```
[an error occurred while processing this directive]
```

Lisp as a second language,composing programs and music.

Peter Desain and Henkjan Honing

Chapter III Object-Oriented Style II

Draft mars 11, 1997

1. Draw
2. Time dependent transformations
3. Searching through part-of structures while maintaining onset times
4. Mapping and onset times
5. Before and after methods
6. Drawing and the window system
7. Changing structure
8. Maintaining graphical consistency
9. Applying transformations to parts of musical objects
10. Introducing multi-methods
11. A window with a selection
12. Multiple inheritance
13. Binding our own programs to existing menu commands
14. Binding commands to our own menu and control keys
15. multi-methods, method combination and multiple inheritance combined
16. Showing structure
17. Making selections visible, around method
18. Making musical objects mouse sensitive
19. Definitions made
20. Literature references made
21. Glossary references made

Draw

As an example of the use of `mapc- elements- onsets`, and to gain an easy way to inspect the results of our transformations, we will write a graphical program to draw a piano-roll notation of a musical structure using this iterator. We will assume that a graphical primitive `draw- rectangle` exists that draws a rectangle, given a list of the left, top, right and bottom sides. The window on which the pianoroll is drawn is created by the `make- piano- roll- window` function This window takes care of the drawing of the backdrop of the piano roll (the grid lines and the labels on the axes). **[to do: hier een referentie naar een piano-roll project maken ?]**

```
(defmethod draw (object &rest args)
"Draw a graphical piano-roll representation of a compound musical object"
(let ((window (apply #'make-piano-roll-window args)))
(draw-musical-object object 0 window)
window))

(defmethod draw-musical-object ((object compound-musical-object) onset window)
"Draw a graphical piano-roll representation of a compound musical object"
(mapc-elements-onsets object onset #'draw-musical-object window))

(defmethod draw-musical-object ((object note) onset window)
"Draw a graphical representation of a note (a rectangle)"
(draw-rectangle window (boundary object onset window)))

(defmethod draw-musical-object ((object pause) onset window)
"Draw nothing"
(declare (ignore onset window)))

(defmethod boundary ((object note) onset window)
"Return a list of the sides of a graphical representation of a note"
(declare (ignore window))
(list onset ; left
(+ (pitch object) .5) ; top
(offset object onset) ; right
(- (pitch object) .5))) ; bottom
```

As you can see, drawing a compound musical object amounts to drawing all its elements at the proper position, drawing a note entails drawing a rectangle. Rests are ignored for the moment. This was in fact the program used to draw Figure 1, Figure 2 and Figure 3, as the reader who is using the software that comes with this book can check. Only the grid and the labels on the axes were added. Isn't it surprising how such a simple program can achieve this quite complex task?

```
(draw (example)) =>
```

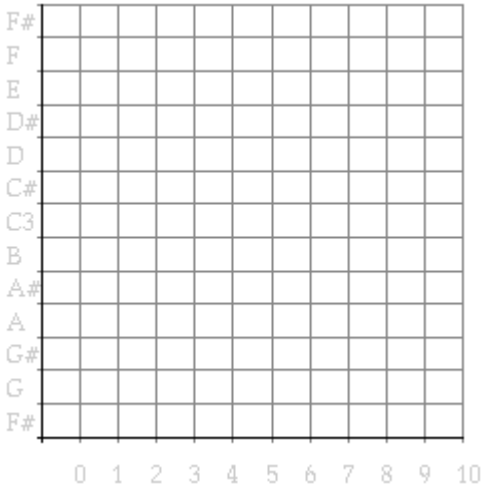

Figure 1. Piano-roll notation of the example.

```
(draw (original-and-transform
(example)
(pause :duration 1)
#'transpose 2)) =>
```

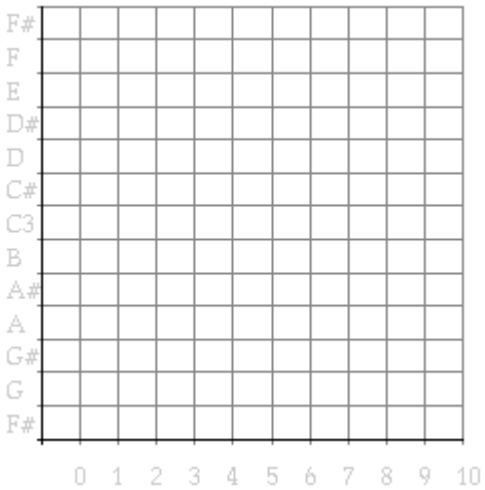

Figure 2. The example and its transposition over a whole tone.

```
(draw (original-and-transform
(example)
```

```
(pause :duration 1)

#'mirror "C3")) =>
```

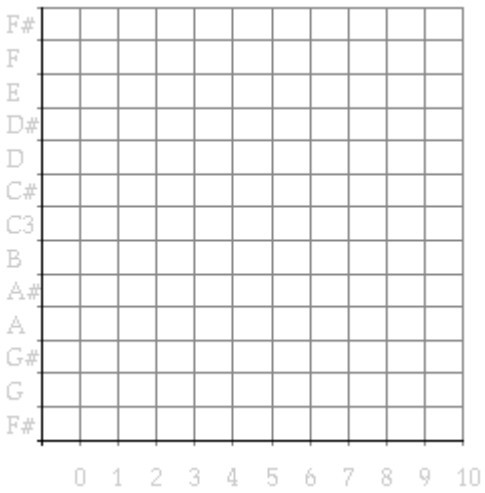

Figure 3. The example and an inversion around middle-c.

In Figure 4 the call hierarchy is given for the drawing program. It illustrates which functions (methods) call which others. The mapper itself is left out. The direction of the arrows indicate the “calls” or “invokes” relation, gray arrows mean that the call is made indirectly, through a functional argument that was passed to e.g. a mapper. **[to do: tree draw zo maken dat ook grotere cycles getekend kunnen worden, dan kan hier de mapper zelf ook in het diagram]**

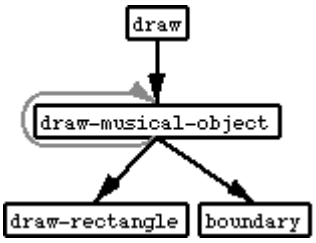

Figure 4. Flow of control in the draw program.

Time dependent transformations

Because `mapc- elements- onsets` neatly administers the onset time of each musical object it has become easy to define time dependent transformations as well. As example let’s program a gradual tempo change, using a general time-time map.[1] The `map- time` method is given a musical object, its onset and a function from time to time. It calculates new durations by applying this function to the onset and offset

times of all basic musical objects, and updates the duration slots accordingly. The `make- tempo- change` function is given an initial and a final tempo and the duration of the tempo change. It calculates the time-warp function.

```
(defmethod map-time ((object compound-musical-object) onset time-map)

"Apply a time mapping to the notes of the object"

(mapc-elements-onsets object onset #'map-time time-map))


(defmethod map-time ((object basic-musical-object) onset time-map)

"Adjust the duration of a basic musical object according to the time map"

(let* ((new-onset (funcall time-map onset))

(new-offset (funcall time-map (offset object onset))))

(setf (duration object)(- new-offset new-onset))))
```

[to do: funtion van score naar perf tijd, dan tempo factoren 1/x. laten we er een mooi sundberg ritard van maken]

```
(defun make-tempo-change (duration begin end)

"Return a time-warp function based on begin and end tempo factors"

#'(lambda (time)

(+ (* begin time)

(* (/ (- end begin)

(* 2 duration))

(square time)))))


(defun square (x)

"Return the square of a number"

(* x x))


(draw (map-time (repeat (example) 3 (pause :duration 1))

0

(make-tempo-change 14 1 .33))) =>
```

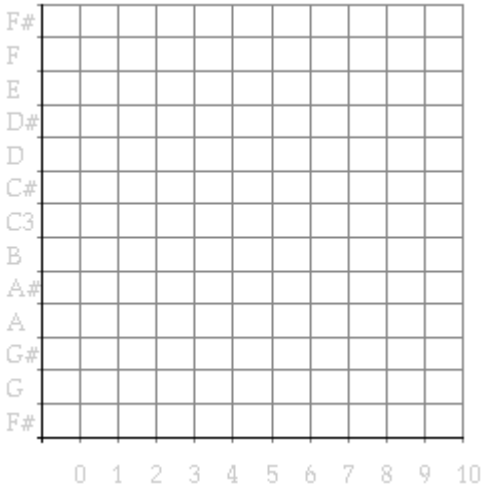

Figure 5. A gradual tempo change.

Searching through part-of structures while maintaining onset times

A refinement of our `find- element` function resembles the way in which `mapc- elements` was extended into `mapc- elements- onsets`. It maintains the time of onset and expects the predicate to have at least one extra argument: the onset of the object that it is testing.

```
(defmethod find-element-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing elements and their onset"

(loop for element in (elements object)

for onset in (onsets object onset)

thereis (apply predicate element onset args)))


(find-element-onset (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

0

#'(lambda(object onset)(= onset 1)))

->

t
```

The above example searches through the elements of a sequential musical object and returns `t` when it encounters an element starting at time 1. Note the use of `thereis` clause in the body of `find- element- onset`. This does not just return `t` when the predicate yields a non-`nil` value, it returns the value returned by the predicate (just like `and`, when all its arguments are non-`nil`, returns the value of its last argument). This enables us to use not-so boolean predicates that return something meaningful, e.g., the object which satified it. In the example below the first note which starts at 0 is returned. This approach makes extra demands to the set of predicates for musical objects, and the ways in which they can be combined. However, it lifts the need for specialized iterators.

```
(find-element-onset (sequential (note :duration 1 :pitch "C3")

(note :duration 2 :pitch "D#3")

(note :duration 1 :pitch "C#3"))

0

#'(lambda(object onset)

(and (note? object) (= onset 1) object))

)

->

(note :duration 2 :pitch "D#3")
```

Recursively calling `find- element- onset` yields again a refinement of a more general search function that can search through all levels of a musical object using a subtle mutual recursion of `find- element` and `find- musical- object- onset`, passing each other through the control structure by way of functional arguments.

```
(defmethod find-musical-object-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing sub-objects and their onset"

(or (apply predicate object onset args)

(apply #'find-element-onset object onset

#'find-musical-object-onset predicate args)))


(defmethod find-musical-object-onset ((object basic-musical-object)

onset predicate &rest args)

"Return predicate's result, when satisfied for this object at its onset"

(apply predicate object onset args))
```

To ‘read’ the code presented above: `find- musical- object- onset`, first tests the predicate and returns

its value when non-`nil`. When it is `nil`, and the object under consideration is compound, it calls find-musical-object on itself (passing it aguments as well) to have itself applied to all elements of the object. Note how the use of `&rest` arguments in the definition of both `find- musical- object` and `find- musical- object- onset` allows for the passing of extra parameters that remain invariant. This makes the use of closures to capture the value of those parameters in the functions themselves unneccesary. Note how the predicate itself is shifted in and out of the role of extra argument at the different levels of the recursion: for `find- musical- object- onset` it is a main parameter, but when passed on to `find- element- onset` it becomes an extra one, with `find- musical- object- onset` taking the role of the predicate again. When `find- element- onset` in turn calls `find- musical- object- onset`, it passes its extra parameters, with the predicate in the first place, to `find- musical- object- onset` which thus receives the predicate as a main parameter.

