

Object-Oriented Design in the MODE

Stephen Travis Pope

Nomad Object Design, *Computer Music Journal*,
Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music, Stanford University
P. O. Box 60632, Palo Alto, California 94306 USA
(currently) Swedish Institute for Computer Science: SICS
Box 1263, S-16428 Kista, Sweden
stp@CCRMA.Stanford.edu, stp@SICS.se

Abstract

This paper presents two aspects of the design and implementation of the Musical Object Development Environment (MODE) system with respect to the application of object-oriented analysis, design and development methodologies. The MODE is a framework and tool kit for the development of music-related software applications written in the Smalltalk-80 programming language and embedded in the Objectworks\Smalltalk (TM ParcPlace Systems, Inc.) programming environment.

The analysis and modeling of the basic magnitudes of musical objects is addressed in the design of class libraries for SmOke, the MODE's basic music representation. The second topic is the design and implementation of a highly-reusable framework of user interface components for graphical event, event list, function, sound, signal, and other data inspectors, editors, and browsers—the Navigator MVC framework and tool kit.

Introduction

The Musical Object Development Environment (MODE) is a Smalltalk-80 (Goldberg and Robson, 1989; ParcPlace Systems, 1992) framework and tool kit for music description, score editing, interactive performance, and digital signal processing. It consists of approximately 180 Smalltalk-80 class modules with over 2000 methods in about 900 KBytes of source code (90% Smalltalk, 10% C). The MODE consists of Smalltalk-80 classes that address five areas: (1) the representation of musical parameters, sampled sounds, events and event lists; (2) the description of middle-level musical structures; (3) real-time MIDI and sound I/O and DSP scheduling; (4) a high-level user interface framework and component set for building function, signal, event, and structure manipulation applications; and (5) several built-in end-user applications for signal and event structure editing and digital signal processing.

Figure 1 is a schematic presentation of the basic modules of the MODE environment. The items in Fig. 1 are the MODE packages, each of which consists of a collection or hierarchy of Smalltalk-80 classes. The items on the right that are displayed in courier font are written in C and are the interfaces to the external environment via coprocesses that handle MIDI and sampled sound I/O.

>>> Figure 1 MODE Software Components<<<

The “kernel” of music magnitudes, functions and sounds, events, event lists and event structures is known as the Smallmusic Object Kernel (SmOke) music representation. SmOke is described in terms of two related textual description languages (music input languages), a compact binary interchange format, and a set of data structures (one implementation of the representation itself). The other high-level packages—voices, sound/DSP, compositional structures, and the user interface framework—are shown surrounding the SmOke kernel in Figure 1.

“Executive Summary” of the SmOke Music Representation

The SmOke representation can be summarized as follows. Music (a musical surface or structure) can be represented as a series of “events,” which generally last from tens of msec to tens of sec. Events are modeled as property lists or dictionaries; they can have named properties whose values are arbitrary. Property values may be music-specific objects (such as pitches or spatial positions), and models of many common musical magnitudes (e.g., duration, pitch, and loudness) are provided.

Voice and accessor objects determine the interpretation (semantics) of event properties, and may use “standard” ones such as pitch, loudness, voice, duration, or position. Events are grouped into event lists by their start times (described as durations relative to the enclosing scope). Event lists are events themselves and can therefore be nested into trees, i.e., an event list can have another event list as one of its events, etc; they can also map their properties onto their component events. This means that an event can be “shared” by being in more than one event list at different relative start times and with different properties mapped onto it.

Events and event lists are “performed” by the action of a scheduler passing them to an interpretation object or voice. A scheduler expands and/or maps event lists and “sends” their events to their voices. A voice maps event properties onto the parameters of some device or stream format; there is a rich hierarchy of voices for abstract MIDI and sampled sound I/O from/to real-time interfaces or files.

Stored data functions can be defined and manipulated as breakpoints for interpolation, summation parameters, “raw” data elements, or combined functions of the above. Sampled sounds are also describable by means of synthesis or signal processing scripts involving a vocabulary of sound manipulation messages.

The SmOke representation can be linearized as text in any data description or programming language that supports the following immediate types: arbitrary-precision integers, fractions, and 32- and/or 64-bit floating-point numbers; ISO strings and unique symbols; basic data structures such as (key-value) tuples, dictionaries, and sorted collections; and closures (as in Lisp), execution block contexts (as in Smalltalk).

