[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 III

Draft mars 11, 1997

From iterator to recursor again

Remember the `find- musical- object- onset` recursor, which walks throuh a complete musical structure, uses `find- element- onset` to move in the width direction (through the elements list of an compound musical object), and supplies itself to this iterator to have itself applied recusively to proceed in the depth direction. In the same way we can use `mapc- elements- onsets` as basic horizontal iterator to create a true recursor to apply some operation to each (sub) object of a musical structure. But instead of a predicate to test, this recursor expects an operation to apply. [1]

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

onset operation &rest args)

"Apply the operation to this object, and to each sub-..-sub-object and their onsets"

(apply operation object onset args)

(apply #'mapc-element-onset object onset #'mapc-musical-object-onset operation args))


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

onset operation &rest args)

"Apply the operation to this object and its onset"

(apply operation object onset args))
```

Betere uitleg?, beginnen met basic, dan eerst uitschrijven als

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

onset operation &rest args)
"Apply the operation to this object, and to each sub-..-sub-object and their onsets"

(apply operation object onset args)

(loop for element in (elements object)

for element-onset in (onsets object onset)

do (mapc-musical-object-onset element element-onset operation args)))
```

This recursion scheme is the one used implicit by the `draw- musical- object` function. Although distributing the management of the flow of control through the object oriented code is quite natural, each method calling the code to operate on parts of it, in a subtle way it may give rise to lots of code duplication, when many different operations in fact walk through the data structures in the same way. Making the walk itself into an explict method, like we did in `mapc- musical- object- onset`, in a sense we create not a real program, we only capure the behavior of walking through a musical structure and has still to be supplied with an argument which specifies the ‘thing to do’ while visiting each part[2]. This yields a nice modularity, and allows for dense definitions of complex processes like the draw program that can be written now in a couple of lines.

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

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

(mapc-musical-object-onset #'draw-musical-part onset window))


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

"Draw a graphical representation of the musical object itself"

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


(draw (example)) =>
```

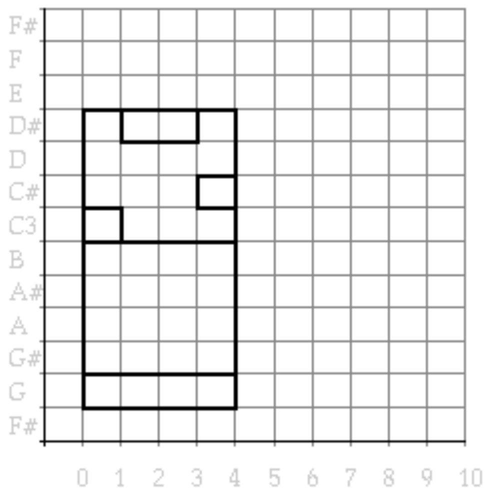

Figure 1. A test, not to appear in book

Of course, a simpler `mapc- musical- object`, which does not calculate onset times can be based in an analogue manner on `mapc- element`.

```
(defmethod mapc-musical-object ((object compound-musical-object) operation &rest args)

"Apply the operation to this object and each sub-..-sub-object"

(apply operation object args)

(apply #'mapc-element-onset object #'mapc-musical-object operation args))


(defmethod mapc-musical-object ((object basic-musical-object) operation &rest args)

"Apply the operation to this object"

(apply operation object args))
```

[to do: This recursor can be trimmed for use by operations that need only be applied to specific objects by adding a predicate. E.g. the transpose (combine-and (wrap #'note?) operation) of hele ding weglaten ?]

And again from iterator to recursor

The final lift of "horizontal" iteration code, to a full recursor definition is needed for `map- elements- onsets`. The difficulty is here that, in contracts to `mapc- elements- onsets` which returns no useful result (it is onnly called for side-effect) and `find- elements- onsets` which returned only one result, this new recursor needs to combine the results at each level of the structure with the results of lower levels and return that to the caller.

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

onset combination operation &rest args)

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

(apply combination

object

onset

(apply operation object onset args)

(apply #'map-elements-onsets

object
onset

#'map-musical-object-onsets

combination

operation

args)

args))
```

```
(defmethod map-musical-object-onsets ((object basic-musical-object)

onset combination operation &rest args)

"Apply the operation tho the object and its onset"

(apply operation object onset args))
```

If not understood, uitschrijven functional style: can substitueren body van map-elements-onsets recursive call for each element again if not understood substitute body of map-musical-object-onsets

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

onset combination operation &rest args)

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

(apply combination

object

onset

(apply operation object onset args)

(loop for element in (elements object)

for element-onset in (onsets object onset)

collect (apply #'map-musical-object-onsets element

element-onset

combination

operation

args))

args))
```

Our example that uses this recursor will calculate a flat, non-hierarchical, representation of musical objects, a kind of note list (as is often used in composition systems).

event lists

[to do: tekst aanpassen]

To facilitate translating our musical objects to other representations that use flat agenda-like structures (like standard MIDI files of score files for Csound or Cmusic), we will write a function that can distill an event list, a list that contains pairs of start times and objects in some format. The `event- list` method collects all the calls `object- note- list` on musical objects to obtain the event list which represents by the object itself: a list of the onset-object pair. Compound musical objects only have to append the note list of their components to yield the event list for that object itself.

```
(defmethod note-list ((object musical-object) onset)

"Return a list of onset-event pairs of all parts of a musical object"

(map-musical-object-onsets object onset #'combine-event-lists #'object-note-list))


(defmethod combine-event-lists ((object compound-musical-object) onset self parts)

"Append the event lists of the parts of an object"

(apply #'append (cons self parts)))


(defmethod object-note-list ((object musical-object) onset)

"Return a basis event list consisting of one onet-part pair"

nil)


(defmethod object-note-list ((object note) onset)

"Return a basis event list consisting of one onet-part pair"

(list (list onset object)))


(pprint (note-list (example) 0)) =>

((0 (note :duration 1 :pitch "C3"))

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

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

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

Simplify to understand, transparency, substitute and sumstitute back

```
(defmethod note-list ((object musical-object) onset)
"Return a list of onset-event pairs of all parts of a musical object"

(combine-event-lists object

onset

(object-note-list object onset)

(loop for element in (elements object)

for element-onset in (onsets object onset)

collect (note-list element element-onset))))
```

This version is quite specific, it requires a redefinition of `object- note- list` whenever another format is needed. A slightly more general version allows passing each part the method to use for calculating the format of the basic event list. **[to do: parameterize as general method]**

```
(defmethod event-list ((object musical-object) onset object-event-list-method)

"Return a list of onset-event pairs of all parts of a musical object"

(map-musical-object-onsets object

onset

#'combine-event-lists

object-event-list-method))
```

Using two simple methods for `object- note- list` can now, when plugged into the `event- list` function yield an event list which only contains the notes, a result which is needed when the representation that we are converting our musical objects to supports no structure. **[to do: this is the reconstruction of note-list]**

```
(defmethod note-list ((object musical-object) onset)

"Return a list of onset-event pairs of all parts of a musical object"

(event-list object onset #'object-note-list))
```

[to do: voorbeeld met ander formaat hier bv al Cmusic]

[to do: vuistregel: algemene geval is makkelijker]

Though the function succeeds in extracting al notes and their onsets from a structured musical object, the resulting note list does not necessarily appear in time order, a requirement that is often encountered. One solution is to sort the list after it has been made. Note how the `sort` function is used with a proper key to retrieve the onset time as the first of each pair, this will act as the index upon which the list is sorted.

```
(defmethod sorted-note-list ((object musical-object) onset)

"Return a note list: (onset note) pairs, with onsets in ascending order"

(sort-event-list (note-list object onset)))


(defun sort-event-list (event-list)

"Sort a list of onset-object pairs in order of increasing onsets"

(sort event-list #'< :key #'first))


(pprint (sorted-note-list (example) 0)) =>

((0 (note :duration 1 :pitch "C3"))

(0 (note :duration 4 :pitch "G2"))

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

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

However, sorting is an expensive operation, and e.g. the event-lists for the components of a sequential musical object will already all follow each other in time. Appending them together directly is possible, without further sorting. And for parallel objects, assuming that the recursive calls of as-event-list derive event-lists for the components that are sorted themselves, they only have to be merged together, which is a much more efficient processes than sorting.

We need to write the auxiliary functions that combine event lists. Because the predefined `merge` function only can deal with two lists at a time we will need to extend its use to be able to merge a set of sorted lists. The `reduce` function is a handy tool in these situations. It can generalize binary operations (operations of two arguments) to the n-ary case. E.g. `(reduce #'+ '(a b c)) = (+ a (+ b c))`. The `merge` function is first tailored to use for event lists by requiring it to return its result as a list and by supplying it with a key parameter: the function that extracts from each event the aspect on which the sorting has to be done. This is here the time of the event: the first element of the onset-object pair.

```
(defun event-merge (list1 list2)

"Return a sorted event list of all events of both arguments"

(merge 'list list1 list2 #'< :key #'first))
```

[to do: naam event-merge beter kiezen]

```
(defun merge-events (&rest events)

"Merge a list of sorted agenda's into one sorted agenda"

(reduce #'event-merge events))
```

```
(defmethod combine-event-lists ((object parallel) onset self parts)

"Merge a list of sorted agenda's into one sorted agenda"

(apply #'merge-events (cons self parts)))


(defmethod combine-event-lists ((object sequential) onset self parts)

"Concatenate a list of sorted agenda's into one sorted agenda"

(apply #'append (cons self parts)))
```

Now event lists will be guaranteed to be sorted at every level directly, and the top function like `note-list` needs not sort the result anymore - a much more efficient solution.

```
(pprint (note-list (example) 0)) =>

((0 (note :duration 1 :pitch "C3"))

(0 (note :duration 4 :pitch "G2"))

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

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

[to do: remark on optimization as extra specialization methods]

Promoting event lists to musical objects

From here we can go two ways: further upward, trying to incorporate some sort of event list notion into our musical objects (and allow transformations, drawing etc.), and further downward towards simple data structures and score file writers. We will indeed travel both ways, starting with the former. It is claimed often that the object oriented style enables programmers to quickly add small pieces of code that takes care of new situations. We will do so now by introducing a new musical object and integrating it into the code suite we have build up. This will test the flexibility of the classes and methods we defined and form a kind of summary in which the different constructs are used once again.