To test our search function we compose a predicate which will return its argument whenever it is applied to a note, its onset, and a time at which the note would be sounding. Because we are composing the `sounding- at?` predicate of three arguments (the object, its onset and a time) with simple ones of only one argument, we need to wrap those into a function that can be given more arguments, but ignores all abut the first.

[to do: use-only-first-arg-wrap uit functional, of ref naar S K combinators ?]

```
(defun use-first-arg (function)

"Wrap the function such that it can be given spurious arguments"

#'(lambda(&rest args)

(funcall function (first args))))


(find-musical-object-onset (example)

0

(compose-and (use-first-arg #'note?)

#'sounding-at?

(use-first-arg #'identity))

3)

->

(note :duration 2 :pitch "D#3")
```

Please observe that the above search function does not really represent an object oriented style at all: it makes excessive use of functional arguments. It should be kept in mind that the method could have been written in a truely object oriented way with distributed control over the recursion, as we did in the `draw- musical- object` method. However, the solution presented above is quite elegant and we will use this approach a lot. [2]

Mapping and onset times

Of course our `map- elements` should be extended to be able to maintain the onset times of the objects to which the operation is applied as well, just like `mapc- elements` was.

```
(defmethod map-elements-onsets ((object compound-musical-object)

onset operation &rest args)

"Return the results of the operation applied to each element and its onset"

(loop for element in (elements object)

for element-onset in (onsets object onset)

collect (apply operation element element-onset args)))
```

To illustrate the use of this iterator, let us write a method that allows us to collect the set of pitches that is being used in a certain time interval in a musical piece. The function to be used when combining the results of the recursive calls on each element of a compound musical object is `set- union`, which has to be defined by us because Common Lisp only supplies the function to calculate the union of two sets.

```
(defun set-union (&rest args)

"Return the union of any number of sets"

(reduce #'union args))


(defmethod sounding-pitches ((object compound-musical-object) onset from to)

"Return a list of all pitches that occur in time interval [from, to]"

(apply #'set-union (map-elements-onsets object onset #'sounding-pitches from to)))


(defmethod sounding-pitches ((object note) onset from to)

"Return a list of the pitch of the note, when sounding in interval [from, to]"

(when (sounding-during? object onset from to)

(list (pitch object))))


(defmethod collect-sounding-pitches ((object pause) onset from to)

"Return the empty list representing silence"

nil)


(mapcar #'midi-number-to-pitch-name

(sounding-pitches (example) 0

2 4))
```

```
->

("C#3" "D#3" "G2")
```

Before and after methods

To illustrate how behavior defined by means of methods can be extended further, let us modify the piano roll drawing by filling each note rectangle with a dark shade. Here again, we would like to add a piece of code instead of rewriting the `draw- musical- object` method. In CLOS, methods come in different qualities (we have only used the *primary methods* unto now). And when more methods are applicable, they are combined by the system using the so called *method combination* procedures. In our case, the added behavior can operate by *side effect*, and the extra work can be put in a so called *before method* or *after method*. We will assume that a primitive `paint- rectangle` exists to fill a rectangle with a shade and write a before method for `draw- musical- object`. The character of this methods is indicated with the *method qualifier* `:before` that appears between the name and the argument list.

```
(defmethod draw-musical-object :before ((object note) onset window)

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window (boundary object onset window) :light-gray))
```

When a `draw- musical- object` call is to be executed, the system finds all *applicable methods* depending on the class of object that the method is applied to and combines them into a whole, the so called *effective method*. In *standard method combination* all before and after methods and the *primary method* are assembled in their proper order. Any behavior that can be conceptualized as a side effect to take place before or after some main behavior that was already defined, can be added in this way. The use of *declarative method combination* frees the programmer from writing code that searches for methods and calls them in the correct order.

```
(draw (example)) =>
```

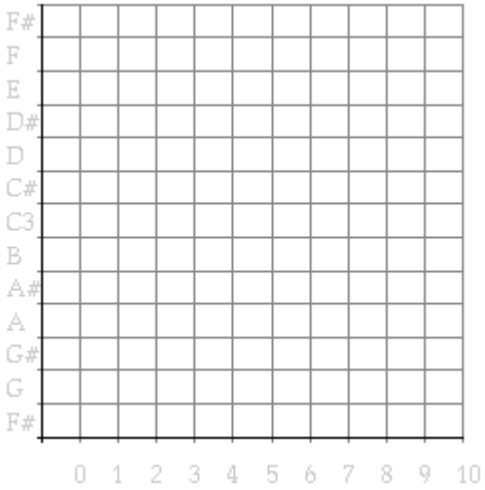

Figure 6. Shaded piano-roll notation of example.

Note that the source code of the main primary method is not changed - it needs not even be available. Thus also predefined behavior, e.g., of other authors ,or of the Lisp system itself, can be modified or elaborated. The tree diagram in Figure 7 depicts where the different `draw- musical- object` methods are defined, and which will run in the different situations. `draw- musical- object` has its own definition for a note, a rest and any compound musical object, while one before method exists to do the shading, which is specialized for the class `note`, and will only run before a note is drawn by its primary method.

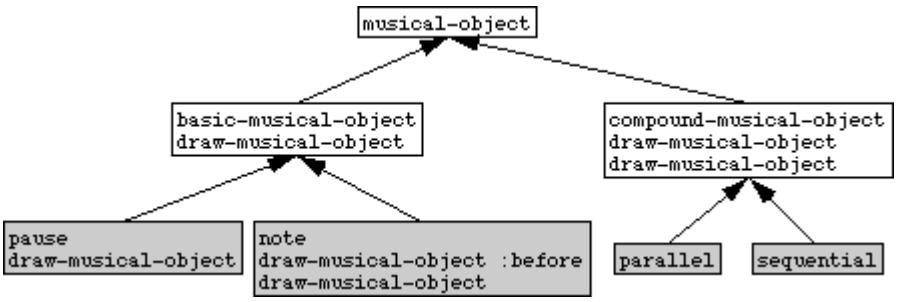

Figure 7. The draw-musical-object method definitions and musical object classes.

[to do: make the methods look different in a different font then the classes (how?)]

Looking for possible optimization of our code, we see that both `draw- musical- object` methods for notes call the `boundary` function, they are repeated below.

```
(defmethod draw-musical-object :before ((object note) onset window)

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window (boundary object onset window) :light-gray))


(defmethod draw-musical-object ((object note) onset window)

"Draw a graphical representation of a note (a rectangle)"

(draw-rectangle window (boundary object onset window)))
```

Thus, because the before and the primary method cannot communicate information, the `boundary` method is wastefully called twice for each note, once in the before method to draw a shaded rectangle, and once in the primary method to draw the boundaries around it. By defining an auxiliary function (`draw- boundary`), and moving the code that needs the boundary from `draw- musical- object` (the shared code in both functions) to this new function, this duplicated calculation can be prevented. The `draw- boundary` embodies thus the action: draw a note given its boundary box.

```
(defmethod draw-musical-object ((object note) onset window)
"Draw a graphical representation of a note (a rectangle)"
(draw-boundary object (boundary object onset window) window))

(defmethod draw-boundary ((object note) boundary window)
"Draw a box with a specified boundary (left top right bottom)"
(draw-rectangle window boundary))

(defmethod draw-boundary :before ((object note) boundary window)
"Fill a graphical representation of a note with a gray color"
(paint-rectangle window boundary :light-gray))
```

Before and after methods form a powerfull way to slice-up behaviour in tiny morsels that can be added, managed and grouped much better than bulky code that has to be rewritten for each modification. The possibility to define more methods specialized to the same class with different qualifiers, and in a declarative way state the order in which they must be called, is one of the features that set CLOS aside from many other object oriented languages.