Three other documents describe MODE and its design. (Pope, 1991) is a basic description of the MODE version 1.0; (Pope 1992a) focuses on the system design issues and low-level interfaces of the MODE running on a SPARCstation computer, and (Pope 1992b) describes the SmOke representation language in detail.

MODE OOA/D Topics

Several components of the MODE are especially interesting because of the use of object-oriented abstraction and factoring for flexible representation, ease of extension, or very high levels of reusability. The components presented below are: (1) the music magnitudes describing the basic properties of event and sound objects in music; and (2) the visualization and editing components for manipulating signals, scores and other data structures.

MODE Music Magnitudes

Building a music representation begins with the definition of models of the fundamental parameters of musical events and surfaces—the partially- or well-ordered scalar or multi-dimensional properties of musical objects: time/duration, pitch/frequency, loudness/amplitude, spectrum/timbre, position/spatialization, etc. Music description languages in the literature differ greatly in the extent to which they provide abstract and flexible models of these magnitudes; in the design of SmOke, a group of composers and software engineers defined a set of desired operators and modes of arithmetic combinations to be used for signal and event description (Smallmusic, 1992). A pitch magnitude, for example, should be describable as a string or symbolic note name (e.g., ‘c3’ pitch), a MIDI¹ key number (56 key), a Hertz frequency (440.0 Hz), a fractional ratio to another pitch ((11/9) of: (440.0 Hz)), or an offset in cents (logarithmic percent of a well-tempered semitone) (‘f#4’ pitch + 8 cents). We would like to be able to describe compound pitch ex-

1. MIDI is the Musical Instrument Digital Interface, a system based on an RS-232 hardware connection and simple protocol used for communicating event-oriented data between computers, controllers (e.g., electronic piano keyboard devices), and synthesizers.

pressions such as ('f5' pitch + 24 cents + 6 Hz) and to allow voices to coerce pitches into specific representations, e.g., ('f5' pitch asMIDIKeyNumber) or ((54 key) asHz). The framework for music magnitudes should be easily understandable and readily extensible so that composers can customize (e.g., adding a new type of pitch description for one specific piece).

Earlier versions of the MODE² had hierarchies of classes with abstractions such as Pitch, Loudness, or Duration, and many concrete subclasses with largely duplicated code that used nasty type-checking messages, case statements, and the questionable notion of generality to try to provide abstract and concrete music magnitude behaviors as described above.

It became obvious that the issue of “ representation versus implementation” was not being handled elegantly; what was needed was a good method to factor the classes in two dimensions so that, e.g., the pitch object (440.0 Hz) had some behaviors of Pitch and some of Floating-point-number. Using multiple inheritance would have blurred these dimensions by having overlapping inherited behaviors, required a large number of coercion messages, and distributed the behavior in an arbitrary manner among the representation and implementation classes. Using standard single inheritance (as in the HyperScore Toolkit), lead to a very large implementation that required complex and distributed changes to extend and in which abstract models of the representation dimension (e.g., the concept of Chronos that unifies time and duration and has subclasses for Duration and Meter/ Metronome) were missing and difficult to ad.

The decision was made to model music magnitudes using the Smalltalk-80 “ species” mechanism for the representation dimension, and the class inheritance hierarchy for the implementation dimension. The species mechanism is very simple; class Object implements a method for the message “ species” that answers the receiver’s class. One can override this in application classes to answer any class, and use this to share behavior across class hierarchies by having methods in concrete classes that delegate via the (abstract) species. Each concrete music magnitude instance delegates some behaviors via its species, and uses some inherited from its superclass.

This design yields two related class hierarchies, one of abstract and “ concrete” representational models (which are still not instantiated), and one of implementation classes as shown in Figure 2. Note that both hierarchies have abstract and concrete classes, but that the representation classes (and the abstract implementation classes) are not instantiated.

2. known in various versions as the HyperScore Toolkit and Topaz

Class `MusicMagnitude` is a subclass of Smalltalk-80's `Magnitude` class and inherits and refines comparison behavior from it. It adds a “value” instance variable, and has basic behaviors common to the representation and implementation subclasses, e.g., arithmetic, identity testing, and simple operations. Figure 2 shows the representation classes in the left-hand column, with abstractions for the basic models; the order and indentation of this list shows the class inheritance hierarchy of these classes. The right-hand column is the standard implementation hierarchy with abstract and concrete models. The lines between the hierarchies show the species relationships of several common concrete classes. The representation classes are not often subclassed, but each serve as the species for a family of concrete implementation classes (though the members of this family need not be directly related in the implementation hierarchy, as shown in the Figure). The abstract classes in the implementation hierarchy introduce assumptions about the type and range of the magnitude object's value, as outlined on the right of Figure 2.