We want to introduce an object with a more absolute way of specifying the onsets of its elements, similar to an event list. Or alternatively, we want to extend our code, which is completely score-based, a bit more towards performance data. Then we need to define an object that can arbitrarily specify the timing of objects as well - not just a strict parallel or sequential order. It will be a compound musical object, but next to a list of elements, it will hold a list of onset times, one for each element. These onset times will be relative to the start time of the object itself, a nice middle ground between absolutely absolute time specification of flat event lists and a completely relative time specification like in the compound musical objects. We will call his agenda-like structure a `collection`. It can be used in a hierarchical way, nested with another collection or compound musical object and it can contain other collections or compound musical objects. It resembles the event-lists of e.g. SmOKe (Pope, 1992) closely. The `collection` class can inherit the `elements` slot from the abstract `compound- musical-object` class.

[to do: plaatje hier neerzetten?]

```
(defclass collection (compound-musical-object)

((delta-onsets :accessor delta-onsets :initarg :delta-onsets :initform nil

:documentation "The relative start times of the elements"))

(:documentation "A group of musical objects, starting at specified times"))
```

Now we can define a constructor function for a collection that makes its call easy to read, by alternating onsets and elements.

```
(defun collection (&rest agenda)

"Return a collection of musical objects starting at specified onsets"

(let ((delta-onsets (loop for delta-onset in agenda by #'cddr

collect delta-onset))

(elements (loop for element in (rest agenda) by #'cddr

collect element)))

(make-instance 'collection :delta-onsets delta-onsets :elements elements)))


(collection 0.3 (note :duration 1 :pitch "C3")

1.7 (note :duration 2 :pitch "B2")

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

->

#<error printing collection #xD80EF9>
```

[to do: which reminds us of printing need to be show invoked because compound]

[to do: definition print-object ophalen]

show method that needs to be defined and, for consistency, needs to reflect the textual representation of the way in which we construct collections.

```
(defmethod show ((object collection))

"Return a list describing a collection of musical objects"
(cons 'collection

(loop for onset in (delta-onsets object)

for element in (elements object)

collect onset

collect (show element))))
```

We can check the `show` method and define an example for later use. [3]

```
(defun example-collection ()

"Return a simple collection of 5 notes"

(collection 0.0 (note :duration 4 :pitch "G2")

0.3 (note :duration 1 :pitch "C3")

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

1.7 (note :duration 2 :pitch "B2")

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


(example-collection)

->

(collection 0.0 (note :duration 4 :pitch "G2")

0.3 (note :duration 1 :pitch "C3")

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

1.7 (note :duration 2 :pitch "B2")

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

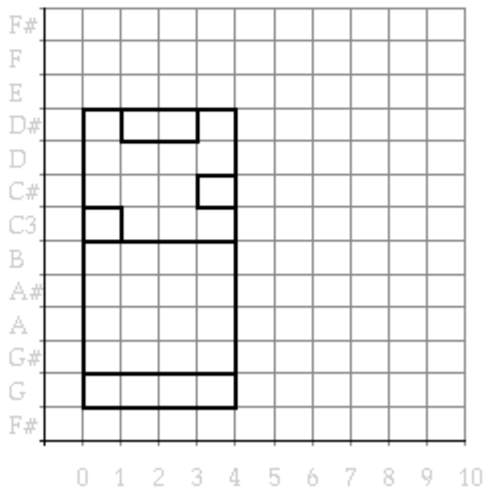

Figure 2. A collection.

This collection is shown in Figure 2, a figure whose drawing has to be postponed till we have defined some more methods to work for collections as well

The accessor function defined on musical objects all have to be checked and possibly defined or re-defined. One of the main ones is the onsets method on which most of the iterators and recursors are based. The onsets of the elements of a collection are calculated by adding their relative onsets to the onset of the collection itself.

```
(defmethod onsets ((object collection) &optional (onset 0))

(loop for element-onset in (delta-onsets object)

collect (+ element-onset onset)))
```

Collections are different than othe compound musical objects in that it makes sense to talk about changing the onset time of their elements. Setf methods can be made to have more arguments, and it is natural to design updates to the onsets of the elements of a collection as having an optional argument that represents the onset time of the object itself.

```
(defmethod (setf onsets) (onsets (object collection) &optional (onset 0))

"Set the onset times of a collection, relative to the onset of the object"

(setf (delta-onsets object)

(loop for element-onset in onsets collect (- element-onset onset))))
```

[to do: &optional ??]

[to do: knippen]

Since onsets and elements are the only access methods needed by the map-elements-onsets method for compound musical objects, functions can be directly based on it

The duration of a collection is calculated by returning the offset time of the component that ends last. A type predicate and the equality check are easily defined as well.

```
(defmethod duration ((object collection))
"Return the duration of a collection"

(apply #'max (map-elements-onsets object 0 #'offset)))


(defmethod collection? ((object t))

"Return t if the object is an instantiation of class collection, nil otherwise"

(typep object 'collection))


(defmethod same? and ((object1 collection)(object2 collection))

"Are two two parallel musical objects equal?"

(and (equal (delta-onsets object1)(delta-onsets object2))

(same-set (elements object 1)(elements object2) :test #'same?)))


(duration (example-collection))

->

4.1
```

Drawing now works

To make `retrograde` work for collections as well, a method definition needs to be added. It calculates the time left from the end of each component till the end of the whole collection. These times form the new onsets of the components of a retrograde collection. The result of this transformation applied to a collection is shown in Figure 3.

```
(defmethod retrograde ((object collection))

"Reverse all components of the object in time"

(setf (onsets object)

(loop with duration = (duration object)

for element in (elements object)

for element-onset in (delta-onsets object)

collect (- duration

(offset element element-onset))))

object)
```

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

(pause :duration 1)

#'retrograde)) =>
```

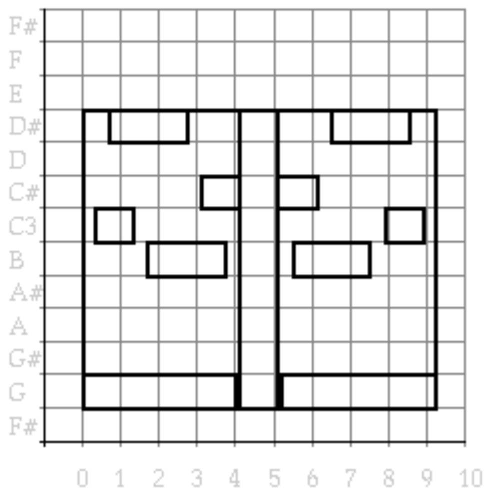

Figure 3. The collection and its retrograde.

Because show works well, so will the `copy- object` and thus the `repeat`, `canon`, and `original- and-transform` functions. Most transformations rely on one of the mappers, and the definition of the mappers for compound musical objects rely only on the availability of an `onsets` method to calculate proper onset times for the elements. Thus the definitions of the mappers will turn out to work just fine for collections. [4] And `transpose` and `mirror` etc. will work well for collections automatically, because they only need a well defined iterator, as will `draw`. One other function that has to be written is the `combine- event-lists` method for collections. Collection behaves like parallel structures in this respect: when event lists of their components are to be combined, they have to be mergerd in the proper order.

```
(defmethod combine-event-lists ((object collection) onset self parts)

"Merge a list of sorted agenda's into one sorted agenda"

(apply #'merge-events (cons self parts)))
```

All our compound musical objects can be converted to a collection. In this sense a collection is a more general data structure. However, a collection tells less about the time order of the components, it is less specific. [5]

An around method can prevent a call of the primary method

Consider a protection that would be desirable for modifications of the pitches of musical objects. By mistake a transposition or mirror operation can easily create pitches outside a permitted range. E.g. for

MIDI the pitches should be between 0 and 127. Lets try what happens when we create note that is too low.

```
(pitch (transpose (note :pitch "C1") -40))

->

-4
```

Of course we would not be doing CLOS if we were not to be looking for way in which we can program a prper protection without changing the code of `transpose`. A first approach could be to cause an error in a method before the harm is done.

```
(defmethod transpose :before ((object note) interval)

"Break when the transposition would move the pitch outside the MIDI range"

(unless (<= 0 (+ (pitch object) interval) 127)

(error "~A cannot be transposed by ~A semitones" object interval)))
```

```
(pitch (transpose (note :pitch "C1") -40))

=>

> Error:

> (note :duration 1 :pitch "C1") cannot be transposed by -40 semitones

> While executing: #<standard-method transpose :before (note t)>
```

This is not a bad approach, and in many cases it is appropriate. However, would we like to ignore the update and proceed with the rest of the calculations, this method would not work. For these situations CLOS supports a particular kind of method which can control the invocation of other methods: the around method. When such a method (with the qualifier `:around`) is among the applicable methods, that method will be called instead of any before, after or primary one. And it can decide to do all calculation itself and return a result. However, the around method can also actively ask for the primary method (plus possible before and after methods) to run and obtain the result of the primary method. It can do so by evaluating the form `(call- next- method)`. In this sense the method combination is not declarative anymore: calling the primary method is under direct control of the program. When the `(call- next- method)` form is evaluated, the primary method is called with the same arguments as the around method itself. This mechanism can solve our pitch out-of-range protection, the real transposition is only invoked when it is save to do so. Remember that we made transformations always return their object, thus even in this pathological case, whe the transformation refuses to work, we need to adhere to that protocol.

```
(defmethod transpose :around ((object note) interval)

"Don't transpose when this would move the pitch outside the MIDI range"

(if (<= 0 (+ (pitch object) interval) 127)
(call-next-method)

object))



(transpose (note :pitch "C1") -40)

->

(note :duration 1 :pitch "C1")
```

Recognizing that the `mirror` transform needs a similar protection, we really want to shift the guard further downward to prevent any illegal pitch update. Since all modifications of the pitch slot of a note are made using a `setf` writer method which was generated automatically by the system, and we already know these *setf methods* are defined, it is no big step to write a before setf method to properly disallow all illegal pitch updates. Remeber to always let `setf` return its value.

```
(defmethod (setf pitch) :around (value (object note))

"Call the primary method only with a pitch that is kept within [0,127]"

(if (<= 0 value 127)

(call-next-method)

value))



(mirror (note :pitch "C1") "C5")

->

(note :duration 1 :pitch "C1")
```

In a primary method the next method may be called as well, but please don't

The `call- next- method` form can be used in any primary method too (not only in around methods), calling the most specific method shadowed by this one. This fact is considered an invitation to programmers to write ugly code for which actual method combination may become a hard puzzle of control-flow, especially since the decision whether a call to `call- next- method` is done depends on the actual computation, not just on the program text. Therefore it is better to avoid this use of `call- next- method`.