Drawing and the window system

In the Macintosh window system, drawing directly on a window yield a result that is somewhat volatile because whenever, e.g., in response to a user action, such a window is covered by another one and subsequently uncovered, part of the drawing is gone. In this window system it is the obligation of the window itself (not of the covering window) to maintain a consistent graphical image. In order to allow a window to redraw its contents after such operations the window system calls the `view- draw- contents` method on the window that was just uncovered. We can define a after method for `view- draw- contents` ourselves, to redraw the pianoroll. For that to work the musical object needs to be stored in the window, such that the method (which is called with the window as its sole argument) can get hold of it whenever a redraw is needed. To maintain modularity we define a new window class, based on the piano roll window, that realises this feature. For easy generalization we give it an extra slot with the function that actually does the drawing - such that it will be usable with many drawing programs.

```
(defclass piano-roll-contents-draw-window (piano-roll-window)
((object :accessor object :initarg :object :initform nil)
(draw-function :accessor draw-function :initarg :draw-function :initform nil))
(:documentation "A window for drawing and re-drawing musical objects"))

(defmethod view-draw-contents :after ((window piano-roll-contents-draw-window))
"(Re)draw the stored object as a pianoroll"
```

```
(when (object window)

(funcall (draw-function window) (object window) window)))
```

[to do: uitleg when object conditie]

At window creation time, we now have to supply a musical object, a drawing function. Any drawing and redrawing will be initiated by the window system, as appropriate. Even initial drawing, just after the window is created (like in our original `draw` function), is unnessecary because the window system itself will call `view- draw- contents` on any newly created windows. In this way our drawings are better integrated in the workings of the Macintosh window environment. Again, to allow for future extensions, we add one extra level of inheritance and define a class `draw- window` based on the just defined `piano-roll- contents- draw- window` class, to allow for future window behavior.

```
(defclass draw-window (piano-roll-contents-draw-window)

()

(:documentation "A simple pianoroll window class"))
```

The `draw` method now just has to make an instance of such a window, and store a musical object and a draw function in the appropriate slots (instead of also drawing the object). The invocation of the drawing action itself is taken care of by the window system. Furthermore, to allow the passing of other initialization arguments to the window (such as its title or size), we make the `draw` method accept any extra argument. They are all passed as keyword arguments in a list to the `make- instance` function. In this way, known and future window classes will work with specific initialization arguments not known right now.

```
(defmethod draw (object &rest args &key (window-class 'draw-window) &allow-other-keys)

"Draw a graphical piano-roll representation of a compound musical object"

(apply #'make-instance window-class

:object object

:draw-function #'(lambda(object window)

(draw-musical-object object 0 window))

args))
```

In summary, covering and uncovering (part of) a window will shedule (asynchronously) a `view- draw- contents` command. Whenever that command is executed, first the `view- draw- contents` method of the piano roll window itself is run, which draws the grid. Then our own after method is run, which calls the `draw- musical- object` method on the object stored in the window slot. This finally results in the musical object being redrawn on the screen. To optimize this process, the window system takes care to set the so called view port, the area of the screen that is actually updated in drawing, to only cover the region that has been affected and needs to be redrawn. The drawing primitives will only perform their work in this area.

Changing structure

[to do: deze sectie kan evt naar voren, als voorbeeld van OO-style gedistribueerde recursie]

The main operation in our transformations was applied to notes only, the structure of the musical object remained invariant. This will no longer be the case in the next example: a `retrograde` transformation for musical objects that reverses each sequential (sub-)structure. This transformation destructively modifies only the sequential objects encountered, the others are visited but left untouched. The result of applying this transform is illustrated in Figure 8.

```
(defmethod retrograde ((object parallel))

"Reverse all components of a musical object in time, recursively"

(mapc-elements object #'retrograde)

object)


(defmethod retrograde ((object basic-musical-object))

"Do nothing"

object)


(defmethod retrograde ((object sequential))

"Reverse all components of the object in time"

(setf (elements object)

(reverse (elements object)))

object)


(draw (original-and-transform

(example)

(pause :duration 1)

#'retrograde)) =>
```

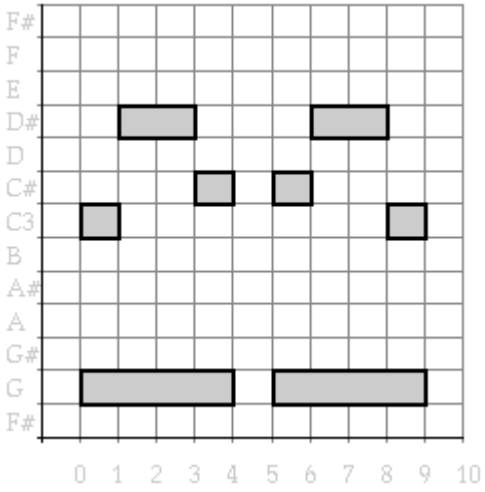

Figure 8. The example and a reversal.

Maintaining graphical consistency

Since musical objects can be stored in windows, we could direct our transformations of musical objects to these windows themselves (accessing them for the moment by title string). The `window- operation` function fist chech whether it is given a window, before it applies the operation to the musical object stored in it, to allow `window- operation` to be called on functions that search a window (like `find-window`) but may return `nil`.

```
(defun window-operation (window operation &rest args)

"Apply the operation to a musical object stored in a window"

(when window

(apply operation (object window) args))

window)
```

[to do: hidden hier vanaf hier het eigen draw window gebruiken voor figuur generatie]

```
(draw (example) :window-title "example musical object")

(window-operation (find-window "example musical object") #'retrograde) =>
```

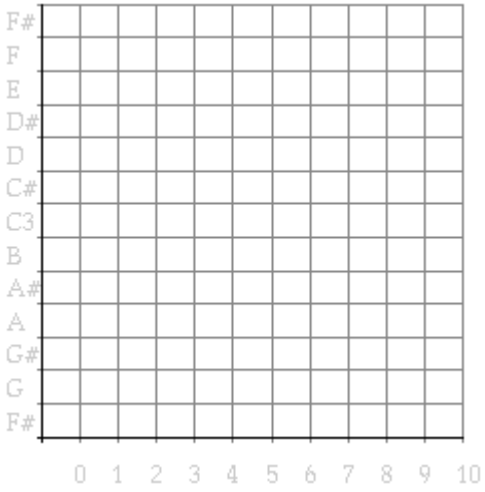

Figure 9. Unconsistency

[to do: `window- operation` method op `piano- roll contents- draw window` ? En slot-name object -> musical-object?]

However, applying a destructive transformation to an object that is displayed on a window yields an inconsistency between the graphical representation and the musical object, e.g. the chance made by the `retrograde` transformation above is not yet shown in the window. Covering the window, by selecting another one or moving another one on top of it, and uncovering it, may force a redraw and reconsitute the consistency, but this is not a very elegant solution. **[to do: recording mixin zo maken dat dit werkt voor de lezer]**It is posible to force a redraw ourselves by calling `invalidate- view`. This method forces the window system to assynchronously (at some time in the future) call `view- draw- contents`, which will redo the drawing. We will use `invalidate- view` to be able to do destructive transforms on objects displayed in a window while maintaining consitency between object and its graphical presentation. The extra argument `t` to `invalidate- view` signals that the window needs to be wiped before any redrawing can be done.

```
(defun window-operation (window operation &rest args)

"Apply the operation to a musical object stored in a window"

(when window

(apply operation (object window) args)

(invalidate-view window t)

window))
```

As long as we call all transforms on musical objects in windows by way of `window- operation` consistency is guaranteed. Instead of using `find- window` we will now assume that the window in which the object of interest is displayed is the front-most of the drawing windows. We can write a `front- draw- window` function that gets hold of that window, bringing it to the front as well (if it wasn't already). Based

on that function it is not difficult to write the `front- window- operation` function which invokes an operation on an object stored in the front most `draw- window`.

```
(defun front-draw-window ()
"Return the frontmost drawing window, bringing it to the front as well"
(let ((window (front-window :class 'draw-window)))
(when window
(window-select window)
window)))

(defun do-front-window (action &rest args)
; onhandig te onthouden, of meer gebruiken hieronder
(let ((window (front-draw-window)))
(when window
(apply action window args)
window)))

(defun front-window-operation (operation &rest args)
"Apply an operation to a musical object stored in the frontmost draw window"
(apply #'window-operation (front-draw-window)
operation
args))
```

Now we can initiate transformations on objects visible in windows as we like it, by bringing the window to the front by a mouse click or using the windows menu. Opperations are issued on the front draw window and the system automatically updates the graphgical representation.

```
(draw (example) :window-titel "example window 1")
(draw (example) :window-titel "example window 2")
; bring one of the window to the front
(front-window-operation #'mirror "C3")
; or the other one
(front-window-operation #'transpose 3)
; and maybe the first again
```

```
(front-window-operation #'retrograde)
```

The operation may, of course, not only do transformations, other actions like, e.g., playing the musical object visible in a window are useful as well.

```
;bring one of the windows with a musical object to the front

(front-window-operation #'play)

(front-window-operation #'play :tempo 240)

(front-window-operation #'retrograde)

(front-window-operation #'play)
```

Applying transformations to parts of musical objects

We have only applied our transformations to complete musical objects, globally, from the outside so to say. However, there is no reason why they shouldn't be used on objects nested (deeply) inside other objects. In Figure 10 we select a part of the example, and apply a transpose operation to it. Our `find-musical- object- onset` function, which searches through an object, comes in handy here.

```
(draw (example)) => 12.A. draw the example

(front-window-operation

#'(lambda(object)

(transpose

(find-musical-object-onset object

0

#'(lambda(part onset)

(and (= onset 1) part)))

3))) => 12.B. Transpose the part which starts at time 1
```

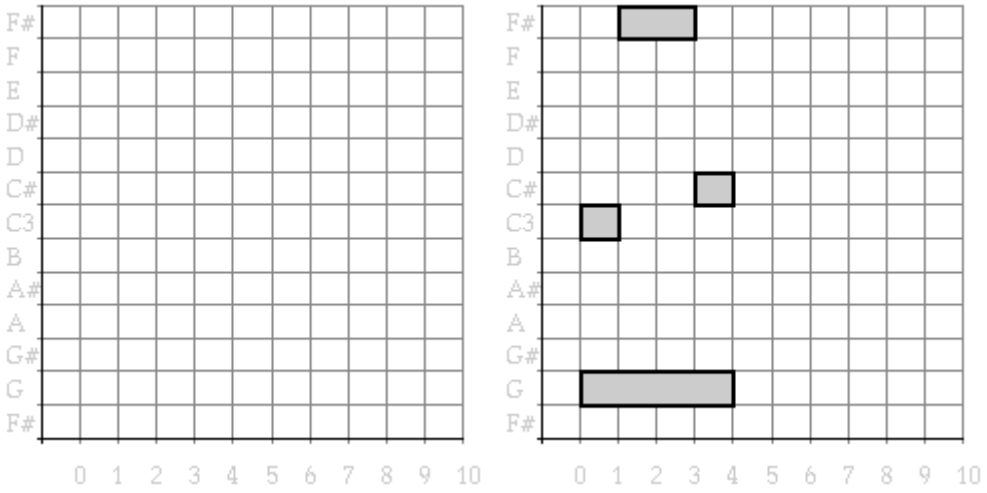

Figure 12. Transposition of the part of the example which starts at time 1

Introducing multi-methods

The method definitions we have written so far all have one argument specialized to a specific class, and in deciding which method is (or which methods are) applicable in a certain case, the method definition specialized to the most specific class of this argument is chosen. This situation, in which a method dispatches on the type of one argument, is still compatible with older object oriented paradigms develloped in LISP and with the *message passing style* of e.g. Smalltalk. In this style, objects handle so called messages (the name of the method plus the other arguments) sent to it, with the message handling selected according to its type. But in some cases, the forced assymetry of deciding which object to sent the message to is quite unnatural. Should we, e.g., write a draw method specialized to different kinds of windows that explicitly inspect the type of the item to be drawn and call the proper procedure, or should we write draw methods specialized to different kinds of grapical items which inspect the type of window to draw on, and call the proper procedures? Or consider an even more symmetrical example, the design of a equality test on musical objects: should this method be specialized to the class of the first object, in the body do a check on the class of the second one, and call the proper procedures? Luckily the answer to these question is: neither. It was a beautiful insight of the designers of CLOS that message passing as view of object oriented programming is not general enough, and that it much better to think of methods as identically named functions with differently typed argument lists, of which a fitting one is selected for each call. Thus in the cases mentioned as examples here it is much better to define methods based on the classes of both their arguments.This dispatching on more arguments, by so called *multi-methods*, boosts the expressivity of CLOS enormously. As a first example, the predicate that tests for equality of musical objects is given below. For each two same-typed objects there is a method defined which checks for equality of elements or attributes. One general method for any two musical objects will catch all other cases, when the predicate is applied to objects that are not of the same class, it can return `nil` immediately.