>>> Figure 2 Music Magnitude class trees <<<

This architecture leads to high levels of reuse among the implementation classes, making them very “light weight.” Concrete classes (e.g., `HertzPitch`), will now be subclasses of one of the representational abstraction classes (e.g., `NumericalMagnitude`), and will answer one of the model abstraction classes (e.g., `Pitch`) as their species. Mixed-mode arithmetic is now performed by defining a generality hierarchy for each species that is used for type coercion using double dispatching—implementing multiple polymorphism by dispatching to the argument of arithmetic expressions so that [`f#3' pitch + 28 Hz`] becomes [`28 Hz plusMIDI: 'f#3' pitch`] (Hebel and Johnson 1990). Each concrete class implements a small number of common coercion messages, with the rarer forms left to the species (which coerces to the most general form for the two instances and delegates).

The SmOke description allows music magnitudes to be described using either messages sent to the model classes (verbose format), or with post-operators sent to numbers, strings, etc. (terse format). Several description and coercion examples are shown below.

>>> Figure 3 Music Magnitude Examples <<<

Music magnitudes are used in MODE as the properties of event objects, whose semantics is given by the voices that they are played on or read from, and the structure accessors that are used as interfaces to them. They allow a composer to play a melody (e.g.,) into a composition system from a MIDI synthesizer keyboard, then pass the data to be interpreted by a graphical score editing application (which may add many new properties to the events), then perform the events using a software sound synthesis language by “playing” the events on voices that interpret their properties to generate note list files.

Extending the representation by adding new concrete music magnitudes is a three-step process that consists of: (1) subclassing one or more of the implementation classes refining the coercion and I/O methods; (2) adding coercion messages to the species class; and (3) adding new post-operator messages to some immediate class (e.g., String or Number).

The implementation of MusicMagnitudes in Smalltalk-80 in the MODE is compact, readable, and easily extensible with a high degree of reuse; it provides SmOke with a powerful description and modeling layer. The technique of using linked species and class hierarchies for the dimensions of representation and implementation leads to a cleaner factoring of the behaviors—and greater ease of extension—than a solution based on uniform single or multiple inheritance.

The technique described here is portable in that the species mechanism can easily be added to any object-oriented language that supports classes as objects. It is worthy of note that this type of description of families of domain-specific magnitudes is often needed in other description, modeling or simulation applications, and that models for many domains (other than music) can be implemented using similar techniques.

Navigator MVC and Score and Structure Editors

The issue of high-level applications frameworks is of great interest within the OOP community. The Smalltalk-80 Model-View-Controller (MVC) user interface paradigm (Krasner and Pope 1988), is well-known, widely imitated (e.g., MacApp or NeXTStep), and also widely misunderstood. The traditional three-Part MVC architecture is illustrated in Figure 4, which shows the central model object representing the state and behavior of the domain model—in our case, this could be an event list structure or sound signal, for example. The view object presents the state of the model on the display—it's the output; and the controller object sends messages to the model and/or the view in response to user input—typically communicated via the mouse or the keyboard. Note the separation between modeling, presentation and interaction, and the loose coupling (via the Smalltalk-80 dependency mechanism) between the model and the view/controller pair. The control flow in MVC is that the user takes some action that is noticed by the controller, which will probably send a message to the model in response. The model changes state and broadcasts the change to the—zero or more—“dependent” objects (e.g., the view). The dependent objects then may update themselves, as in a view redisplaying some aspects of the model's state in a window on the screen. These change/update messages may be parameterized by “aspects” of the model, so that multiple views or subviews can present several perspectives of the state of a complex model.

>>> Figure 4 3-part MVC <<<

Many Smalltalk-80 applications extend this structure somewhat in that a fourth object is added to factor out the state of the application from the model, as shown in Figure 5. These intermediary objects are normally called inspectors, editors, or browsers. They may hold onto the selection state of the application (so that multiple applications can manipulate the same model at the same time), or translate messages between the view / controller and the model (so that these are more reusable across models).

>>> Figure 5 4-part MVC <<<

A further refinement of the MVC architecture was developed by David Leibs, Adele Goldberg, and the author in “ Navigator MVC”(Pope, Harter, and Pier 1989), an alternative factoring of the editor and view for higher levels of reuse. The fundamental feature of this architecture is that all applications are built as display list editors (i.e., the generic tool is “ smart MacDraw”), with special objects for translating the model structure into a graphical display list representation and for translating structure interaction into model manipulation.