An around method can change the arguments of its primary method

There is still another way to protect pitch updates outside of the allowed range: by first clipping the new value to be inside that range. Thus we would like to call the update method but now with different arguments. Around methods can do that by calling `call- next- method` with a new set of arguments. This is a fine way to wrap a protection around an assignment.

```
(defmethod (setf pitch) :around (value (object note))

"Call the primary method with a pitch that is kept within [0,127]"

(call-next-method (clip 0 value 127) object))
```

[to do: import this function]

```
(defun clip (min value max)

"Return the value, clipped within [min,max]"

(cond ((< value min) min)

((> value max) max)

(t value)))


(pitch (mirror (note :pitch "C1") "C5"))

->

127
```

As a last extension we can allow the pitch to be set to a pitch name directly, a translation that we had already implemented in the initial note construction.

```
(defmethod (setf pitch) :around (value (object note))

"Call the primary method with a pitch number"

(let* ((number (if (stringp value)(pitch-name-to-midi-number value) value))

(safe-number (clip 0 number 127)))

(call-next-method safe-number object)))


(let ((object (note)))

(setf (pitch object) "C#5")

object)

->

(note :duration 1 :pitch "C#5")
```

An around method can modify the results returned by the primary method

Now we can try a rough solution to the problem that the boundary boxes in our piano-roll drawings do not nest properly. It is obtained by enlarging the boundary enclosing each object a bit before it is returned.

This is an example of the use of an around method that amends the return value of the primary method. is not at all a trivial task to define an algorithm that draws the hierarchical structure, the boxes around compound musical objects, in such a way that edges do not overlap. That is because the diagramatic time-pitch plot of the piano-roll notation does not leave space for depicting the logical hierarchical structure. But a quite nice solution can be found if we agree on stealing a bit of space from some note boxes (and if we disregard the possibility of parallel grouping with largely overlapping pitch ranges). **[to do: dit is niet waar hier, no stealing]**The result of this kind of drawing is shown in Figure ? that can be compared to the original Figure 7 which had no nesting. [6]

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

()

(:documentation "A mixin to draw the structure in a nested fashion"))



(defmethod boundary :around ((object compound-musical-object) onset (window nested-
mixin))

"Draw the boundary as side effect of its calculation"

(declare (ignore onset))

(enlarge-rectangle (call-next-method)

(nested-margin object window)))



(defmethod nested-margin ((object musical-object) (window nested-mixin))

"The margins of a nested compound boundary box"

(let ((horizontal .15)(vertical .2))

(list (- horizontal) vertical horizontal (- vertical))))



(defun enlarge-rectangle (box margins)

(mapcar #'+ box margins))



(defclass draw-window (mouse-select-mixin

nested-mixin

structured-mixin

visible-selection-mixin

piano-roll-contents-draw-window)

()

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

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

(pause :duration 1)

#'mirror 60)) =>
```

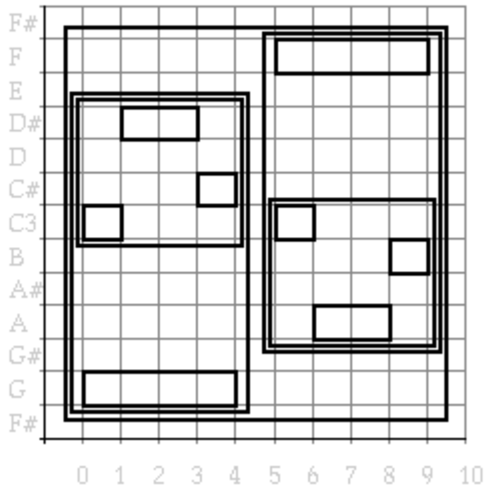

Figure 4. Nested drawing

This complicated program change for the nested drawing of musical objects thus boils down to the modification of the calculation of the boundary of a box by adding one around method. Apart from the clever design of the CLOS language that makes this possible, it illustrates how the use of fine grained modularity in writing programs, pays itself back in ease of program modification. The modification would have been much harder and more error prone had we not introduced an auxilary boundary method in the body of the original draw object methods. Students of object oriented programming should train the sensitivity to spot the places in a program where an extra hook, in the form of the call to an auxilary function, could become indispensible in later extentions or modifications. Reusing and rewriting pieces of code a lot, and reflecting on the cases in which reusing was not possible due to design flaws makes good practice for increasing that sensitivity. Reusable code is too often a commercial slogan instead of an approriate description of object oriented programs - that should change.

[to do: wat kleiner maken noten aan de hand van diepte, met shrink-boundary ?]

Naming musical objects

In searching through musical objects, the compound objects can only be distinguished by onset time and duration and the pich range they cover. In modeling musical knowledge using our classes, it would be nice to be able to distinguish, say a sequential structure representing a bar, from one representing a voice of a piece. Instead of incorporating much more musical knowledge into our class hierarchy[7] we will introduce names for musical objects. These names can be the link to knowledge that will be kept outside the system itself. The name slot can be added to the existing class definition for compound musical objects.

```
(defclass named-musical-object-mixin ()

((name :accessor name :initarg :name :initform nil))

(:documentation "A named object"))


(defclass compound-musical-object (musical-object named-musical-object-mixin)

((elements :accessor elements

:initarg :elements

:documentation "The components"))

(:documentation "A named group of musical objects ordered in time"))
```

To allow the name slot to be filled when we create a compound object, we define some new creators that are similar to the old ones, but have a name as first argument and pass it to make-instance.

```
(defun S (name &rest elements)

"Return a named sequential musical object"

(make-instance 'sequential :name name :elements elements))


(defun P (name &rest elements)

"Return a named parallel musical object"

(make-instance 'parallel :name name :elements elements))


(defun C (name &rest agenda)

"Return a named collection"

(let ((onsets (loop for delta-onset in agenda by #'cddr collect delta-onset))

(elements (loop for element in (rest agenda) by #'cddr collect element)))

(make-instance 'collection :name name :delta-onsets delta-onsets :elements elements)))
```

This allows an example to be defined with some names.

```
(defun named-example ()

"Returns a simple musical object with named structure"

(P "Example" (S "Melody"
(note :duration 1 :pitch "C3")

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

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

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

The `show` method can be patched with an around method in case of named objects if so, it builds a list in the format according to the new creators, otherwise it uses the old ones. An around method ain is the way to get hold of the result returned by our old primary `show` method.

```
(defmethod show :around ((object named-musical-object-mixin))

"Add the constructor and possibly the name to the showing of the elements"

(if (name object)

(list* (named-constructor object) (name object) (rest (call-next-method)))

(call-next-method)))


(defmethod named-constructor ((object sequential)) 's)

(defmethod named-constructor ((object parallel)) 'p)

(defmethod named-constructor ((object collection)) 'c)
```

[to do: of al de functie constructor-name schrijven, goed voorbeeld voor method and combination]

Because we have already bound show into the system printing, the definitions will be used directly for any output of named musical objects.

```
(named-example)

->

(P "Example" (S "Melody"

(note :duration 1 :pitch "C3")

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

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

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

For printing the names in the piano roll notation we need to reserve some space, extending the boundary a bit upward, whenever a compound musical object has a name. An extra before method for `draw-boundary` can print the name at the appropriate place. The new around method is ‘wrapped around’ other possible around methods, in this case the around method that is responsible for proper nesting of the boxes. This makes the new behaviour properly combine with the old, and even the mouse sensitivity and visible selections are made in a consistent way, reflecting the new largere boundaries

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

()

(:documentation "A mixin to print name tags of objects"))



(defmethod boundary :around ((object compound-musical-object) onset (window named-mixin))

"Draw the boundary as side effect of its calculation"

(declare (ignore onset))

(enlarge-rectangle (call-next-method)

(named-margin object window)))



(defmethod named-margin ((object compound-musical-object) (window named-mixin))

"Leave space in a boundary box for the name"

(if (name object)

(list 0 .7 0 0)

'(0 0 0 0)))
```

[to do: this should be like (window-font-height window)]

```
(defmethod draw-boundary :after ((object compound-musical-object) boundary (window named-mixin))

"Draw a label in the box when a compound musical object is named"

(when (name object)

(draw-label window boundary (name object))))
```

[to do: de enlarge aanroep in een method van de algemen window class dan kan deze ook zonder nested]

```
(defclass draw-window (named-mixin

mouse-select-mixin

nested-mixin

structured-mixin

visible-selection-mixin

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

(:documentation "The window class named pianoroll"))


(draw (named-example)) =>
```

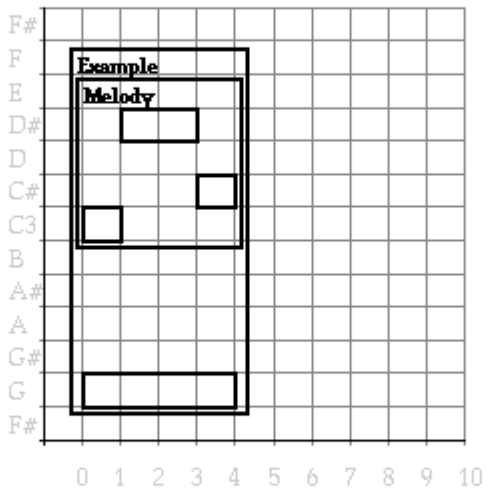

Figure 5. Named musical objects.