[to do: Remark: `same?` can't be implemented with our recursor map-elements-onset, since its need two musical objects as argumetn (instead of one). We therefore use athe distributed form of recursion here instead.]

```
(defmethod same? ((object1 sequential)(object2 sequential))
```

```
"Are two sequential musical objects equal?"

(same-list (elements object1)(elements object2) :test #'same?))


(defmethod same? ((object1 parallel)(object2 parallel))

"Are two two parallel musical objects equal?"

(same-set (elements object1)(elements object2) :test #'same?))


(defmethod same? ((object1 note)(object2 note))

"Are two notes equal?"

(and (= (pitch object1) (pitch object2))

(= (duration object1) (duration object2))))


(defmethod same? ((object1 pause)(object2 pause))

"Are two rests are equal?"

(= (duration object1) (duration object2)))


(defmethod same? ((object1 musical-object)(object2 musical-object))

"Return nil as default case: two musical objects of different type are unequal"

nil)
```

For two sequential musical objects to be equal, objects from both lists of elements have to be pairwise equal. For two parallel musical objects equality must hold for the elements of each object, but with the list of elements considered as unordered set. For notes and rests all attributes have to match. For all other combinations `nil` should be returned. Because CLOS always selects *the most specific method* to run, in case where more are applicable, the method specialized for, e.g., two notes is run whenever two notes are supplied as argument, even though another method, specialized to two musical objects, is applicable as well: the `note` class is more specific than the `musical- object` class because the first inherits from the second. Thus in those cases the method specialized for notes effectivelly shadows the less specific one.[3]The auxillary functions that test for equality of lists and sets are written in true Common Lisp style, with an extra argument that will function as the equality test to be used for their items. Thus they thus embody only control structure of visitng elements in lists and sets while testing for equality, and they can be supplied with the same? predicate in case the test is for equality of lists or sets of musical objects, as is done above.

```
(defun same-list (list1 list2 &key (test #'eql))

"Do two lists contain the same items in the same order"

(loop for item1 in list1

for item2 in list2
```

```
always (funcall test item1 item2)))


(defun same-set (set1 set2 &key (test #'eql))

"Do two list contain the same items in any order"

(and (= (length set1)(length set2))

(loop for item1 in set1

always (member item1 set2 :test test))))
```

The `same?` predicate can be used as general equality test, allowing the many Common Lisp fuctions that can handle a test predicate to be applied directly to musical objects. In the example below we search for the place of one note in the elements slot of a musical object using the Common Lisp function `position`.

```
(position (note :duration 4 :pitch "G2")

(elements (example))

:test #'same?)

->

1
```

A window with a selection

Since we have created the possibility to do an operation on a musical object stored (and displayed) in a window, and we have written transformations that can operate on parts of a musical object, it seems a good idea to be able to select such a part and store it in a window, to be able to act on it without having to search for it every time. As a first solution we will add this behavior to draw-window: it will get an extra slot to store a selection, a part of its musical object, and some ways to update this selection. As acces functions to this facility we define a `select` method to store a specified part as selection, a `select- all` method which indicates that the whole musical object is selected, a `deselect` method to reset the selection to contain noting. Because we forsee that a change of the selected part may need to trigger some more actions, e.g., when we will have create a distict visible appearance of a selected object, all updates to the selction are channeled through a call to `select`. This narrow interface in than a nice hook to provide for future extensions. because in CLOS we cannot enforce the use of such an interface, and prohibit e.g. direct updates of the `selection` slot, a CLOS programmer needs some extra discipline to design and documents these design decisions. However, later we will see how in the meta-object protocol we can program such encapsulation ourselves. **[to do: meta object protocol object encapsulation interface]**

```
(defclass draw-window (piano-roll-contents-draw-window)

((selection :accessor selection :initarg :selection :initform nil))

(:documentation "Our new pianoroll window class"))


(defmethod select ((window draw-window) object)
```

```
"Select an object in the window"

(setf (selection window) object))


(defmethod select-all ((window draw-window))

"Select the whole musical object in the window"

(select window (object window)))


(defmethod deselect ((window draw-window))

"Set the window selection to nil"

(select window nil))
```

The next step is to add support for acting on selections. The `selection- operation` function closely resembles `window- operation`, but now the operation is don on the selection, not the whole musical object. It has to be taken into account that a selection may be empty, in that case no operation needs to be carried out.

```
(defun selection-operation (window operation &rest args)

"Apply the operation to a musical object stored in a window"

(when (and window (selection window))

(apply operation (selection window) args)

(invalidate-view window t))

window)
```

Adding some more auxilary functions will make it convenient to act on a selection in the front window.

```
(defun front-selection-operation (operation &rest args)

"Apply an operation to the selection stored in the frontmost draw window"

(apply #'selection-operation (front-draw-window)

operation

args))


(defmethod front-select-all ()

"Select the complete musical object in the front window"

(do-front-window #'select-all))
```

```
(defmethod front-deselect ()
"Deselect everything in the front window"
(do-front-window #'deselect))

(defmethod front-select (selector &rest args)
"Apply a function which will state what to select in the front window"
(do-front-window
#'(lambda(window)
(select window
(apply #'find-musical-object-onset (object window) 0 selector args)))))
```

[to do: ; eruit ? gebruikt allen voor debug]

```
(defmethod front-selection ()
"Return the selection of the front window"
(do-front-window #'selection))
```

These facilities can now be tested:

```
(draw (example)) => 16.A. draw the example
(front-select-all)
(front-selection-operation #'transpose 3) => 16.B. transposition of the full selection
(front-select (compose-and (use-first-arg #'sequential?)
(use-first-arg #'identity)))
(front-selection-operation #'mirror 60) => 16.C. reflect the sequential part
```

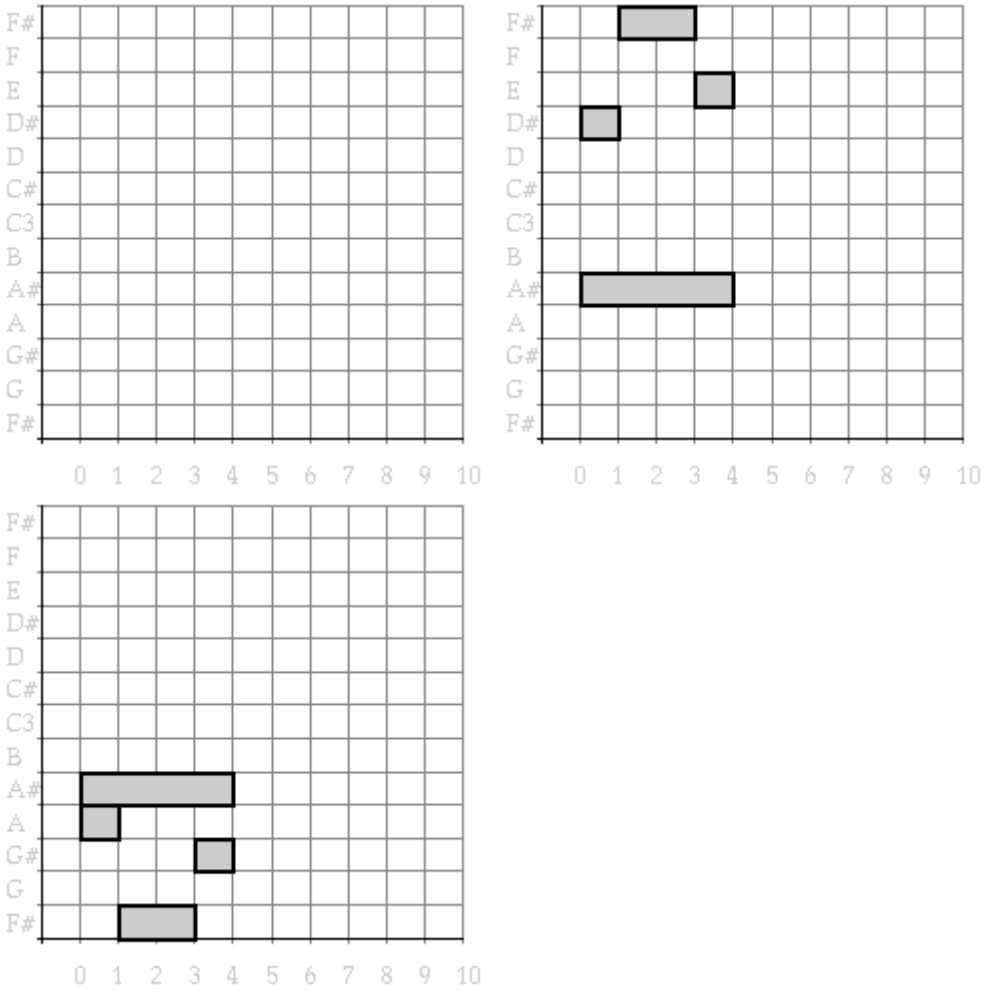

Figure 16. testing operations on a selection

[to do: boom op de kop maken voor een soort boom]