Several OOD principles were used here; the first is that those classes that have complicated algorithms or complex behaviors—e.g., layout managers and views—should be reusable as elements in a tool kit (i.e., by composition), whereas those components that are application-specific and will frequently be refined by subclassing—e.g., structure accessors and editors—should be very simple and highly parameterized. The second principle is to provide multiple paradigms for modeling each layer or domain, meaning for example a declarative or procedural style for describing SmOke magnitudes, or imperative or declarative models of behavior in editors and structure accessors.

The structure of a Navigator application is shown in Figure 6, with the view holding a structured graphics display list (and possibly also a cached pixel map), the editor holding the model, a layout manager (used to generate the display list from the model), and a structure accessor (used by the layout manager and editor for accessing the model’s internals via a standard protocol). The implementation of the Navigator framework in the MODE includes several layers of highly-reusable components for building applications using many types of models.

>>> Figure 6 Navigator MVC <<<

The basic class hierarchies for display items, -lists, -views, -editors, and -controllers support abstract structured graphics display and editing. The protocol for the creation of display lists supports many interfaces for programmatic generation of graphical displays. Display list viewing is eased by the rectangle intersection and translation protocol of display lists, which allows them to be used as standard visual components within Smalltalk

graphics contexts. Display list view subclasses generally add new view creation methods and a few instance methods to generate display items using the appropriate layout manager and structure accessor.

The middle layer is the rich hierarchy of layout manager and structure accessor classes, which support list, outline, tree, sequence, pitch/time and other structured layouts. According to the design method discussed above, structure accessors and editors are modeled using “pluggable adaptor” or “protocol translator” objects that provide a standard message interface to layout managers and editors and are parameterized with symbolic message selectors or block closures to interoperate with the model. An example of this is the class `TreeAccessor`, which can be used to manipulate nested hierarchical structures. It has parameters for the method that accesses the name and the children of a “node” in the model, and can be used (e.g.,) for navigating within a class hierarchy, a structured document, or any other “tree-like” data structure. A layout manager that reads tree structures and draws pretty trees with the root at the left and links shown with lines can use this accessor to draw any of these structures in the same presentation style.

The top level is the MODE applications—a set of browsers, inspectors, graphical editors and views/controllers for related MODE model classes. The basic tool kit includes function, signal, sound, event, event list, and other structure editors and type-specific browsers. The partial hierarchy of the MODE application views, each of which has related editor and structure accessor classes, is shown in Figure 7.

>>> Figure 7 MODE Application View Hierarchy <<<

Figures 8 through 12 show several views from the MODE’s musical and digital signal processing applications. Figure 8 shows three different signal manipulation tools: a multimodel function editor; a sound hierarchy browser; and a speech analysis editor for linear prediction-based cross-synthesis of sounds. Event list editors of various types are shown in Figure 9, where time sequence, pitch-time and fancier views use one- or two-levels of layout managers for generating positions and display items. Figure 10 shows several hierarchy views for tree- or list-like layout of class or event list hierarchies. The sound file mix editor, which uses two layers of layout managers, is shown in Figure 12.

>>> Figure 8 Signal Editors and Browsers <<<

>>> Figure 9 Event List Editors <<<

>>> Figure 10 Exemplary Hierarchy Layouts <<<

>>> Figure 11 Sound File Mix Editor <<<

Conclusions

The modeling and design of components for music representation and graphical user interfaces to complex music-related data structures can be used as an example domain to present several topics in object-oriented software methodology. The two issues discussed above related to the Musical Object Development Environment (MODE), demonstrate the application of modern software engineering principles to the construction of tools for a variety of domains. The author believes that these principles and design hueristics can easily be applied in other domains, application situations, and programming languages.

Availability

The MODE software system is freely available—as about 5 MBytes of Smalltalk and C source code, SPARCstation binaries of several utility programs, voluminous documentation, and test and example data files—via ftp file transfer from the Internet files anonymous@ccrma-ftp.stanford.edu:/pub/st80/*

Acknowledgments

This manuscript evolved through several versions while the author was associated with the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University, ParcPlace Systems, Inc., the STEIM Institute in Amsterdam, The Center for Knowledge Technology of the Academy of the Arts in Utrecht (HKU), and the Swedish Institute for Computer Science (SICS) in Kista near Stockholm. Many thanks are due to members of the management and staff of these institutions.