On top of the search function we can now define the search for an object with a specific name, e.g. to subject it to a transformation.

```
(defun named (name)

#'(lambda(object onset)

(when (and (name object)

(string-equal name (name object)))

object)))


(draw (named-example))

(front-select (named "Melody")) =>
```

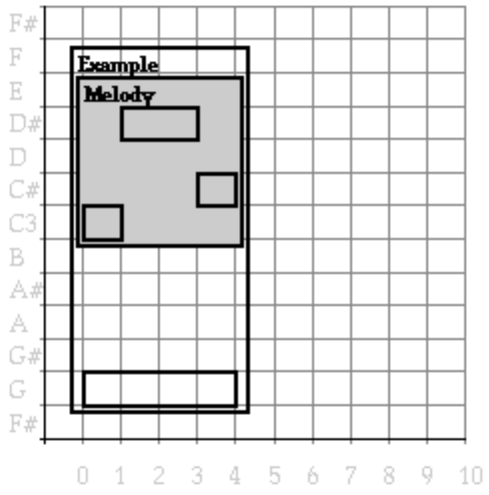

Figure 6. Selection by name

opmerking over before after, draw boundary + selection all draw acties volgorde mixin volgorde geen protocol

initialize instance

In CLOS all the work that is done during creation and initialization of objects is carried out by a few methods (like make-instance) that can be augmented or modified by the programmer. As an example, we will implement a check on the sanity of the compound structures after they is created. The routine that is called by the system after creation of an instance is initialize-instance. It is called with as arguments the object to be initialized and a list of keyword value pairs from (among others) the make-instance call. We will define an after method for our check, remembering to use a compatible lambda list.

```
(defmethod initialize-instance :after ((object compound-musical-object) &rest
arguments)

"Check that each element of a parallel musical object has the same duration"

(declare (ignore arguments))

(loop for element in (elements object)

unless (typep element 'musical-object)

do (error "musical object may not contain a ~A: ~A" (type-of element) element)))
```

This method will be run by the system after creating an instance of the class parallel. When the test on the equalitity of the duration of the components fails, the execution will be halted with an error message. Note how no attempt is made to print the whol musical object: our show method would choke on the element just detected.

```
(parallel (note) 1.3 (pause :duration 3) nil)

=>

> Error: musical object may not contain a double-float: 1.3
```

[to do: dubbele text ?, doorschrijven]

customizing initialization

After CLOS has instantiated a class, but before returning the object as the result of `make- instance`, the object is subjected to the `initialize- instance` method. This method is given also all the initialization arguments presented to `make- instance`. This is the perfect place to add some of our own initialization behavior. For example, we have prevented pitch updates outside the reasonable range, but we haven't prevented the creation of a note with such a pitch. Because our setf around method can deal with the details of clipping the pitch, we only have to give it a chance to do so.

```
(defmethod initialize-instance :after ((object note) &rest args &key pitch &allow-
other-keys)

"Allow the pitch asignment protection to do its work at creation time as well"

(setf (pitch object) pitch))
```

The above will take care of the translation of a pitch name to a midi value as well. The other type of initization preprocessing, the traetment of an initial frequency specification, that we supported in a rather ugly way in de note constructor function, can no also beget a more elegant form. This makes our tinkering with the constructor function itself superfluous: all initialization is handled by the `initialize- instance` after method.

```
(defmethod initialize-instance :after ((object note) &rest args

&key pitch

frequency

&allow-other-keys)

"Allow the pitch asignment protection to do its work at creation time as well"

(when pitch (setf (pitch object) pitch))

(when frequency (setf (frequency object) frequency)))


(defun note (&rest attributes)

"Return a note object with specified attributes"

(apply #'make-instance 'note attributes))
```

```
(note :frequency 440)

->

(note :duration 1 :pitch "A3")
```

Setf methods

Just like the duration of a basic musical object can be updated, it may be necessary to change the duration of compound musical objects, compressing or expanding all their elements. If we were to write a procedure that establishes that, called say `set- duration`, the programmer has to remember that durations of basic musical objects are changed using `setf` on the duration slot, and that the `set-duration` procedure has to be used to change the durations of compound musical objects. It is again best to make a uniform interface for updating, by writing a setf method for duration, specialized for compound musical objects. This method needs to calculate a stretch factor (a ratio between desired and actual duration) and elongates (or shrinks) the duration of all its elements with that factor. The use of mutual recursion between the `(setf duration)` method and the `stretch` transformation comes quite natural here.

```
(defmethod (setf duration) (value (object compound-musical-object))

"Set the duration of a compound musical object by stretching its components"

(let ((factor (/ value (duration object))))

(mapc-elements object #'(lambda(object)(stretch object factor)))

value))


(defmethod stretch ((object musical-object) factor)

"Multiply the duration of a musical object by a factor"

(setf (duration object) (* (duration object) factor))

object)
```

Remember that the order in the argument list of the `setf` method is first the value and then the object and that a `setf` method should always return its value. The `stretch` transformation that changes the duration of a musical fragment is not just an auxiliary function for the `(setf duration)` method, it is useful in itself. In Figure 7 it is applied to the example to yield a fast version.

```
(draw (original-and-transform

(example)

(pause :duration 1)

#'stretch .5)) =>
```

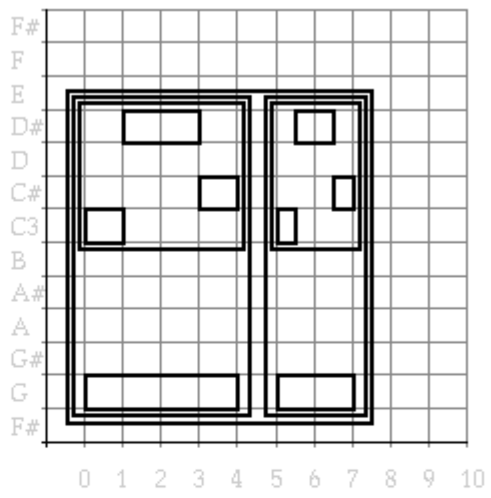

Figure 7. The example and a shortened version.

The definition of a modification macro for multiplications makes the code look more like the other transformations (e.g. `mirror`)

```
(define-modify-macro multf (args) * "Multiply a generalized variable")
```

```
(defmethod stretch ((object musical-object) factor)
"Multiply the duration of a musical object by a factor"
(multf (duration object) factor)
object)
```

The `setf` method for duration defined for compound musical objects works a not well for collections: it shrinks or stretches the duration of all its components, but after that, the onsets of the components, which are stored as data in a collection, need to be adapted as well.

```
(defmethod (setf duration) (value (object compound-musical-object))
"Set the duration of a compound musical object by stretching its components"
(let ((factor (/ value (duration object))))
(mapc-elements object #'(lambda(object)(stretch object factor)))
(setf (delta-onsets object)
(loop for delta-onset in (delta-onsets object)
collect (* delta-onset factor)))
value))

(draw (original-and-transform (example-collection)
(pause :duration 1)
#'stretch .5)) =>
```

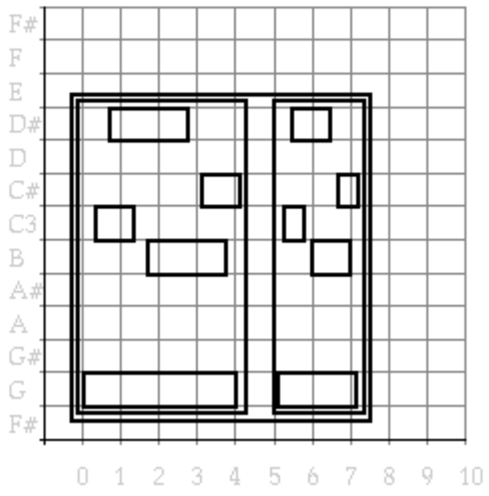

Figure ?. The collection and a shortened version.

Arguments to user commands

The stretch operations seems like a useful addition to our menu of transfomations that can be applied to musical objects in windows. However, it is difficult to find a fixed stretch factor that will satisfy all. We will use one of the turnkey dialogs to have the user enter a stretch factor and use that in the transformation.In case of user input, it is wise to put in some extra checks on the validity of the input.

```
(defmethod stretch-selection ((window selection-mixin)
&optional (factor (get-factor)))
"Stretch the duration of the musical object selected in the window by a factor"
(selection-operation window #'stretch factor))

(defun get-factor ()
"Ask user to type a factor and return it as positive number"
(loop as factor = (read-from-string (get-string-from-user "Factor (> 0)") nil nil)
when (and (numberp factor)(> factor 0))
do (return factor)))
```

For consistency we will also change the mirror operation and add a general transpose operation along the same lines.

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

&optional (interval (get-interval)))

"Transpose the musical object selected in the window by an interval"

(selection-operation window #'transpose interval))


(defun get-interval ()

"Ask user to type an interval in semitones and return it as number"

(loop as interval =

(read-from-string (get-string-from-user "Interval (in semitones)")

nil

nil)

when (numberp interval)

do (return interval)))


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

"Reflect the musical object selected in the window around a center"

(selection-operation window #'mirror (get-pitch)))


(defun get-pitch ()

"Ask user to type an pitch name"

(get-string-from-user "Pitch name"))
```

Now the new commands can be added to the menu and tried.

```
(add-music-command "Transpose" #'transpose-selection-up #\t)

(add-music-command "Mirror" #'mirror-selection #\m)

(add-music-command "Stretch" #'stretch-selection #\S)


(draw (example))
```

```
(front-mouse-click .5 60 nil 1) =>
```

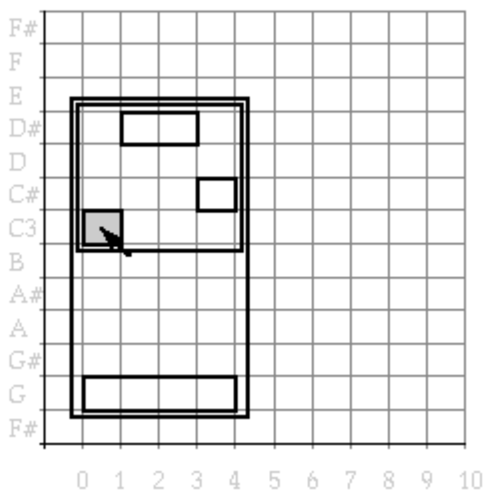

Figure ?. A. Test of the stretch operation

```
(front-draw-window) =>
```

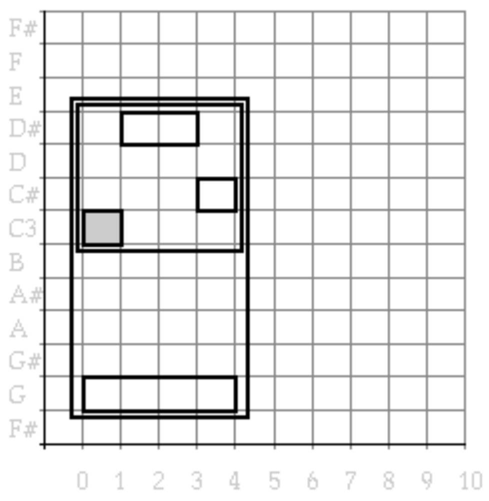

Figure ?. B. Test of the stretch operation

```
(front-draw-window) =>
```

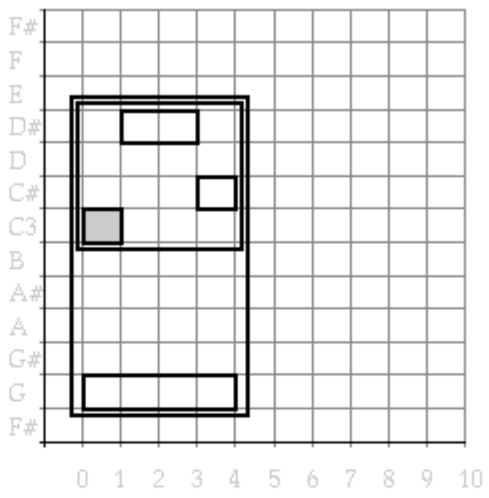

Figure ?. C. Test of the stretch operation