[to do: de aanroep hierachie boom hier zetten]

Multiple inheritance

Though at first sight everything seems fine, our modularity has suffered from adding these new facilities: any draw window now has the added selection behaviour and our virgin draw window is gone forever. It would have been much better had we defined a new window class to contain this behavior. Let us separate the new facilities from the main `draw- window` class.

```
(defclass selection-mixin ()

((selection :accessor selection :initarg :selection :initform nil))

(:documentation "A window mixin to add selection behavior"))
```

```
(defmethod select ((window selection-mixin) object)

"Select an object in the window"

(setf (selection window) object))


(defmethod select-all ((window selection-mixin))

"Select the whole musical object in the window"

(select window (object window)))


(defmethod deselect ((window selection-mixin))

"Set the window selection to nil"

(select window nil))
```

To create a new `draw- window` class which supports a selection we now have to inherit from both the `piano- roll- contents- draw- window` class and the `selection- mixin` class.

```
(defclass draw-window (selection-mixin piano-roll-contents-draw-window)

()

(:documentation "Our current pianoroll window class"))
```

This is the first time we make use of so-called *multiple inheritance*. This powerfull construct is not allowed in many other object oriented languages and, one has to acknowledge, it can be misused easily. The programmer has to exercise discipline in its use to keep the complexity managable. Some discipline was already used above: the reader may have noticed that the selecition mixin class does not inherit from any class, in particulary not from the piano roll or any other window class. In that sense it is not a complete class, it can not be instantiatiated to yield anything usefull. This so called *abstract class* or *mixin class* is only intended to supply a certain chunck of behavior that is to be mixed in with another class to be meaningfull. Because CLOS has no language constructs to specify that our mixin cannot be instatiated on its own[4], part of the programmers discipline is to reflect this nature in the name of the class. Of course the selection window class could have inherited from the piano roll window class. It would work, and all inheritance issues would still be well defined in CLOS, but since then the same methods and slots can be inherited via multiple routes, keeping track conceptually of the design of our system becomes a much more complex issue. [5]:footnote Again, in anticipation, we generalize more then actually needed at the moment of implementing a certain solution.

```
(defclass draw-window-mixin ()()

(:documentation "Additional behavior for draw window"))
```

```
(defclass selection-mixin (draw-window-mixin)

((selection :accessor selection :initarg :selection :initform nil))

(:documentation "Our pianoroll window class"))
```

[to do: ? tree figure of multiple inheritance (with abstract class)]

Binding our own programs to existing menu commands

The `select- all` operation may have looked familiar to you because it is defined in many other Macintosh applications as well. And part of the quality of the design of the Machintosh software lies in the ease which which users can transfer knowledge they learned in one application to other programs. In that same line it is wise to recognize that our select-all is conceptually the same as select-all commands for other applications on other windows, and that its invocation should be supported by the same means. Surprisingly this has already been taken care of: pull down the file menu while a draw window is selected. The select all command will not be grey: because we defined a `select- all` method applicable to this window, it is made accessible as menu action and given the **[to do: clover invoegen]**clover-A command key automaticaly. Try it, and think of what other standard Machintosh operations would be usefull to support.

Binding commands to our own menu and control keys

There are of course commands specific to musical objects like `play` and `transpose`, which cannot find a place in the system menus. We will create a new menu for these types of commands, define a command key for them and let them all operate on the selection made in the front window. The commands to be put in the menu and bound to command keys will, for now, have no arguments other than the window whose selection they have to be applied to.

```
(defmethod play-selection ((window draw-window))

"Play the musical object selected in the window"

(selection-operation window #'play))
```

```
(defmethod transpose-selection-up ((window draw-window))

"Transpose the musical object selected in the window up by one semitone"

(selection-operation window #'transpose 1))
```

```
(defmethod transpose-selection-down ((window draw-window))

"Transpose the musical object selected in the window down by one semitone"

(selection-operation window #'transpose -1))
```

```
(defmethod mirror-selection ((window draw-window))

"Reflect the musical object selected in the window around Middle C"
```

```
(selection-operation window #'mirror 60))


(defmethod retrograde-selection ((window draw-window))
"Reverse the musical object selected in the window"
(selection-operation window #'retrograde))


(add-music-command "Play" #'play-selection #\p)
(add-music-command "Transpose up" #'transpose-selection-up #\u)
(add-music-command "Transpose down" #'transpose-selection-down #\d)
(add-music-command "Mirror" #'mirror-selection #\m)
(add-music-command "Retrograde" #'retrograde-selection #\r)


(draw (example))
(front-select-all) => 20.A. select the whole example
(front-draw-window) => 20.B. choose the retrograde command in the menu
(front-selection-operation #'Retrograde) => 20.C. the result
```

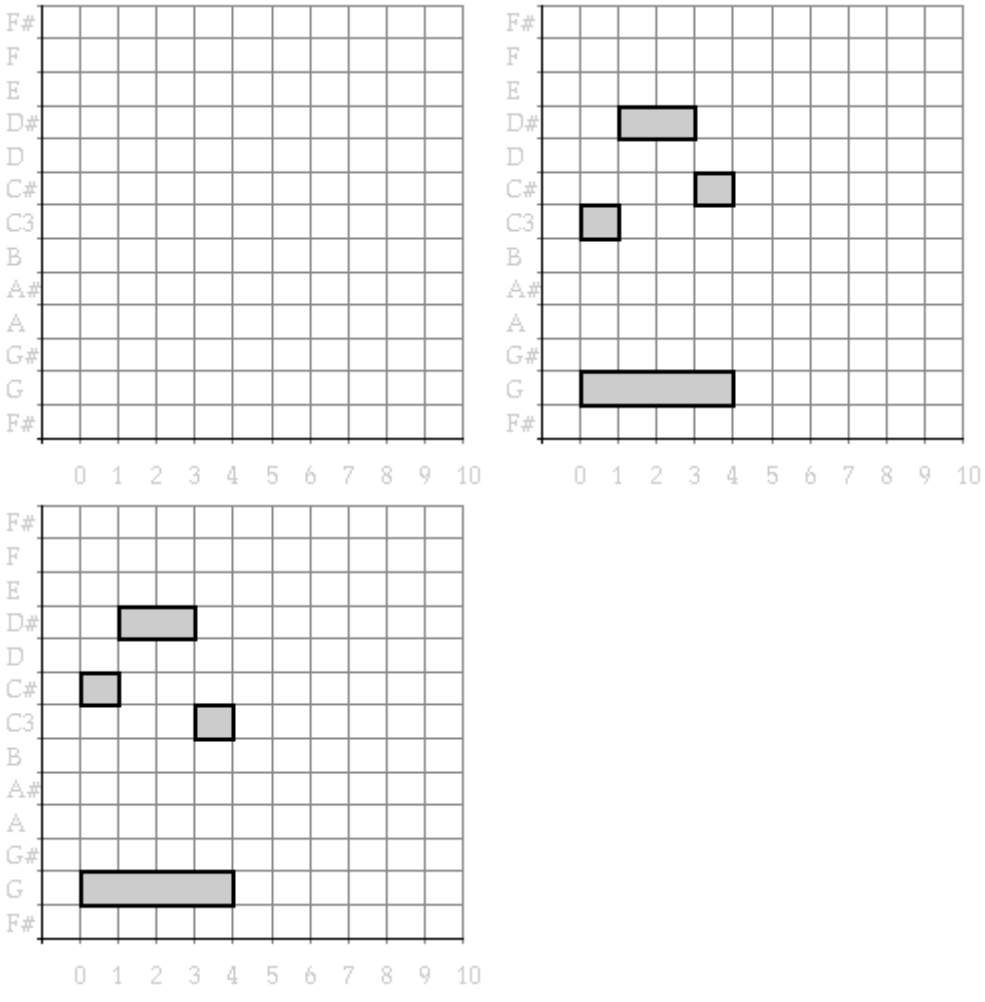

Figure 20. testing operations on a selection

[to do: plaatje van het music menu]

multi-methods, method combination and multiple inheritance combined

The reader might have noticed that the addition of shading to the drawing of notes is hard-wired into our draw-window, it cannot be switch off anymore. Were we to add the possibility of both shaded and unshaded drawing to our program in a traditional way, we would add an extra parameter (a flag), and consult it in the drawing of each note to check whether shading is desired. This would make the program bulky and less modular, especially when more drawing options would arise (and more parameters need to be added). A better, and thoroughly object oriented solution is based on the realisation that there are different classes of windows, each with its own 'look'. This entails that drawing behavior does not only depend on the kind of musical object, but on the type of window as well, which is just the kind of issue that multi methods can address elegantly. Thus shading could have been formalised as a window mixin class as well, instead of an addition to the draw window itself, and indeed it should have. The shading takes place in the `draw- boundary` :before method which takes a musical object, a boundary and a window. This multi-method only needs to be redefined for a new `shaded- mixin` class, instead of for all draw windows.

```
(defclass shaded-mixin (draw-window-mixin)

()

(:documentation "A window that shades the notes of the pianoroll"))


(fmakunbound 'draw-boundary)


(defmethod draw-boundary :before ((object note) boundary (window shaded-mixin))

"Fill a graphical representation of a note with a gray color"

(paint-rectangle window boundary :light-gray))


(defmethod draw-boundary ((object note) boundary (window draw-window))

"Draw a box with a specified boundary (left top right bottom)"

(draw-rectangle window boundary))


(defclass draw-window (shaded-mixin

selection-mixin
```

```
piano-roll-contents-draw-window)

()

(:documentation "Our (forever changing) pianoroll window class"))
```

In the example above we thus have used three main constructs of CLOS which form such a potent mixture in combination: Multiple inheritance (to have the `draw- window` class inherit from more classes, each supplying part of the behavior needed), multi-methods (to select the `draw- boundary` method on the basis of both the musical object and the window class) and declarative method combination (to supply a piece of code that has to be run before the main drawing of a boundary). Not only is the extension of our drawing program implemented completely as an addition of a small piece of code (a class and a method), but that also old behavior is not affected (old fashioned draw-windows without shading can still be build and will function in the familiar way.

