniment are played out of phase), expressive tempo fluctuations (e.g. rubato), dynamic contour (the sequential patterns of loudness linked to e.g. the metric structure of bars and beats and dynamic balance (e.g. the distribution of loudness over melody and accompaniment or the different notes in a chord. The richness of this domain is a result of the interplay between attributes of notes and the structural musical objects that form the context of the note. However, the types of expression classes needed to describe the intrinsic expressive aspects of isolated notes are easy to deal with. The data structures used in Expresso to represent expression are empty, they contain no data, which might be surprising. But because the classes of these objects form part of an elaborate inheritance network, the expression objects themselves function as hooks for specific procedural knowledge controlling method dispatching, e.g., indicating which method has to be used for modifying a certain type of expression.

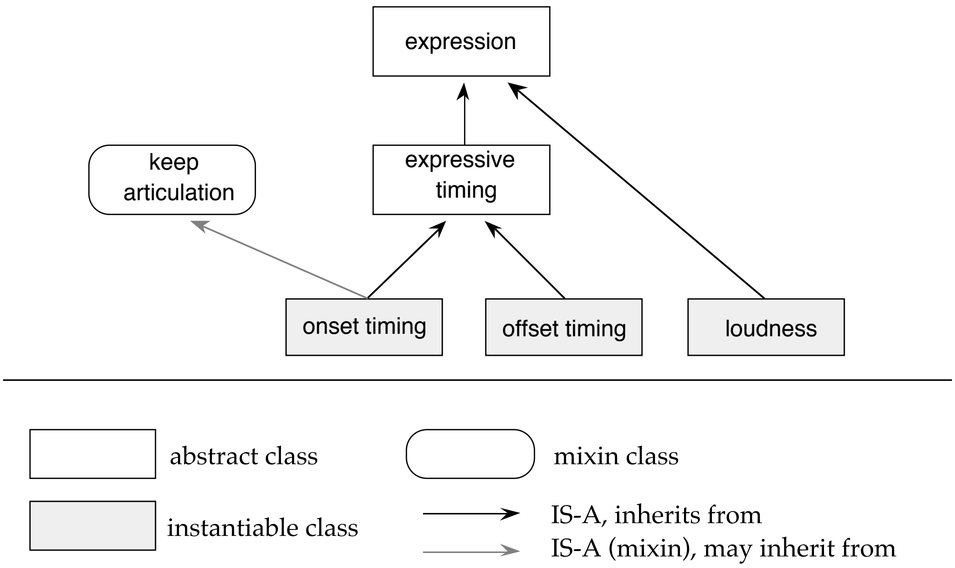

Figure 5 Simplified expression type hierarchy for timing and loudness expression.

The type of inheritance used for modeling expression is *mixin inheritance*. It allows a class to be used as an optional modifier of the behavior of a class that already functions well on its own. A mixin is an abstract class, it will never be used on its own, but always in combination with some other class to which it adds structure and behavior. Typically, this customization is supported in *before*, *after* and *around methods*. A mixin class is designed not to interfere with other aspects of behavior, apart from the aspect of the behavior that it affects. Thus often the customization is expressed in the additional code, no primary methods are shadowed or rewritten. The design of a collection of mixin classes should be as an orthogonal independent set, such that each can be included or left out at will when mixing a new class.

Mixin inheritance proved an elegant mechanism to model interactions between related types of expression. For example, a `keep-articulation` mixin provides additional behavior to `onset-timing` expression. When the onset timing is affected, for instance by a manipulation of the rubato, the offsets are moved consistently to maintain the same sense of articulation. And, of course, the `articulation` class itself is a choice among possible alternatives. In Figure 5 a part of the inheritance structure is shown, using single inheritance for expression and mixin inheritance to add offset consistency as an option to `onset-timing`.

Multi methods, as used in extracting expression from musical objects

A method will be evoked only on arguments that satisfy its parameter specializers, the method dispatch takes care of selecting the appropriate method depending on the classes of its arguments. In some object-oriented languages (like Smalltalk and older object-oriented Lisp extensions), a method can specialize on one object only. This is congruent with a message-passing paradigm. But often when a function has more arguments it behaves more or less symmetrical with respect to them, and deciding which argument to send the message seems arbitrary. For instance, it makes no sense to decide if it is better to ask a musical object to extract a certain type of expression or to ask a musical expression to consider a certain musical object. And if one is forced to make that choice (as in the message passing paradigm) unnatural programs result. It is much more natural to be able to write a method for extracting expression that captures only a combination of a specific temporal structure and a specific expression type. In CLOS these are available as so called *multi methods*, functions that are polymorphic in more than one argument.