```
(front-draw-window) =>
```

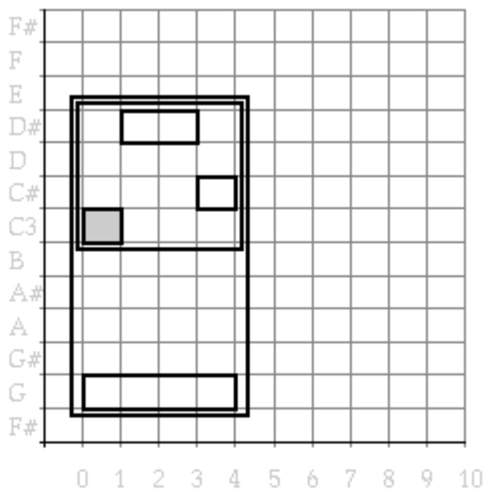

Figure ?. D. Test of the stretch operation

```
(do-front-window #'stretch-selection 2.5)

(front-draw-window) =>
```

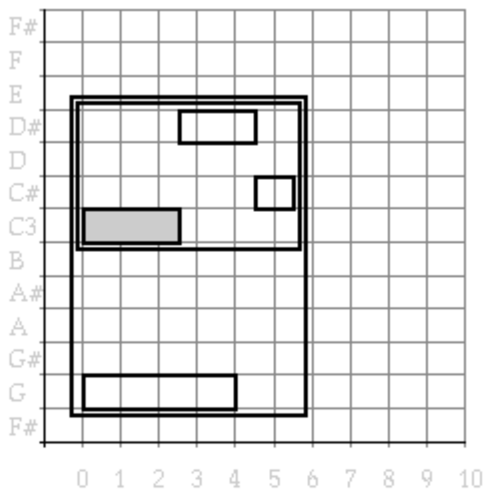

Figure ?. E. Test of the stretch operation

Integer note maken en naar boven schuiven, na mixin, na after around, na after init, voor change window class

All notes would work as well with fraction apitch numbers, niet gelijk zwevende stemmingen etc transpose, intonatie. Maar MIDI kan niet aan, dus moet afronden. later zal met allocatie op kanalen wat verlichten. Support as mixin for notes. To guaranty a proper creation of a midi note, we write an after method for the initialisation

```
(defclass MIDI-note-mixin ()

()

(:documentation "A note mixin which will guarantee an integer pitch number"))


(defmethod initialize-instance :after ((object MIDI-note-mixin) &rest args)

(setf (pitch object) (round (pitch object))))
```

A concrete class and a constructor is easily defined, after which we can check the creation of

```
(defclass MIDI-note (MIDI-note-mixin note)

()

(:documentation "A note which will guarantee an integer pitch number"))


(defun MIDI-note (&rest args)
(apply #'make-instance 'MIDI-note args))

(defun MIDI-note? (object)
(typep object 'MIDI-note))

(pitch (MIDI-note :pitch 60.3))
->
60
```

Again we can write an setf around method to correctly implement any later updating of the pitch too.

```
(defmethod (setf pitch) :around (value (object MIDI-note-mixin))
(call-next-method (round value) object))

(let ((note (MIDI-note)))
(setf (pitch note) 60.3)
note)
->
(note :duration 1 :pitch "C3")
```

Now, one of the wonders of CLOS is the possibility to change the class of an existing object. Below we update the pitch and print the note before and after its class has changed to`MIDI- note`.

```
(let ((object (note)))
(setf (pitch object) 61.9)
(print (pitch object))
(change-class object 'MIDI-note)
(setf (pitch object) 61.9)
(print (pitch object)))
=>
61.9
62
```

To prevent inconsistency we of course need the the proper hooks to have our own programs deal with such complex cases. After CLOS changed the class of an object, it call the shared-initialize method reinitialize ?? This dummy method can of course be decorated with an after method to repair the situation in which a note with a fractional pitch is changed to class MIDI-note. We rely on our setf method to do the rounding. or reinitialize instance for new class ??

```
(defmethod update-instance-for-different-class :after

((old note)(new MIDI-note) &rest initargs)

(setf (pitch new)(round (pitch old))))


(let ((object (note :pitch 60.3)))

(print (pitch object))

(change-class object 'MIDI-note)

(print (pitch object)))

=>

60.3

60
```

```
(defun intonation-example ()

(parallel (sequential (note :duration 1 :pitch 60.2)

(note :duration 2 :pitch 62.9)

(note :duration 1 :pitch 61.3))

(note :duration 4 :pitch 55.7)))


(draw (intonation-example)) =>
```

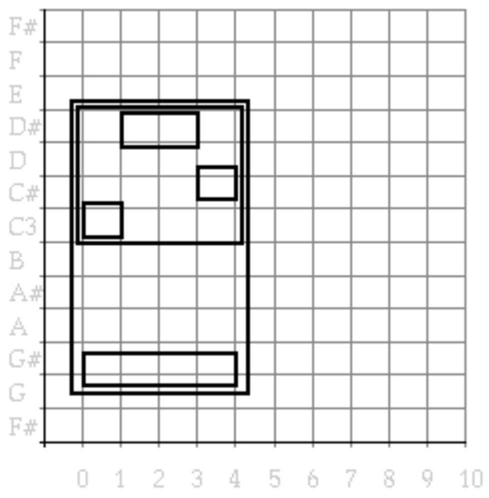

Figure ?. Use of fractional pitches

Changing notes to midi notes and vise versa by the suer can be supported by a menu command workining on the notes in a selection.

```
(defmethod change-note-class ((object compound-musical-object) class)

(mapc-elements object #'change-note-class class))


(defmethod change-note-class ((object note) class)

(change-class object class))


(defmethod change-note-class ((object pause) class)

)
```

(add-music-command “Change to MIDI” #’midi-selection #) (add-music-command “Free from MIDI” #’unmidi-selection #)

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

(when (selection window)

(change-note-class (selection window) 'MIDI-note)))


(defmethod unmidi-selection ((window draw-window))

(when (selection window)
(change-note-class (selection window) 'note)))
```

eventueel ook anders eruit zien op midi-awareness-mixin

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

()

(:documentation "A mixin to show distinction between MIDI and free notes"))


(defmethod draw-boundary ((object note) boundary (window midi-awareness-mixin))

(cl-user::draw-round-rectangle window boundary 15 2 ))


(defmethod draw-boundary ((object MIDI-note-mixin) boundary (window midi-awareness-
mixin))

(draw-rectangle window boundary))


(defclass draw-window (midi-awareness-mixin

named-mixin

mouse-select-mixin

nested-mixin

structured-mixin

visible-selection-mixin

piano-roll-contents-draw-window)

()

(:documentation "The window class named pianoroll"))
```

[to do: als selected dan ook afgerond grijs]

[to do: met mooie muis click windows]

```
(draw (intonation-example))

(front-mouse-click 3.6 61.2 nil 1) =>
```

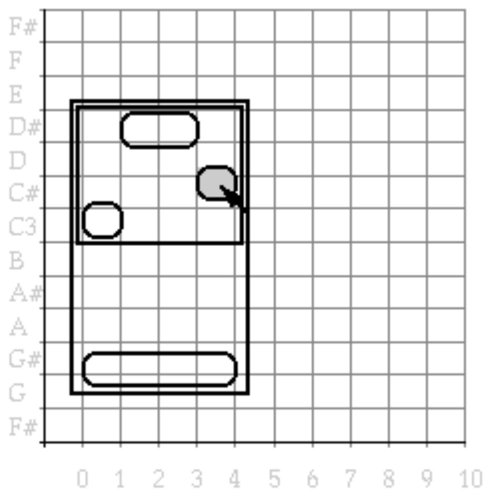

Figure ?. A. changing the class of the notes in the selection

```
(front-draw-window) =>
```

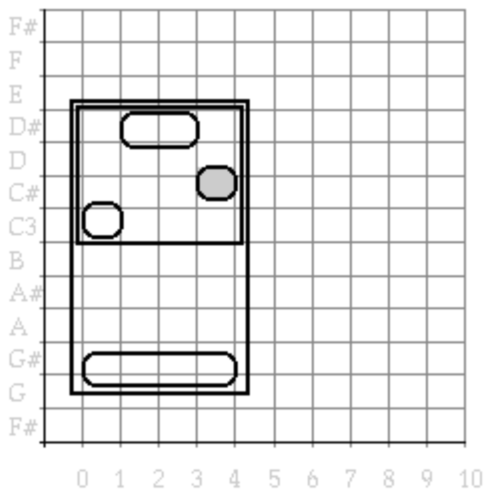

Figure ?. B. changing the class of the notes in the selection

```
(front-selection-operation #'change-note-class 'MIDI-note) =>
```

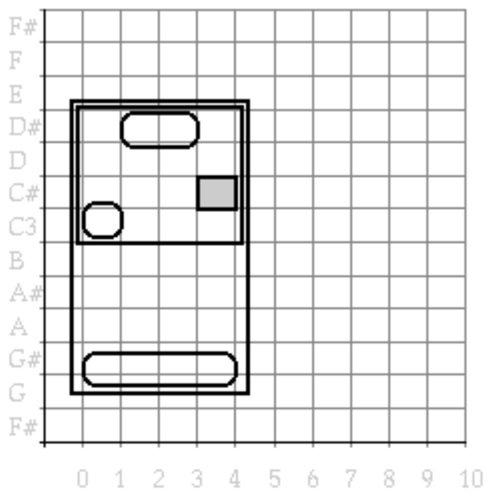

Figure ?. C. changing the class of the notes in the selection

Note how for midi notes on a midi-awareness-mixin the draw-boundary method shadows the less specific method for any note which in turn shadows the method for notes on ... window (because the midi-awareness-mixin appear before the ... window in the list of superclasses of the draw window.

Meta Object Protocol: introspection allows objects to know their class

[to do: in dit stukje een Marx grapje]

Recall the two show methods for compound musical objects:

```
(defmethod show ((object sequential))

"Return a list describing a sequential musical object with its components"

(cons 'sequential

(mapcar #'show (elements object))))


(defmethod show ((object parallel))

"Return a list describing a parallel musical object with its components"

(cons 'parallel

(mapcar #'show (elements object))))
```

They are identical, apart from the name of the class that is put in the front of the list. If we had a way to get hold of the name of the class of an object, one method would suffice. In pure CLOS that is not possible: the so-called programmer interface does not allow explicit access to constructs like classes.

However, part of the beauty of CLOS lies in the so called meta-object protocol. At this level all CLOS constructs (classes, methods etc.) are again implemented as CLOS objects, and thus accessible for the programmer. A simple use of this layer, the so called introspective part, only retrieves innocent objects, like names of classes of objects for use by the programmer. These facilities will be explored a bit first.