Showing structure

Sequential and parallel structure is paramount in our examples, but it is not visible in the piano roll graphics. To make up for that omission we can visualize the structure of a musical object in the form of a box extending for the duration of the whole object and covering its pitch range. The first idea would be to add a before or after method to draw-musical-object for compound musical objects, to calculate the boundary and draw a box around them. This may later give rise to inefficencies, since new draw mixins may need the boundary of the compound objects as well - just like the before method in the shade mixin did for notes. We can best admit defeat, our basic draw window did not supply enough hooks in the drawing code to make it reusable enough. [6] Luckily it is fun to try to evolve the base class definitions such that many extensions can be hooked unto it. So that is what we need to do here: add a call to `draw-boundary` in the drawing of compound musical objects. And add stub methods that do nothing to calulate and draw boundaries for objects other then notes. Thus a control structure is errected in which every object is visited with a call to `draw- boundary`, which could trigger the drawing of a musical object with a given boundary, possibly implemented in a window mixin. And the calculations of the boundaries themselves are to be implemented as a method for boundary, also possibly implemented in a mixin class. Our rewrite is shown below.

```
(fmakunbound 'draw-musical-object)


(defmethod draw-musical-object ((object compound-musical-object)

onset

(window draw-window))

"Draw a graphical piano-roll representation of a compound musical object"

(draw-boundary object (boundary object onset window) window)

(mapc-elements-onsets object onset #'draw-musical-object window))


(defmethod draw-musical-object ((object basic-musical-object)

onset
```

```
(window draw-window))

"Draw a graphical representation of a basic musical object"

(draw-boundary object (boundary object onset window) window))


(defmethod draw-boundary ((object note) boundary (window draw-window))

"Draw a note with a specified boundary (left top right bottom)"

(draw-rectangle window boundary))


(defmethod draw-boundary ((object musical-object) boundary (window draw-window))

"Draw nothing"

(declare (ignore boundary))

)


(defmethod boundary ((object note) onset (window draw-window))

"Return a list of the sides of a graphical representation of a note"

(declare (ignore window))

(list onset ; left

(+ (pitch object) .5) ; top

(offset object onset) ; right

(- (pitch object) .5))) ; bottom


(defmethod boundary ((object musical-object) onset (window draw-window))

"Return nil, boundary of musical object is not known"

(declare (ignore onset))

nil)
```

The above modularity has the added advantage that the calulation of boundaries are separated from the drawing based on them. This allows, e.g., for mixing in a class to prevent the drawing of objects which lay completely outside the visible part of the window, which needs the calculation of boundaries of compound musical object but not nessary the drawing of them. But let's get back on track and design how to draw contours around sequential and parallel object, and how to calculate these countours or boundaries. The boundary method for compound musical objects has to recursively call itself on each element and combine the results by calculating the highest, rightmost, lowest and leftmost of the sides of the boundaries of its elements. Just like a graphical representation cannot easily be invented for rests, it is not easy to define boundaries for compound objects containing only rests. So we will not calculate those in the hope that later better ideas will arise.

```
(defclass combine-boundary-mixin (draw-window-mixin)

()

(:documentation "A mixin to combine boundary boxes"))


(defmethod boundary ((object compound-musical-object) onset (window combine-boundary-
mixin))

"Return a list of the sides of a box enclosing a compound musical object"

(apply #'combine-boundaries

(map-elements-onsets object onset #'boundary window)))


(defmethod combine-boundaries (&rest boundaries)

"Combine a list of boundary boxes into one box that encloses them all"

(let ((true-boundaries (remove nil boundaries)))

(when true-boundaries

(loop for (x1 y1 x2 y2) in true-boundaries

minimize x1 into left

maximize y1 into top

maximize x2 into right

minimize y2 into bottom

finally (return (list left top right bottom))))))
```

The minimization and maximization of times and pitches can be done easily using the loop macro facilities. Since the `boundary` method may return nil for certain kinds of objects (like rests), we have to carefully remove these before the list of boundaries is combined into one. [7]

Now we can define the class which actually draws the boundaries. It inherits from `combine- boundary- mixin` because it critically depends on the calculation of boundaries defined in that class. This makes the listing of the `combine- boundary- mixin` class in the list of classes of which the `draw- window` inherits superfluous: all inheritance of methods of the `combine- boundary- mixin` is taken care of via `structured- mixin`. For simplicity we ignore for the moment that one might want to properly nest the boxes graphically - enclosing boxes will be drawn over each other. The extension only amounts to a redefintion of the `draw- boundary` method that may now be applied to any musical object. We only have to prevent drawing when it is called with a `nil` boundary.

```
(defclass structured-mixin (combine-boundary-mixin)

()
```

```
(:documentation "A mixin to visualize internal structure"))


(fmakunbound 'draw-boundary)


(defmethod draw-boundary ((object musical-object) boundary (window structured-mixin))
"Draw a box with a specified boundary (left top right bottom)"
(when boundary
(draw-rectangle window boundary)))


(defclass draw-window (structured-mixin
shaded-mixin
selection-mixin
piano-roll-contents-draw-window)
()
(:documentation "The flexible pianoroll window class"))


(draw (repeat (example))) =>
```

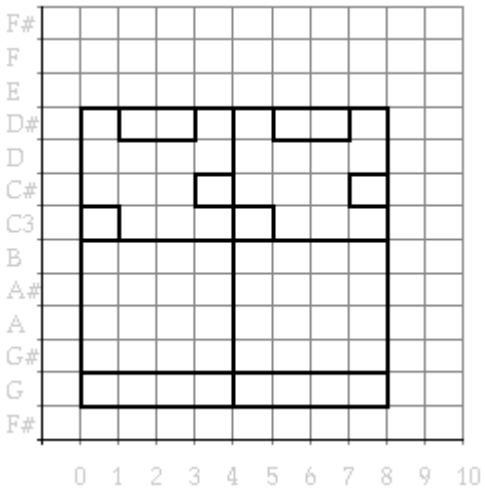

Figure 21. A musical object with its structure visualized.

The call hierarchy is given in Figure 22. We will show later how this structure can be simplified a bit.

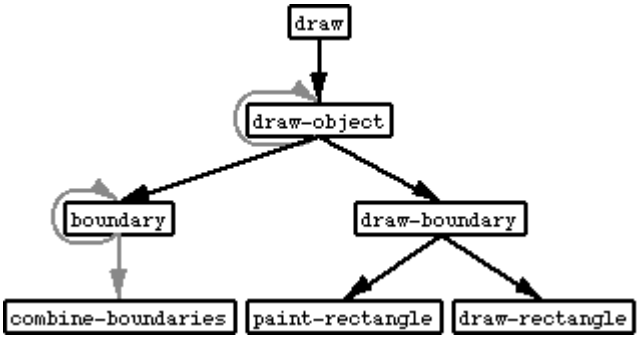

Figure 22. The complex flow of control while drawing.

[to do: check of dit klopt]

Making selections visible, around method

For our window that supports a selection now the rule that in a graphical user interface all internal state that affects the user actions should be made visible, needs to be applied: we need to show a selected (sub)object in a distinguised fashion. As always we will supply the extra behavior in the form of a window mixin. There are several possibilities to visualize a selection. We will opt here for giving the object a grey background. Thus the fact that we made the shading of notes a mixin already pays-off, we can now easily remove it from the list of classes that draw-window inherits from, and use shading for a different purpose. To make the selection appear consistent, we need to add an after select method which informs the window system that a redraw is needed. Because all updates to the selection are made via `select`, this good disipline pays of here: we can be sure that this is the only place that needs patching.

```
(defclass visible-selection-mixin (selection-mixin)

()

(:documentation "A class to support visible selections in draw windows"))


(defmethod draw-boundary :before ((object musical-object)

boundary

(window selection-mixin))

"Paint selected objects"

(when (eql object (selection window))

(paint-rectangle window boundary :light-gray)))


(defmethod select :after ((window selection-mixin) object)
```

```
(invalidate-view window t))


(defclass draw-window (structured-mixin

visible-selection-mixin

selection-mixin

piano-roll-contents-draw-window)

()

(:documentation "The pianoroll window class, selection-controlé"))
```

Note how the select after method is not specialized to musical objects: it may have to be applied to nil.

```
(draw (example)) => 26.A. draw the example

(front-select-all) => 26.B. selection of whole musical object

(front-select (compose-and (use-first-arg #'sequential?)

(use-first-arg #'identity))) => 26.C. selection of the sequential part
```