References

- Goldberg, Adele, and David Robson, 1989. *Smalltalk-80: The Language*. Menlo Park: Addison-Wesley (updated version of the 1983 volume).
- Hebel, Kurt J., and Ralph E. Johnson. 1990. " Arithmetic and Double Dispatching in Smalltalk-80. " *Journal of Object-Oriented Programming* 2(6): 40-44.
- Krasner, Glenn and Stephen T. Pope. 1988. " A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming* 1(3): 26-49.
- ParcPlace Systems, 1992. *Objectworks\Smalltalk Release 4.1*. Sunnyvale, California: ParcPlace Systems, Inc.
- Pope, Stephen T., Nanette Harter, and Ken Pier, 1989. *A Navigator for UNIX*. Video presentation at the 1989 ACM SIGCHI Conference. Available from the ACM on the CHI '89 video collection.
- Pope, Stephen T. 1991. " An Introduction to MODE." in S. T. Pope, ed. *The Well-Tempered*

Object: Musical Applications of Object-Oriented Software Technology. MIT Press.

Pope, Stephen T. 1992a. " The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers." *Computer Music Journal* 16:3 (Fall, 1992)

Pope, Stephen T. 1992b. " The Smallmusic Object Kernel: A Music Representation, Description language, and Interchange Format." *Proceedings of the 1992 International Computer Music Conference*. San Francisco: International Computer Music Association.