```
(defmethod show ((object compound-musical-object))

"Return a list describing a compound musical object with its components"

(cons (class-name (class-of object))

(mapcar #'show (elements object))))
```

Another introspective use of the meta object protocol is to retrieve an inheritance tree of a class. So we are truely having a program inspect the structure of our programs. The `class- subclasses` is by the way also the function used to generate he inheritance diagrams in this chapter automatically.

```
(defun class-subclasses (class)

"Return the tree of classes inheriting from this one"

(cons (class-name class)

(loop for sub-class in (class-direct-subclasses class)

collect (class-subclasses sub-class))))
```

The function must be called on a true class meta object, not a class name. The `find- class` function finds such a class with a given name if it is defined.

```
(pprint (class-subclasses (find-class 'musical-object)))

=>

(musical-object

(compound-musical-object

(collection)

(parallel)

(sequential))

(basic-musical-object

(pause)

(note)))
```

Nice to inspect our growing window class hierarchy, see shaded not used in draw window because shading used for selection, how both mouse-select and visible-select supply selection mixin behavior to draw window

```
(pprint (class-subclasses (find-class 'draw-window-mixin)))

=>

(draw-window-mixin

(named-mixin (draw-window))

(nested-mixin (draw-window))

(combine-boundary-mixin

(structured-mixin (draw-window)))

(shaded-mixin)

(selection-mixin

(mouse-select-mixin (draw-window))

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

when a flat list is needed, no tree, easy to adapt the function

```
(defun all-class-subclasses (class)

"Return the tree of classes inheriting from this one"

(cons (class-name class)

(remove-duplicates (loop for sub-class in (class-direct-subclasses class)

append (all-class-subclasses sub-class)))))
```

(all-class-subclasses (find-class ‘draw-window-mixin)) (draw-window-mixin named-mixin nested-mixin combine-boundary-mixin structured-mixin shaded-mixin selection-mixin mouse-select-mixin visible-selection-mixin draw-window)

Next to introspection, the meta object protocal can be put to extremely powerful use in extending the CLOS language itself, as will be the topic of the next section.

Meta Object Protocol: mixing class cocktails

Somehow it feels not completely right to have to redefine our draw window each time we have invented a new mixin for it. The fact that CLOS only supports creating an instance from a predefined class prevents us here from being able to create an object of a list of ingredient classes directly. Because we programmers always try to push the limits of the language we have to work with, let us try some clever but dirty kludges to achieve our class coctails, before the use of the meta object protocol that was designed for elegant ways of pushing the language boundaries.

The first realisation is that a defclass form is just a lisp list, which we know very well how to construct. And evaluating a class definition created by our program should yield a new class accoring to our wishes. We will use `gensym` to have lisp invent a name for it and return it us. [8]

```
(defun define-cocktail-class (ingredients)

(let ((name (gensym)))

(eval (list 'defclass name ingredients ()))

name))
```

Now we need to instantiate an object directly from a cocktail, passing on any initialisation arguments needed.

```
(defun make-cocktail-instance (ingredients &rest args)

(apply #'make-instance (define-cocktail-class ingredients) args))
```

Because make-instance itself is also a method that can be specialised, we can do a bit better in plain CLOS, by specializing its first class argument to a list. In that way we have supplied a clean interface to the user of class cocktails, even though our implementation is still lousy.

```
(defmethod make-instance ((ingredients list) &rest args)

(apply #'make-instance (define-cocktail-class ingredients) args))


(draw (named-example) :window-class '(named-mixin

nested-mixin

structured-mixin

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

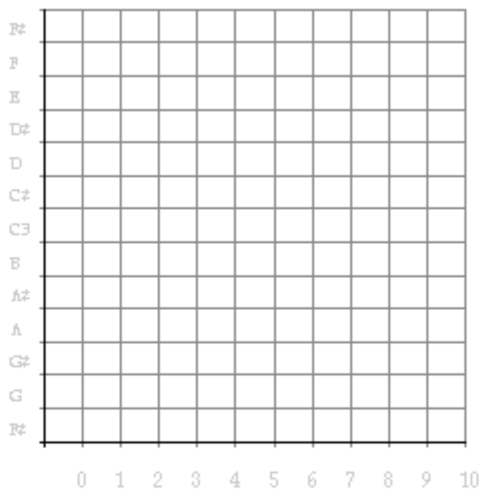

Figure ?. A. Display of the same example on different cocktail windows

Figure ?. B. Display of the same example on different cocktail windows

Now we can proceed to define a CLOS class in a neatly programmed fashion. Below the layer of macro's (like defclass) that are intended to have constant arguments written by the programmer and appearing at the top level in the program, there is a layer of functions which do the real work, and which can of course be called by use directly to construct a class definition under program control. Because classes are, at the meta level, just ordinary CLOS objects, they can be created by make inststance as well. [9]That can also be implemented easily using the metaobject protocol. The class of a class is called `standard- class`. To create an instance of that meta class we have to supply, next to its name, a list of classes of which inherits. Remember that on this level we are always dealing with class meta objects: class names do not suffice.

```
(defun define-cocktail-class (ingredients)

(make-instance 'standard-class

:name (gensym)

:direct-superclasses (mapcar #'find-class ingredients)

:direct-slots nil))

(defun define-cocktail-class (ingredients)

(let* ((name (intern (string (gensym "COCKTAIL-"))))

(class (make-instance 'standard-class

:name name
:direct-superclasses (mapcar #'find-class ingredients)

:direct-slots nil)))

(setf (find-class name) class)

name

))
```

To create a window class of our choice, we could now program a menu. Since all our drawing mixins inherit from `draw- window- mixin` we can lok those up and thus even make the menu dynamic: when new draw mixins arise they will automatically appear in the menu. For extra juicyness we display not the name of the class but its documentation.

```
(defun select-direct-subclasses (class)

(mapcar #'first

(select-item-from-list

(loop for class in (class-direct-subclasses (find-class class))

as name = (class-name class)

collect (list name (documentation name 'type)))

:window-title "Ingredients"

:selection-type :disjoint

:table-print-function #'(lambda(item stream)(princ (second item) stream)))))


(defun get-window-cocktail ()

(define-cocktail-class (append

(select-direct-subclasses 'draw-window-mixin)

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

use command to multiple select

```
(draw (example) :window-class (get-window-cocktail))
```

[to do: plaatje van het menu]

The class cocktail mixer cannot only be used to create new window classes, an existing window can also be instructed to change it class. Thus we can alter the appearence of our musical objects on the fly.

```
(defmethod window-change-class ((object draw-window-mixin) &optional (class (get-
window-cocktail)))

(change-class object class))



(add-music-command "Change appearence" #'window-change-class)
```

[to do: refinements, betere gensym, documentation ophalen en mixen, voorbereiden voor een maal instantieren. zoeken of ie er al is shared initialize, bv de selectie weer op all zetten of zoiets window title in menu zetten menu met checkboxen maken]

[to do: opmerking over gebruik change-class niet voor note wel voor window]

Adding behavior to class definitions

Repetitive definitions, constructor of same name and type def change the way the class is defined, by having it not being an instantiation of standard class but of our own meta class

```
(defclass our-class (standard-class)())



(defmethod initialize-instance :after ((object our-class) &rest args

&key name &allow-other-keys)

(make-constructor name name)

(make-type-predicate (intern (string-upcase (format nil "~A?" name))) name))
```

[to do: hoe kemen we hier aan de extra initargs die bv de naam van predicaat en constructor kunnen doorgeven uit de defclass form (defclass note (basic-musical-object)() (:constructor ‘note :type-predicate ‘note?))]

```
(defun make-constructor (name class)

(when name

(define name

#'(lambda(&rest args)(apply #'make-instance class args)))))



(defun make-type-predicate (name class)

(when name

(define name

#'(lambda(object)

(typep object class)))))
```

```
;(trace define)
```

following warns user redefinition, if confirmed new class with auto type and constructor

```
(defclass note (basic-musical-object)
((pitch :accessor pitch
:initarg :pitch
:initform 60
:documentation "Pitch in MIDI numbers"))
(:documentation "A pitched note")
(:metaclass our-class))
```

should pass initargs name for accessor, visible, but refine left for user (see book MOP)

A defclass form mentioning our meta class instead of defaulting to `standard- class` will result in the creation of a proper type predicate and a constructur function. E.g:

In this way we have effectively extended the CLOS language itself while programming in CLOS. This does not mean that mofdularity and transparencey is violated: our changes to the language cannot affect other parts of the program - only classes defined to a of our metaclass exhibit the new behavior.

We hope that with the above short introduction to the meta object protocol the reader need not be convinced anymore of the power and beauty of CLOS. We will now work on some more mundane issues of object oriented programming, to return later to the this meta-sphere.

Eql specialization, moet voor mouse-click

specialize soms op een bv eql nil afvangen midi note on = 0 modifier keys, click# commandos run :window :object :selected

mop one/few instance classes

use in allocate note, midiport, i/o-format en natuurlijk class cocktail (een tequila sunsire is a tequila sunrise) identiteit

```
(defclass one-instance-only-mixin ()())
```

```
(defvar *one-instance-only-table* (make-hash-table))


(defmethod make-instance :around (class &rest args)
(cond ((null (member 'one-instance-only-mixin
(class-direct-superclasses (find-class class))))
(call-next-method))
((gethash class *one-instance-only-table*))
(t (setf (gethash class *one-instance-only-table*)
(call-next-method)))))


(defclass test1 (one-instance-only-mixin)())
(defclass test2 (one-instance-only-mixin)())


(make-instance 'test1)
```

Standard method combination recap

around before after + diagram + reeds gedaan example beschrijven

build in and/append method combination

Non standard but built in, other ways in which methods are combined e.g. look at 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)))
```

could not share the fact that for basic musical objects the duratiuons should be the same. Can as around is exercise

But more natural, and combination of results of methods. first undefine. then notify a special method combination to be used to the generic function

```
(fmakunbound 'same?)