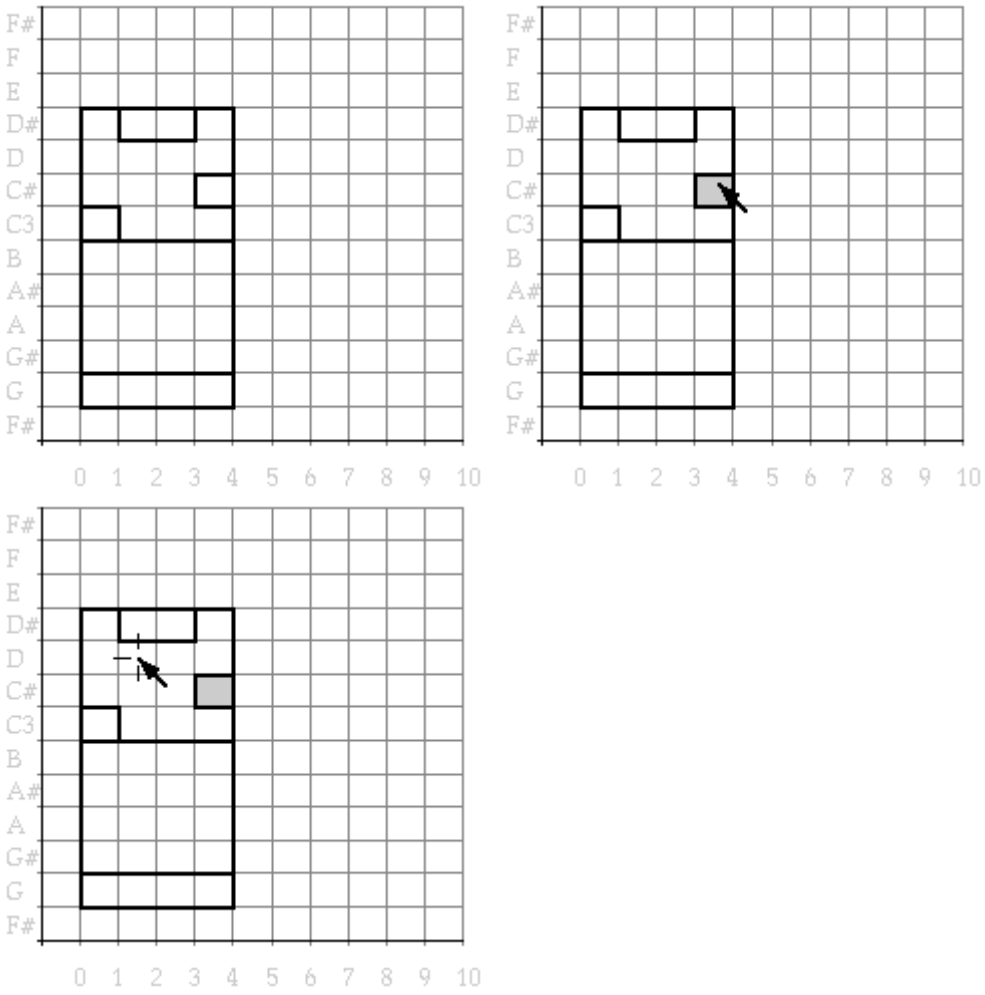

Figure 26. testing operations on a selection

Making musical objects mouse sensitive

You knew this had to come sooner or later didn't you: we want to select musical objects with the mouse. Again there are several possibilities to define the area that will select a compound musical object, one of them is the boundary itself. The other possibility, that we will explore here interprete mouse-clicks in the smallest boundary thta contains it. Thus we first have to look for notes, and whether the mous was on one of them, then we go one level up to check whether the mouse was in the boundary box of a compound musical object (but not on its notes), etc. The flow of control is thus exactly oppposite the top-down way in which `find- musical- object- onset` searches through the structure. By reversing the order of the clauses in the `or` in the body of that function, our bottom-up search can be realized with one simple change.

```
(defmethod deep-find-musical-object-onset ((object compound-musical-object)

onset predicate &rest args)

"Return first non-nil predicate's result, testing its sub-objects, then itself"

(or (apply #'find-element-onset object onset

#'deep-find-musical-object-onset predicate args)

(apply predicate object onset args)))


(defmethod deep-find-musical-object-onset ((object basic-musical-object)

onset predicate &rest args)

"Return predicate's result, when satisfied for this object at its onset"

(apply predicate object onset args))


(defmethod covers? ((object musical-object) onset window time pitch)

"Does the point (time, pitch) lay in the boundary of the musical object?"

(destructuring-bind (onset max-pitch offset min-pitch)

(boundary object onset window)

(and (<= onset time offset)

(<= min-pitch pitch max-pitch)

object)))


(defmethod select-cover ((window window) time pitch)
```

```
"Select the musical object at (time, pitch) in the window"

(select window

(deep-find-musical-object-onset (object window) 0 #'covers?

window time pitch)))
```

We can check our search by hand before we have access to the mouse sensitivity.

```
(draw (example))

(do-front-window #'select-cover .5 60) =>
```

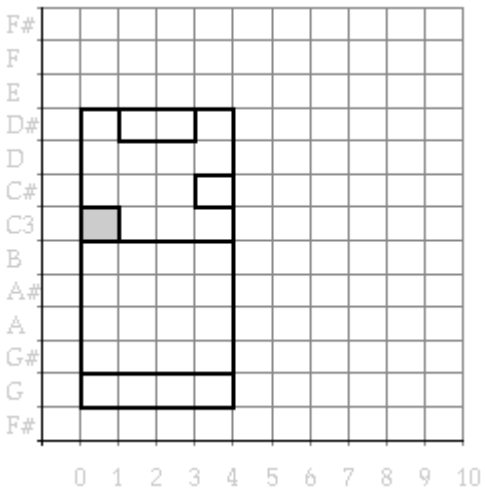

Figure 27. Testing the selection by pointing to a position

We rely on the piano roll window to translate the screen coordinates to our time-pitch domain because that is the proper place for that knowledge to reside, (think, e.g., of a later adition of a scroll bar in the window, which effectively modifies the screen to domain mapping). [8] The piano roll window window will send a `mouse- click` message (with the moused time and pitch positions, the modifier keys and the number of clicks) to itself that we catch and act upon by selecting the smallest sub-object that covers mouse. An `eql` specialisation on the last two arguments ensures that only single mouse clicks without modifier keys are being acted upon.

```
(defclass mouse-select-mixin (selection-mixin)

()

(:documentation "A mixin that makes musical objects mouse selectable"))
```

```
(defmethod find-position-in-musical-object ((object musical-object)
(window mouse-select-mixin) time pitch)
"Return the (sub)object at (time, pitch)"
(find-musical-object-onset object 0 #'covers? time pitch))

(defmethod mouse-click ((window mouse-select-mixin)
time pitch (modifiers (eql nil)) (count (eql 1)))
"Select in response to a single mouse click at (time, pitch) without modifier keys"
(select-cover window time pitch))

(defclass draw-window (mouse-select-mixin
structured-mixin
visible-selection-mixin
piano-roll-contents-draw-window)
()
(:documentation "A mouse-friendly pianoroll window class"))
```

Now the fun can really start: mouse-selecting objects, applying transformations on the by using the menu or command keys: our basic user interface is in place.

```
(defmethod front-mouse-click (&rest args)
(apply #'do-front-window #'mouse-click args))

(draw (example)) => 29.A. click on an object
(front-mouse-click 3.6 61.2 nil 1) => 29.B. selected object
(front-draw-window) => 29.C. click elsewhere
(front-mouse-click 1.5 62 nil 1) => 29.D. new selection
```

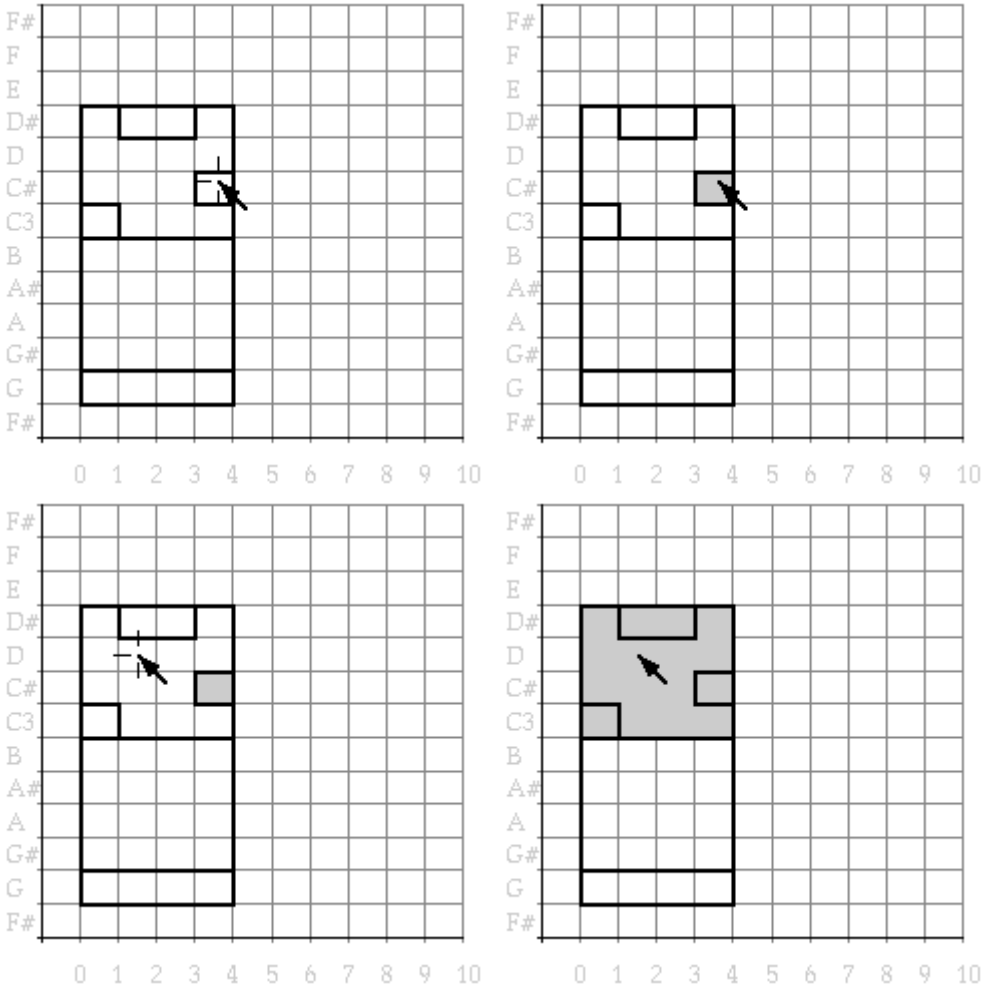

Figure 29. Selections in the example made visible

[to do: waar geven we commando's (bv transpose) argumenten pas in embedded of al eerder, hier?]

[to do: backpointers maken canon en repeat mogenlijk als destructieve window operatie die toevoegen in menu etc.]