Figure Captions

Figure 1: MODE Schematic Class Diagram

Figure 2: MusicMagnitude Representation and Implementation Classes

Figure 3: Music Magnitude Declaration and Usage Examples

Figure 4: Traditional Three-part Model-View-Controller

Figure 5: Four-part MVC for Browsers, Inspectors and Editors

Figure 6: Navigator MVC Architecture in the MODE

Figure 7: MODE Application View Class Hierarchy

Figure 8: Signal Editors and Browsers

Figure 9: Event List Editors

Figure 10: Exemplary Hierarchy Layouts

Figure 11: Sound File Mix Editor

Figures

Figure 1: MODE Schematic Class Diagram

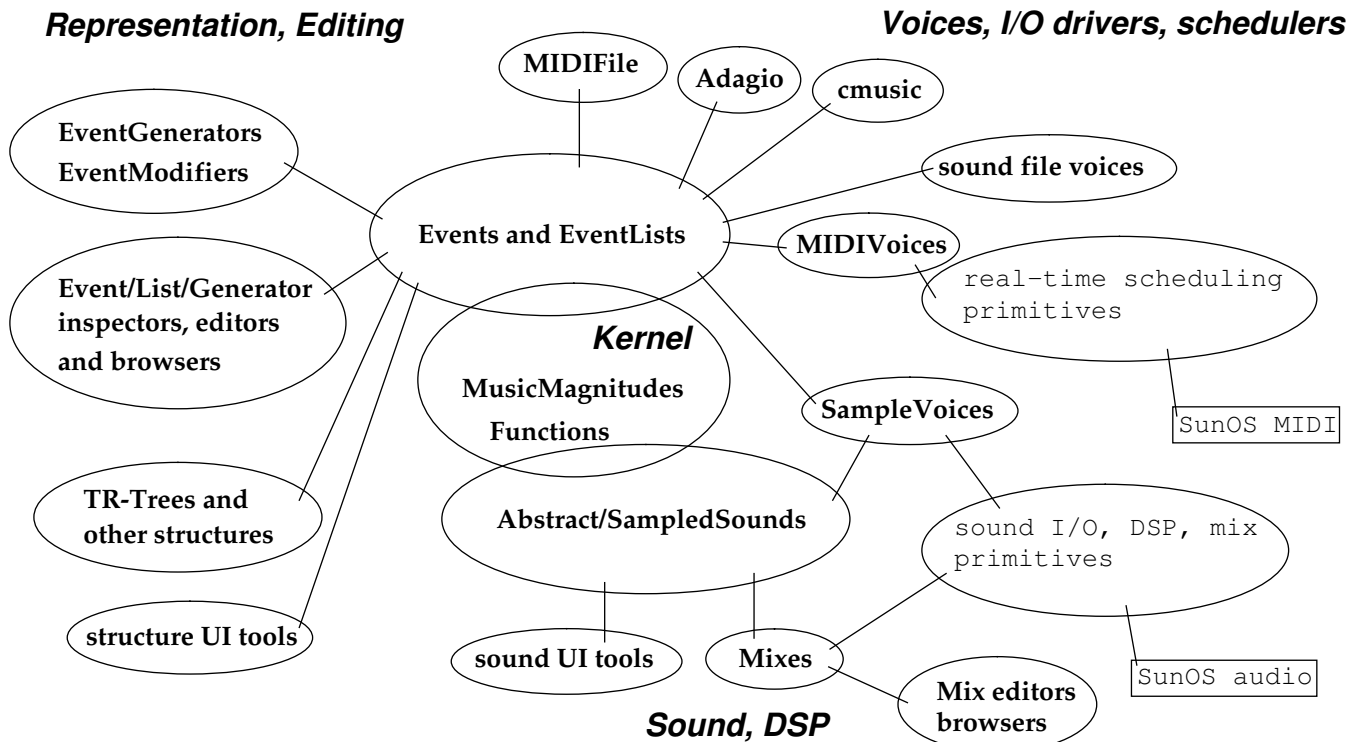


Figure 2: MusicMagnitude Representation and Implementation Classes

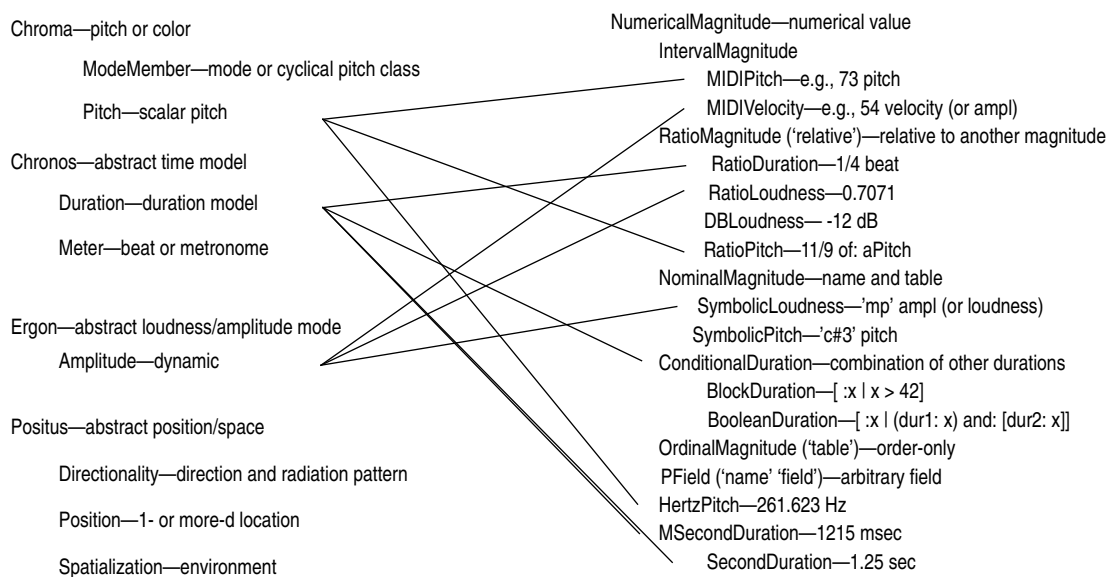


Figure 3: Music Magnitude Examples

"Verbose = class messages sent to rep. classes"
 (Duration value: 1/16) asMS "or [1/16 beat]; answers 62 msec."
 (Pitch value: 36) asHertz value"or [36 pitch]; answers 261.623."
 (Amplitude value: 'ff') asMidi"or [#ff ampl]; = 106 velocity."

"Terse = value + post-op."
 ('f#4' pitch), ('pp' dynamic), (261 Hz asName), (1/4 beat)
 (-38 dB asRatio), (56 velocity), (2381 msec), (0@0 position)
 ((36 key + 200 cents) asHz), ((1/4 beat + 80 msec) asSec)
 (('mp' velocity asRatio value / 2) "answers number 0.15"