(defgeneric same? (object1 object2)
(:method-combination and))
```

now can share code, and run duration automatically, even for new basic objects to be defined

```
(defmethod same? and ((object1 basic-musical-object)(object2 basic-musical-object))
"Are the two durations are equal?"
(= (duration object1) (duration object2)))

(defmethod same? and ((object1 note)(object2 note))
"Are the two pitches equal?"
(= (pitch object1) (pitch object2)))
```

(same? (note :duration 1 :pitch 50) (note :duration 2 :pitch 50)) -> nil

Other definitions for compounds, only one method for now

```
(defmethod same? and ((object1 sequential)(object2 sequential))
"Are two sequential musical objects equal?"
(same-list (elements object1)(elements object2) :test #'same?))

(defmethod same? and ((object1 parallel)(object2 parallel))
"Are two two parallel musical objects equal?"
(same-set (elements object1)(elements object2) :test #'same?))

(defmethod same? and ((object1 collection)(object2 collection))
"Are two two parallel musical objects equal?"
(and (equal (delta-onsets object1)(delta-onsets object2))
(same-set (elements object1)(elements object2) :test #'same?)))
```

the default same? method for werkt niet meer need catch type different methods

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

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

(eql (type-of object1)(type-of object2)))
```

(same? (example)(example)) -> t - (same? (example)(note))

But remember our named mixin,is really wrong upto now

(same? (s “Melody1” (note :duration 1 :pitch “C3”) (note :duration 2 :pitch “D#3”) (note :duration 1 :pitch “C#3”)) (s “Melody2” (note :duration 1 :pitch “C3”) (note :duration 2 :pitch “D#3”) (note :duration 1 :pitch “C#3”))) -> t

[to do: leuk voorbeeld zelfde noten andere naam]

but easy to repair

```
(defmethod same? and ((object1 named-mixin)(object2 named-mixin))

(string-equal (name object1) (name object2)))
```

append build in method combination, na mop class-of

more than and, append, can be use not check attributes but collect descriptions of them e.g. in show. anny later mixins which add attributes to say noets will supply an extra method

```
(fmakunbound 'show)


(defgeneric show (object)

(:method-combination append)

(:documentation ""))


(defmethod show append ((object basic-musical-object))
"Return a list describing the duartion of a musical object"

(list :duration (duration object)))


(defmethod show append ((object note))

(list :pitch (midi-number-to-pitch-name (pitch object))))


(defmethod show append ((object sequential))

"Return a list describing a sequential musical object with its components"

(mapcar #'show (elements object)))


(defmethod show append ((object parallel))

"Return a list describing a parallel musical object with its components"

(mapcar #'show (elements object)))


(defmethod show append ((object collection))

"Return a list describing a parallel musical object with its components"

(loop for delta-onset in (delta-onsets object)

for element in (elements object)

collect delta-onset

collect (show element)))


(defmethod show append ((object musical-object))

(list (class-name (class-of object))))
```

proberen, surprise

```
(example)

->

(((:pitch "C3" :duration 1 note)

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

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

(:pitch "G2" :duration 4 note)

parallel)
```

most specific first, but can control, werkt niet

```
(defgeneric show (object)

(:method-combination append :most-specific-first)

(:documentation ""))

(defgeneric show (object)

(:method-combination append :most-specific-last)

(:documentation ""))
```

behavior named just simple method + around voor naam

```
(defmethod show :around ((object named-musical-object-mixin))

"Add the constructor and possibly the name to the showing of the elements"

(if (name object)

(list* (named-constructor object) (name object) (rest (call-next-method)))

(call-next-method)))
```

defined method combination

add-boundaries + margins enlarge

```
(defmethod boundary :around ((object compound-musical-object) onset (window named-
mixin))

"Draw the boundary as side effect of its calculation"

(declare (ignore onset))

(enlarge-rectangle (call-next-method)

(named-margin object window)))
```

```
(defmethod boundary :around ((object compound-musical-object) onset (window nested-
mixin))

"Draw the boundary as side effect of its calculation"

(declare (ignore onset))

(enlarge-rectangle (call-next-method)

(nested-margin object window)))
```

eigenlijk te veel, bedoeld voor margin optellen alle mixin een margin mag me mixen allemaal applicable combination = eigen method

```
(fmakunbound '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)

(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 boundary :around ((object compound-musical-object) onset (window draw-window)) ; eigen basic
""
(declare (ignore onset))
(let ((boundary (call-next-method)))
(when boundary
(enlarge-rectangle boundary
(margin object window)))))
```

margin methods, add margin to combine, define

```
(defun add-margin (&rest margins)
(apply #'mapcar #'+ margins))
```

make known

```
(define-method-combination add-margin)
```

define it is the combination way

```
(defgeneric margin (object window)
(:method-combination add-margin)
(:documentation ""))
```

and define methods with that combination

```
(defmethod margin add-margin ((object musical-object) (window nested-mixin))
"The margins of a nested compound boundary box"
(let ((horizontal .15)(vertical .2))
(list (- horizontal) vertical horizontal (- vertical))))

(defmethod margin add-margin ((object compound-musical-object) (window named-mixin))
"Leave space in a boundary box for the name"
(if (name object)
(list 0 .7 0 0)) ; uit window height
'(0 0 0 0))

(draw (example)) =>
```

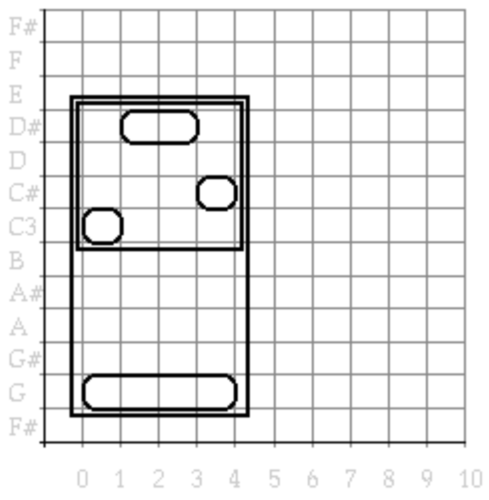

Figure ?. test

[to do: ook voor loudness mixin]

Loudness mixin voor note waar ??
Backpointers

When creating a musical object as part of a compound one, the compound object has access to its parts because these occur in its elements slot. But access the other way: from a musical object to the encompassing compound one is not supported. Such acces is needed to implement, e.g., deletion of a musical object: it has to be removed from the list of elements of its containing compound musical object, if it has one. [10]To program such access, let us first add a slot to a musical object that can contain the reference to a containing compound object.

```
(defclass musical-element ()

((element-of :accessor element-of :initarg element-of :initform nil))

(:documentation "T"))


(defclass musical-object (musical-element)()

(:documentation "The abstract root class of all musical objects"))
```

By supplying an initform value to the slot descriptor, the `element- of` slot will default to `nil` when not specified otherwise. This will be the proper value for a top-most musical structure, which is not an element of anything. But when a compound musical structure is created, incorporating an musical object as its element, this element has to be given a proper backpointer. We will define an after-method for this backpointer initialization, remembering to use a compatible lambda list.

```
(defmethod initialize-instance :after ((object compound-musical-object) &rest
arguments)

"Install backpointers in each component of a compound musical object"

(declare (ignore arguments))

(loop for element in (elements object)

do (setf (element-of element) object)))
```

Now the part-of links between musical objects can be traversed in two ways (see Figure ?).

[to do: backpointers in tree draw program voor het example object, vgl met figure ??]

maintaining back-pointer consistency

never worry, bit expensive later trim. dangling pointers always danger the elements of a compound musical structure change, as we did in the contraction example, the backpointers are in danger to be left dangling ??, i.e. pointing to wrong or deleted musical objects. The backpointer of an object should be realligned to point to the object of which it is an element. Instead of programming this explicitly in each program that modifies the musical structure, it is better to make the primitive modification action, a change of the elements slot, behave such that it will leave a consistent structure. These changes will all be made with the call of a setf on the elements slot of a compound musical structure, and any repair of backpointers can be done in a after method on this setf. In this way all structure modifying transformations will automatically leave consistent backpointers. When storing redundant information is chosen as a method for implementation - as we do for backpointers, but e.g. not for onset times of musical objects - acces function that change the structure are responsible for repairing any inconsistent state that may result. MOET beter in acces

```
(defmethod (setf elements) :before (elements (object compound-musical-object))

(loop for element in (elements object)

do (setf (element-of element) nil)))


(defmethod (setf elements) :after (elements (object compound-musical-object))

(loop for element in elements

do (setf (element-of element) object)))
```

Destructive operations on the stucture of displayed musical objects

Using this new backpointer facility we can write a method to remove an object: we just remove all the references to the object in the object's container:

```
(defmethod clear-object ((object musical-object))

(let ((container (element-of object)))

(setf (elements container)

(remove object (elements container)))))
```

We will use this method for `clear- selection` that will suppor clearing selections in draw-windows that have a selection-mixin. It takes care of three situations. The first is no selection at all, then there is nothing to clear. Second, when the whole object is selected (tested with the predicate `all- selected?`) we remove the object from the window and deselect the selection. Finally, the case were a sub-selection is made, we remove only this object using `clear- object` that handles all references correctly.

```
(defmethod all-selected? ((window selection-mixin))

"Return T if selection is the whole musical object, false otherwise"

(eql (selection window) (object window)))


(defmethod no-selection? ((window selection-mixin))

"Return T if there is no selection, false otherwise"

(eql (selection window) (object window)))


(defmethod clear-selection ((window selection-mixin))

"Clear selection from window"

(cond ((no-selection? window) nil)

((all-selected? window)

(setf (object window) nil)

(deselect window))

(t (clear-object (selection window)))))


(draw (example)) =>
```