In Expresso multi-methods are used to extract expression, depending on both the type of musical object (note, pause, S, P, etc.) and the type of expression (onset-`timing` and loudness). The method `get-expression` for the musical object `pause` returns nothing, for every type of expression, because a rest can carry no expression in itself. Ornaments (inheriting from the class `collateral`) have a specific method that extracts all expression from its main component. The `get-expression` method for `onset-timing` expression for a sequential structure is the onset of its first component, whereas the onset of a parallel structure is the onset of the component that happens to be first in time. Furthermore, for loudness expression parallel voices are summed, whereas for sequential structures the profiles are averaged, and a note can return its amplitude directly. In this way a condense description of the calculation of expression profiles, and how that process is determined by both the type of a musical object and the type of expression, can be achieved using multi methods.

A similar set can be defined to actually set expression of musical objects. The generic function `set-expression` is specialized for a particular type of musical object, a type of expression and an expression map (a data structure containing a description of expression with a class organization similar to musical objects). The different method apply the appropriate change to the particular type of object, propagating the changes through a possibly complex structure of nested `S`, `P`, `ACCIA` and `APPOG` objects. Thus setting the onset timing of a sequential structure will propagate changes to its components in an interpolated way, and setting it for a parallel structure will truncate changes of its components if they are in danger of moving outside the time interval of the enclosing structure. The changes in timing for ornaments also have their own specified behavior.

Method combination, as used to maintain consistency between types of expression

In CLOS, a method can be composed from parts through a technique called *declarative method combination* (as opposed to procedural method combination, used in, for example, Smalltalk or Beta). In method combination, one can control how partial descriptions of behavior of one method for more classes are combined to allow for more complex combinations than mere shadowing. In the *standard method combination* all so-called *before*, *after* and *around methods* are assembled in their proper order and wrapped around the main or primary method. Any behavior that can be conceptualized as a simple side effect to take place before or after some main behavior already defined, can be added as before or after method. The source of the primary method need not be available – also predefined behavior, for example of the system itself, can be modified in this way. A modification that needs the result of the old primary method can call it using `call-next-method`.

In Expresso a `get-expression` method for the multilateral object is defined that just collects the expression of its components in a list. The appropriate around method defined for Sequential and Parallel structures then extracts only has to extract the expression of the whole from the list of expression values from the parts. In that way knowledge about multilateral objects (all parts are equally important and may in principle contribute to the expression) and about temporal order and type of expression (the onset timing of a parallel structure is the onset of its part that happens to be earliest) can be split and described in the appropriate places, avoiding duplication of code.

This distribution of control in standard method combination has a clean declarative style (using before, after and around methods). However, it is allowed to use `call-next-method` in primary methods too. This procedural style blurs the flow of control and is more difficult to debug, because the programmer has to search through the body of all primary methods to check whether it contains a `call-next-method`. In general, it is better to restrict the use of `call-next-method` to around methods.

Besides the standard method combinations, there are built-in method combinations that define other ways of combining primary methods (like appending or adding their results), instead of the standard shadowing. Furthermore, there are fully programmable ways for users to add their own way of method combination.

Metaobject Protocol

In CLOS it is possible to extend the language using CLOS itself because all language constructs (like classes, methods and inheritance mechanisms) are accessible as CLOS objects in a meta-circular manner (see Kiczales, Rivières & Bobrow, 1991). In that way one can, for example, customize how an object is instantiated from a class. This can be used in programming the expression objects. These objects have no slots, they are empty objects and are only used for dispatching the different methods. This means that the creation of new fresh instances of the expression class is not needed more than once, since all instances will be the same. A resource of these instances suffices to prevents the creation of redundant instances. Of course this could be programmed. But one would like to modify the make-instance method such that this can be hidden for the programmer: empty objects seem to be created in the same way as any other object, only in creating new ones an old instance is returned whenever it is available. Because this entails a calculation on a class it is not possible to implement this in CLOS directly on the programming-level interface. But since a CLOS class is like any CLOS object, and each class is an instance of a meta-class, a new meta-class can easily be defined, inheriting all behavior of the standard meta class, but adding some specific behavior to make-instance. In this way constructs that are part of the language definition itself (like the way in which new objects are created) can still be redefined using the metaobject protocol. (Note that the metaobject protocol is in the process of standardization and is not yet fully supported in some Common Lisp implementations.)