Figure 4: Traditional Three-part Model-View-Controller

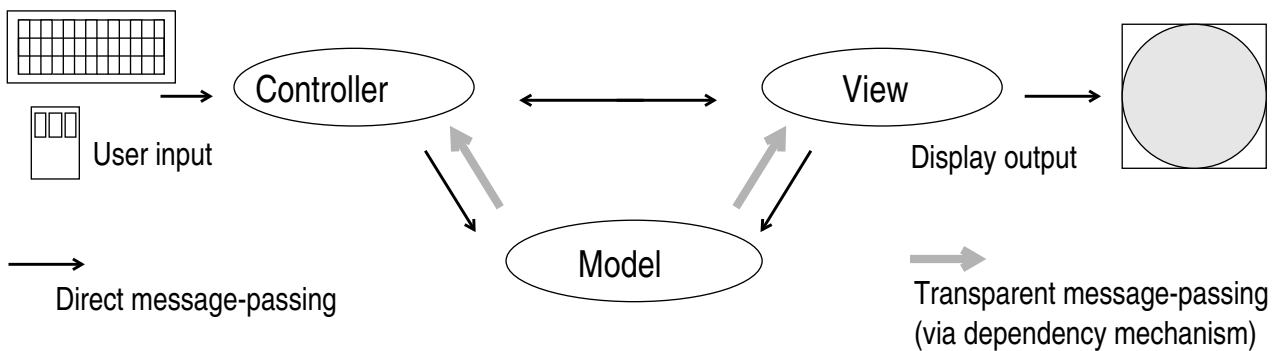


Figure 5: Four-part MVC for Browsers, Inspectors and Editors

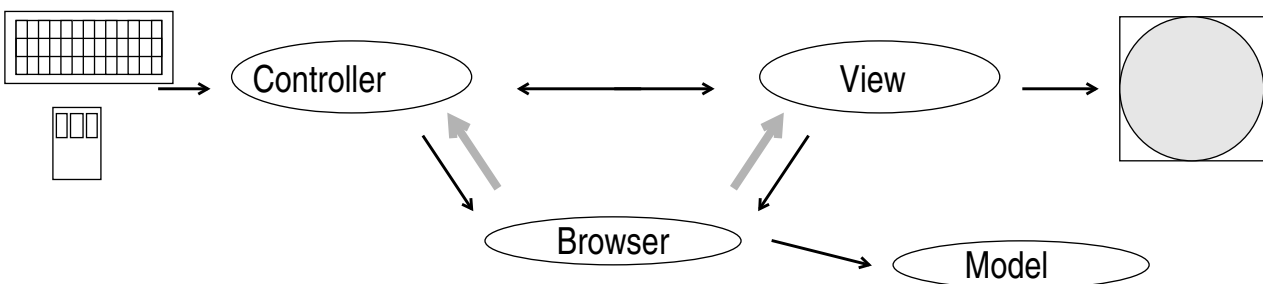


Figure 6: Navigator MVC Architecture in the MODE

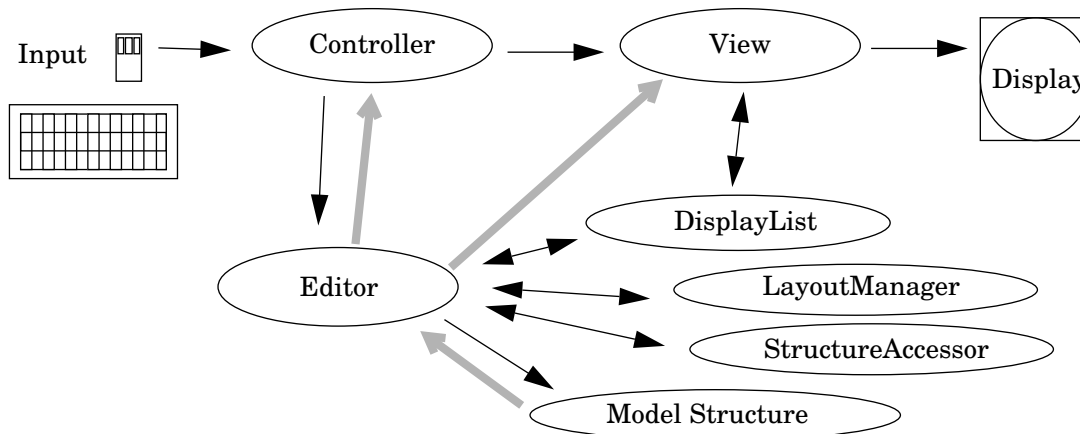


Figure 7: MODE Display List View and LAYout Manager Class Hierarchies

Most view classes have associated layout managers and/or structure accessors. Many also refine the controller behaviors.

```

VisualComponent ()
  VisualPart ('container')
  DependentPart ('model')
  View ('controller')
    AutoScrollingView ('scrollOffset')
      DisplayListView ('displayList' 'pixmap' 'background'
        'zoom' 'grid' 'backgroundColor'
        'foregroundColor' 'redrawn')
    FunctionView ('models' 'aScale' 'vRange'
      'hRange' 'colors' 'width')
    LPCFrameView ('step' 'pScale' 'aScale' 'rScale'
      'pBase' 'aBase' 'rBase')
    PhaseVocFrameView ('colorArray')
    SoundView ('step' 'scale')
    TimeSequenceView ('clefForm' 'xScale' 'xColor'
      'yColor' 'headColor' 'clefColor'
      'itemAccessors')
    GroupingView ()
    PhraseView ()
    PitchTimeView ('pitchOffset' 'yScale')
      CMNView ('steps' 'sharps' 'staves'
        'ledgers' 'xTable')
      HauerSteffensView ()
      MixView ('colorArray' 'buttonOffsets')
      PositionTimeView ()
  