Definitions made

`view-draw-contents (window)` *`method`*

(Re)draw the stored object as a pianoroll

`visible-selection-mixin (selection-mixin)` *`class`*

A class to support visible selections in draw windows

`mouse-select-mixin (selection-mixin)` *`class`*

A mixin that makes musical objects mouse selectable

`combine-boundary-mixin (draw-window-mixin)` *`class`*

A mixin to combine boundary boxes

`structured-mixin (combine-boundary-mixin) class`

A mixin to visualize internal structure

`draw-window (mouse-select-mixin structured-mixin visible-selection-mixin piano-roll-contents-draw-window) class`

A mouse-friendly pianoroll window class

`piano-roll-contents-draw-window (piano-roll-window) class`

A window for drawing and re-drawing musical objects

`shaded-mixin (draw-window-mixin) class`

A window that shades the notes of the pianoroll

`draw-window-mixin nil class`

Additional behavior for draw window

`map-time (object onset time-map) method`

Adjust the duration of a basic musical object according to the time map

`front-select (selector &rest args) method`

Apply a function which will state what to select in the front window

`front-window-operation (operation &rest args) function`

Apply an operation to a musical object stored in the frontmost draw window

`front-selection-operation (operation &rest args) function`

Apply an operation to the selection stored in the frontmost draw window

`selection-operation (window operation &rest args) function`

Apply the operation to a musical object stored in a window

`window-operation (window operation &rest args) function`

Apply the operation to a musical object stored in a window

`combine-boundaries (&rest boundaries) method`

Combine a list of boundary boxes into one box that encloses them all

`front-deselect nil method`

Deselect everything in the front window

`same-set (set1 set2 &key (test #'eql)) function`

Do two list contain the same items in any order

`same-list (list1 list2 &key (test #'eql)) function`

Do two lists contain the same items in the same order

`covers? (object onset window time pitch) method`

Does the point (time, pitch) lay in the boundary of the musical object?

`draw (object &rest args &key window-class &allow-other-keys) method`

Draw a graphical piano-roll representation of a compound musical object

`draw-musical-object (object onset window) method`

Draw a graphical representation of a basic musical object

`selection-mixin (draw-window-mixin) class`

Our pianoroll window class

`draw-boundary (object boundary window) method`

Paint selected objects

`play-selection (window) method`

Play the musical object selected in the window

`mirror-selection (window) method`

Reflect the musical object selected in the window around Middle C

`sounding-pitches (object onset from to) method`

Return a list of the pitch of the note, when sounding in interval [from, to]

`boundary (object onset window) method`

Return a list of the sides of a box enclosing a compound musical object

`make-tempo-change (duration begin end) function`

Return a time-warp function based on begin and end tempo factors

`find-element-onset (object onset predicate &rest args) method`

Return first non-nil predicate's result, testing elements and their onset

`same? (object1 object2) method`

Return nil as default case: two musical objects of different type are unequal

`deep-find-musical-object-onset (object onset predicate &rest args) method`

Return predicate's result, when satisfied for this object at its onset

`find-musical-object-onset (object onset predicate &rest args) method`

Return predicate's result, when satisfied for this object at its onset

`find-position-in-musical-object (object window time pitch) method`

Return the (sub)object at (time, pitch)

`collect-sounding-pitches (object onset from to) method`

Return the empty list representing silence

`front-draw-window nil` *function*

Return the frontmost drawing window, bringing it to the front as well

`map-elements-onsets (object onset operation &rest args)` *method*

Return the results of the operation applied to each element and its onset

`front-selection nil` *method*

Return the selection of the front window

`square (x)` *function*

Return the square of a number

`set-union (&rest args)` *function*

Return the union of any number of sets

`retrograde (object)` *method*

Reverse all components of the object in time

`retrograde-selection (window)` *method*

Reverse the musical object selected in the window

`mouse-click (window time pitch modifiers count)` *method*

Select in response to a single mouse click at (time, pitch) without modifier keys

`front-select-all nil` *method*

Select the complete musical object in the front window

`select-cover (window time pitch)` *method*

Select the musical object at (time, pitch) in the window

`select-all (window)` *method*

Select the whole musical object in the window

`deselect (window)` *method*

Set the window selection to nil

`transpose-selection-down (window)` *method*

Transpose the musical object selected in the window down by one semitone

`transpose-selection-up (window)` *method*

Transpose the musical object selected in the window up by one semitone

`use-first-arg (function)` *function*

Wrap the function such that it can be given spurious arguments

`front-mouse-click (&rest args)` *method*

`select (window object)` *method*

`do-front-window (action &rest args)` *function*

Literature references made

Balaban, M. 1992 Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music. in Balaban, M., K. Ebcioglu & O. Laske (Eds.) Cambridge, Mass.: MIT Press. 111-138

Chowning, 1973 Journal of the Audio Engineering Society, 21(7):526-534. Reprinted inC. Roads and J. Strawn, (Eds.) 1985 Foundations of Computer Music. Cambride, MA: MIT Press. pp. 6-29.

Dannenberg, R.B. 1991 Real-time Scheduling and Computer Accompaniment. in Mathews, M.V and J.R. Pierce (Eds.) . Cambridge, MA: MIT Press.

Dannenberg, R.B. 1993 The Implementation of Nyquist, a Sound Synthesis Language. in San Francisco: ICMA.

Dannenberg, R.B., P. Desain and H. Honing (in Press) Programming Language Design for Music. in G. De Poli, A. Picialli, S.T. Pope & C. Roads (Eds.) Lisse: Swets & Zeitlinger.

Desain, P. and H. Honing (In Preparation) CLOSe to the edge, Advanced Object-Oriented Techniques in the Representation of Musical Knowledge. Jounral of New Music Research.

Desain, P. 1990 Lisp as a second language, functional aspects. (28)1 192-222

International MIDI Association 1983 North Holywood: IMA.

Moore, F.R. 1990 Englewood Cliffs, New Jersey: Prentice Hall.

Pope, S.T. 1992 The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers.16(3) 73-91

Pope, S.T. 1993 Machine Tongues XV: Three Packages for Software Sound Synthesis. 17(2) 23-54

Steele, G.R. 1990 Bedford MA: Digital Press.

Tatar,D.G. 1987 Bedford, MA: Digital Press.

Glossary references made

abstract class

abstract class

access function

all functions part of a data abstraction layer (selector and constructor functions). [loaded from Glossary Functional]

accessor function

after method

anonymous function

A function whose ‘pure’ definition is given, not assigning it a name at the same time. [loaded from Glossary Functional]

applicable method

applicable methods

application

obtaining a result by supplying a function with suitable arguments. [loaded from Glossary Functional]

assignment

atom

in Lisp: any symbol, number or other non-list. [loaded from Glossary Functional]

before method

class

class hierarchy

class options

combinator

A function that has only functions as arguments and returns a function as result. [loaded from Glossary Functional]

compile time

congruent lambda lists

cons

A Lisp primitive that builds lists. Sometimes used as verb: to add an element to a list. [loaded from Glossary Functional]

constant function

A function that always returns the same value [loaded from Glossary Functional]

constructor function

A function that as part of the data abstraction layer provides a way of building a data structure from its components. [loaded from Glossary Functional]

constructor functions

continuations

A way of specifying what a function should do with its arguments. [loaded from Glossary Functional]

coroutines

parts of the program that run in alternation, but remember their own state of computation in between switches. [loaded from Glossary Functional]

data abstraction

A way of restricting access and hiding detail of data structures [loaded from Glossary Functional]

data abstraction

data type

A class of similar data objects, together with their access functions. [loaded from Glossary Functional]

declaration

declarative

declarative method combination

dialect

A programming language can be split up into dialects that only differ (one hopes) in minor details. Lisp dialects are abundant and may differ a lot from each other even in essential constructs. [loaded from Glossary Functional]

direct superclasses

dynamically typed language

effective method

first class citizens

rule by which any type of object is allowed in any type of programming construct. [loaded from Glossary Functional]

free variables

function

A program or procedure that has no side effects [loaded from Glossary Functional]

function composition

the process of applying one function after another. [loaded from Glossary Functional]

function quote

A construct to capture the correct intended meaning (with respect tothe current lexical environment) of a anonymous function so it can beapplied later in another environment. It is considered good programming style to use function quotes as well when quoting the name of a function. [loaded from Glossary Functional]

functional abstraction (or procedural abstraction)

A way of making a piece of code more general by turning part of it into a parameter, creating a function that can be called with a variety of values for this parameter. [loaded from Glossary Functional]

functional argument (funarg)

A function that is passed as argument to another one (downward funarg) or returned as result from other one (upward funarg) [loaded from Glossary Functional]

generalizedvariable

generic

global variables

an object that can be referred to (inspected, changed) from any part of the program. [loaded from Glossary Functional]

higher order function

A function that has functions as arguments. [loaded from Glossary Functional]

imperative style

inheritance

initializationkeyword

initializationprotocol

initialization argument list

initialization keyword

instance

instantiation

iteration

repeating a certain segment of the program. [loaded from Glossary Functional]

lambda list keyword

lambda-list keyword

A keyword that may occur in the list of parameter names in a function definition. It signals how this function expects its parameters to be passed, if they may be ommited in the call etc. [loaded from Glossary Functional]

lexical scoping

A rule that limits the ‘visibility’ of a variable to a textual chunk of the program. Much confusion can result from the older- so called dynamic scoping - rules. [loaded from Glossary Functional]

list

message passing

message passing style

meta object protocol

method

method combination

method combination

The declaritive way in which CLOS allows more methods to be bundled and run, in situations where more are applicable [loaded from Object Oriented I]

method qualifier

methods

mixin class

modification macro's

most specific method

multi-methods

multi-methods

multiple inheritance

multiple inheritance

name clash

name clashes

object oriented programming

A style of programming whereby each data type is grouped with its own access function definitions, possibly inheriting them from other types. [loaded from Glossary Functional]

object-oriented style

parameter-free programming

A style of programming whereby only combinators are used to build complex functions from simple ones. [loaded from Glossary Functional]

part-of hierarchy

polymorphic

polymorphism

prefix notation

A way of notating function application by prefixing the arguments with the function. [loaded from Glossary Functional]

primary method

primary methods

procedural

quote

A construct to prevent the Lisp interpreter from evaluating an expression. [loaded from Glossary Functional]

read-eval-print

record

recursion

A method by which a function is allowed to use its own definition. [loaded from Glossary Functional]

run time

selector function

A function that as part of the data abstraction layer provides access to a data structure by returning part of it. [loaded from Glossary Functional]

selector functions

setf method

shadow

side effect

Any actions of a program that may change the environment and so change the behavior of other programs. [loaded from Glossary Functional]

side effect

slot descriptors

slots

stack

A list of function calls that are initiated but have not yet returned a value. [loaded from Glossary Functional]

standard method combination

statically typed

structure preserving

A way of modifying data that keeps the internal construction intact but may change attributes attached to the structure. [loaded from Glossary Functional]

tail recursion

A way of recursion in which the recursive call is the ‘last’ thing the program does. [loaded from Glossary Functional]

the most specific method

untyped language

[an error occurred while processing this directive]