Maintenance and change

The process of factorizing structure and associated behavior is often not easy. One has to go through several stages of re-design to end up with a stable set of concepts and their relations that naturally reflect the domain modeled. This re-design process is not always well supported by object-oriented languages and their development environments. For instance, to change a solution using inheritance (*is-a* links) into one using links between instances (*part-of* links) is a cumbersome job in every object-oriented language.

Furthermore, while is-a relations are supported quite well, one can identify a lack of support for part-of relations. Often simple concepts like back pointers or mapping constructs that are used heavily in building object networks have to be defined procedurally by the programmer.

But even in the domain of inheritance, some central facilities are lacking. While the programmer might design neat sets of abstract classes and mixins, there is, for example, no support for expressing and enforcing their correct use, like the order in which classes have to be used as superclasses or the methods that have to be defined to implement a new mixin facility of a certain kind.

With respect to CLOS, some solutions to these problems can be solved through the availability of the Metaobject protocol and the software development protocol (see Gabriel, White &

Bobrow, 1991) that further standardize the implementation of the language, and gives the programmer access to the implementation of the language itself.

Conclusion

In this paper, we briefly touched some relatively new aspects of object-oriented programming As available in CLOS, and described the use we made of them in the design of the Expresso system. Normally, object-oriented languages are considered useful in the design of large, multi-faceted systems. But also in the domain of formalizing and representing musical knowledge, where relatively small aspects of that knowledge have to be explored, modeled and combined, an object-oriented style can be of use. However, we found that while part of the problems are resolved by some important new language constructs (like multiple and mixing inheritance, multi methods and method combination), some of the enthusiastic claims are not fulfilled

Acknowledgments

The research of the authors has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences (KNAW). We would like to thank our colleagues from the NICI institute in Nijmegen and the Computational Linguistics department and the ILLC in Amsterdam that form such a stimulating research environment.

References

Anderson, D. P. & R. Kuivila (1991) Formula: a programming language for expressive computer music. Computer IEEE, July.

Clarke, E. F. (1988) Generative principles in music performance. In J. A. Sloboda (Ed.) Generative processes in music. The psychology of performance, improvisation and composition. Oxford: Science Publications.

Desain, P. & H. Honing (1991) Towards a calculus for expressive timing in music performance. Computers in Music Research. Fall Vol. 3: 43-120. Reprinted in Desain & Honing (1992).

Desain, P. & H. Honing (1992) Music, Mind and Machine, Studies in Computer Music, Music Cognition and Artificial Intelligence. Amsterdam: Thesis Publishers.

Desain, P., & Honing, H. (1993). CLOSe to the edge? Multiple and mixin inheritance, multi methods, and method combination as techniques in the representation of musical knowledge. In Proceedings of the IAKTA Workshop on Knowledge Technology in the Arts. 99-106. Osaka: IAKTA/LIST.

Desain, P., & Honing, H. (1994). Does expressive timing in music performance scale proportionally with tempo? Psychological Research, 56, 285-292.

Gabriel, D. G. & G. S. Steele (1990) The failure of abstraction (editorial). Lisp and symbolic computation. Dordrecht: Kluwer.

Gabriel, R. P., J. L. White & D. G. Bobrow (1991) CLOS: Integrating Object-oriented and Functional Programming. Communications of the ACM. 34(9): 28-38.

Honing, H. (1992) Expresso, a strong and small editor for expression. In Proceedings of the 1992 International Computer Music Conference. San Francisco: ICMA. 215-218.

Honing, H. (1993) A microworld approach to the formalization of musical knowledge. Computers and the Humanities, 27: 41-47.

International MIDI Association, (1983). MIDI Musical Instrument Digital Interface Specification 1.0. North Hollywood: International MIDI Association.

Keene, S. E. (1989) Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. Reading, MA: Addison-Wesley.

Kiczales, G., J des Rivières & D. G. Bobrow (1991) The Art of the Metaobject Protocol. Cambridge, MA: MIT Press.

Pope, S. (1991) The Well-Tempered Object, Musical Applications of Object-Oriented Software Technology. Cambridge, MA: MIT Press.

Steele, G. L. (1990) Common Lisp, the Language. Second edition. Bedford, MA: Digital Press.