```

```

Object ()
  LayoutManager ('view' 'orientation' 'itemAccessor')
    HierarchyLayoutManager ('length' 'xStep' 'yStep' 'accessor')
      BinaryTreeLayoutManager ()
        LeafLinearTreeLayoutManager ()
          TRTreeLayoutManager ('sequenceView')
            TreeLayoutManager ()
              IndentedListLayoutManager ()
                IndentedTreeLayoutManager ('list')
NetworkLayoutManager ()
TimeSequenceLayoutManager ('timeScale' 'timeOffset')
GroupingLayoutManager ('levelScale')
PitchTimeLayoutManager ('pitchScale' 'pitchOffset')
  CMNLayoutManager ('stepArray' 'staffTop')
  MixLayoutManager ()
  PositionTimeLayoutManager ()

```

Figure 8: Signal Editors and Browsers

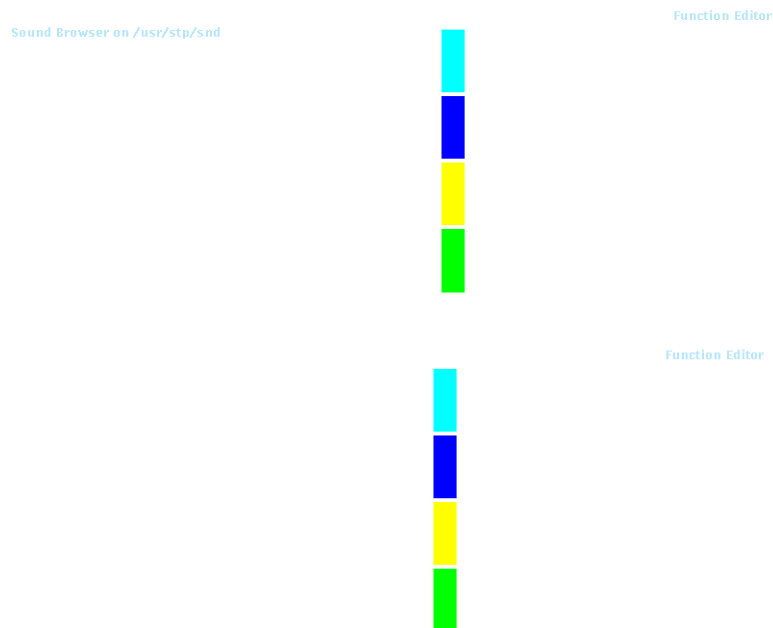
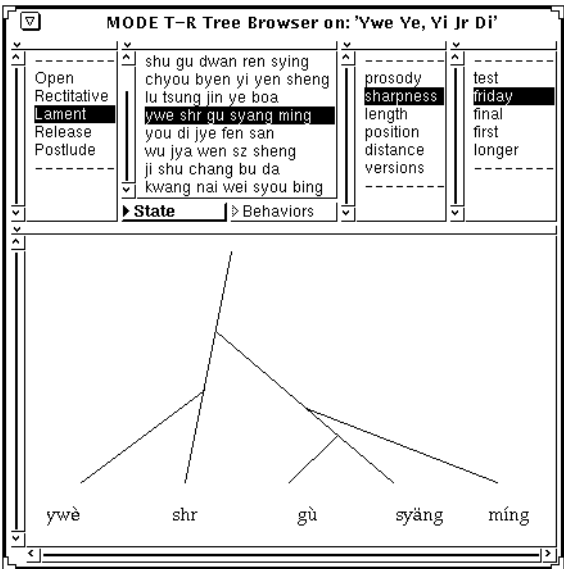


Figure 9: Event List Editors

Pitch-Time Editor

Hauer-Steffens Event List View

Figure 10: Exemplary Hierarchy Layouts



Display List Editor

Display List Editor

Figure 11: Sound File Mix Editor

Mix View on: M2.d4b.sc