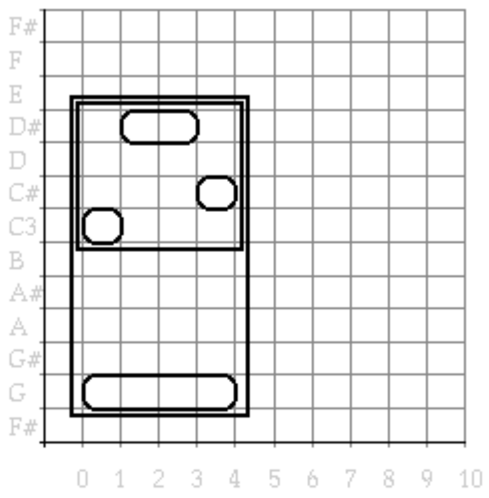

Figure ?. A. Clearing a selection.

```
(front-mouse-click 1 58 nil 1) =>
```

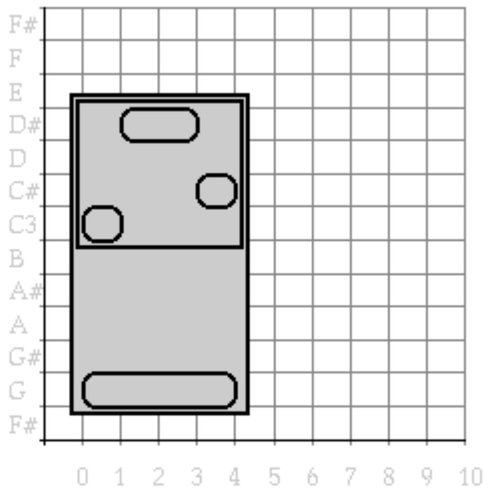

Figure ?. B. Clearing a selection.

```
(do-front-window #'clear-selection)

(front-draw-window) =>
```

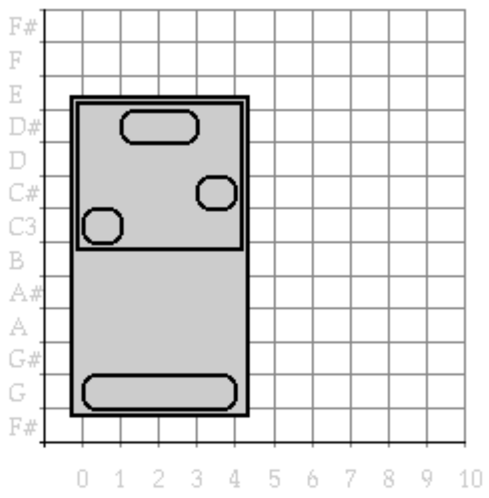

Figure ?. C. Clearing a selection.

The other situation in which we have to keep the backpointers consistent is when we want to replace a selection with another music-object. For this we write a `replace- object` function that replaces an musical-object (the first argument) with another (the second argument). The method replace-selection uses it.

```
(defmethod replace-object ((orginal musical-object)(replacement musical-object))

"Return object with replacement, adjusting backpointer accordingly"

(loop for element in (elements (element-of orginal))

when (eql element orginal)

collect replacement

else collect element))
```

```
(defmethod replace-selection ((window selection-mixin)(object musical-object))

"Replace selection in window with musical-object"

(cond ((no-selection? window) nil)

((all-selected? window)

(setf (object window) object)

(deselect window))

(t (replace-object (selection window) object))))
```

Furthermore, we can write transformations that work appropiately on selections.

```
(defmethod transform-selection ((window selection-mixin) transform &rest args)

(when (selection window)

(replace-selection window (apply transform (selection window) args))))


(defmethod repeat-selected ((window selection-mixin) &optional (n 2))

(transform-selection window #'repeat))


(defmethod canon-selected ((window selection-mixin))

(transform-selection window #'canon))



(add-music-command "Canon selected" #'canon-selected #\C)

(add-music-command "Repeat selected" #'repeat-selected #\R)


(draw (example)) =>
```

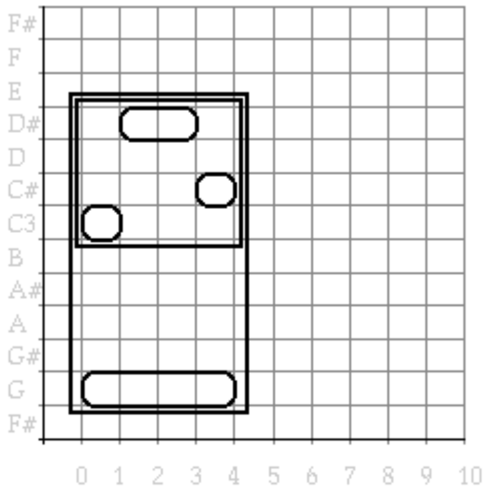

Figure ?. A. Transforming a selection.

```
(front-mouse-click 1 58 nil 1) =>
```

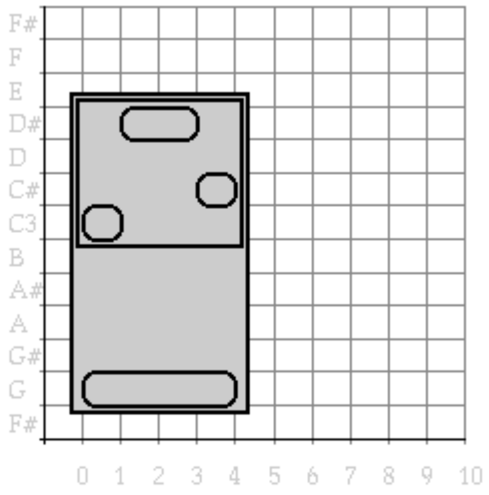

Figure ?. B. Transforming a selection.

```
(do-front-window #'repeat-selected)

(front-draw-window) =>
```

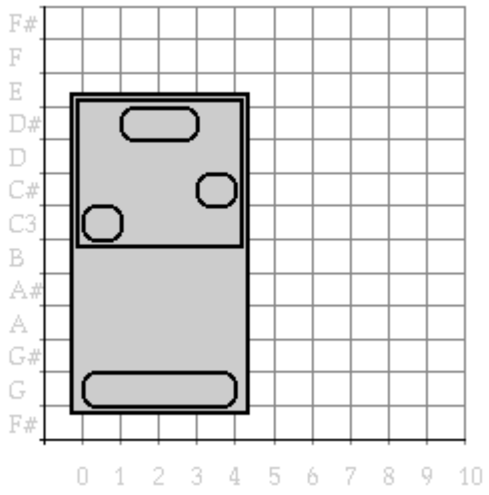

Figure ?. C. Transforming a selection.

[to do: menu extra keywords? enablep op window, op selectie, op non-selecte, ...]

A clipboard for musical objects

We now can make different kind of selections (`clear- selection`, `replace- selection`, etc), in a programmed fashion (using,e.g., :lisp do-front-window

) and directly on the window, with mouse clicks and command-keys. Now assume we have two of these windows. Than we need a way to transfer this information to another window. When programming, we can bind the objects (e.g., in a `let`), when using control keys we need another way of (temporary) storing information. The standard Macintosh way to do this is via the clipboard, a data structure containing information that was cut or copied (using the standard Cut and Copy edit commands) and to make it available for pasting somewhere else (You probably use it all the time for editing text when writing code in your editor). The functions `put- scrap` and `get- scrap` are available for this. [11] communicate information to the clipboard. They need the type of the information and, in the case of `putscap`, the data to put on the clipboard.

```
(put-scrap :text "A string of text.") ->

"A string of text."


(get-scrap :text) ->

"A string of text."
```

We now want to make copying, pasting and cuting of musical-objects from and to windows as easy as text copying is. For this to work, we have to tell the system about musical-objects as a new type that can be handled by the clipboard and define our own scrap-handler (the program that deals with clipboard information). We define first are own class `clipboard`, inheriting from (MCL's) `scrap- handler`. Then we define our own specialized :lisp musical-object-clipboard

that can deal with musical-objects. An initialize-instance after-method tells the system about this new scrap-handler, using the default initialization argument. **[to do: import`ccl::add- scrap- handler` ?, uitleg after default-initargs eerder al gedaan hebben]**

```
(defclass clipboard (scrap-handler) ()

(:documentation "scrap-handler of for musical-objects"))


(defclass musical-object-clipboard (clipboard) ()

(:default-initargs :scrap-type :musical-object)

(:documentation "scrap-handler of type :musical-object"))


(defmethod initialize-instance :after ((object clipboard) &rest args &key scrap-type)

(ccl::add-scrap-handler scrap-type object))


(make-instance 'musical-object-clipboard)
```

We can now get and put our musical-objects from and to the clipboard. Note that true musical-objects are put on the clipboard, not just their printed represention.

```
(put-scrap :musical-object (note)) ->

(note :duration 1 :pitch "C3")


(get-scrap :musical-object) ->

(note :duration 1 :pitch "C3")
```

Standard edit operations on window with musical objects

Of course, we want use the musical-object clipboard type with the standard Macintosh edit facilities. For example, copy an object from a window and paste it in another. First, we will write a set of edit functions, using the previously defined selection functions and our new clipboard facility for musical-objects.[12]

```
(defmethod copy ((window selection-mixin))

"Set scrap to current selection in window"

(put-scrap :musical-object (selection window)))


(defmethod clear ((window selection-mixin))

"Remove current selection from window"

(clear-selection window))


(defmethod cut ((window selection-mixin))

"Set scrap to current selection and remove selection from window"

(put-scrap :musical-object (selection window))

(clear-selection window))


(defmethod paste ((window selection-mixin))

"Paste musical-object from scrap in window"

(replace-selection window (get-scrap :musical-object)))
```

Second, we have to tell our draw-window if, and in what situation these operations are allowed. The Lisp system checks whether the common macintosh edit actions (like `select- all`) are supported for a specific window that happens to be the front window, and in that case the corresponding menu item, and the corresponding command key are enabled. By writing a `window- can- do- operation` method specifically for the `selection- mixin`, we can use,for example, the command-C keystroke and the “Copy” menu item under the edit menu on any window whose class is based on the `selection- mixin`.

This is a way in which system code was prepared to be extended: it uses a method which is defined for the genaral default case (`window- can- do- operation` returns :lisp nil

for arbitrary windows), and any user-code can define specialized methods for it. Below we define the method for the four stand edit operations. Check the situations in which they are enabled:

```
(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'copy))

&optional item)

"Can copy when window has a musical-object selected"

(selection window))


(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'clear))

&optional item)

"Can clear when window has a musical-object selected"

(selection window))


(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'cut))

&optional item)

"Can cut when window has a musical-object selected"

(selection window))


(defmethod window-can-do-operation ((window selection-mixin)

(operation (eql 'paste))

&optional item)

"Can paste when a musical-object is on the clipboard"

(get-scrap :musical-object))
```

Before we will try our newly defined edit operations, we will add another function to our music -menu: a function to make a new and empty piano-roll window: **[to do: add-music-command needs extra arg: menu-item should always be enabled]**

```
(defmethod new-draw-window ((window selection-mixin) &rest args)
(apply #'draw nil args))
```

```
(add-music-command "New Draw Window" #'new-draw-window #\N)
```

Now we will make two windows, one with our familiar example, the other empty.

```
(draw (example) :window-title "draw window A") =>
```

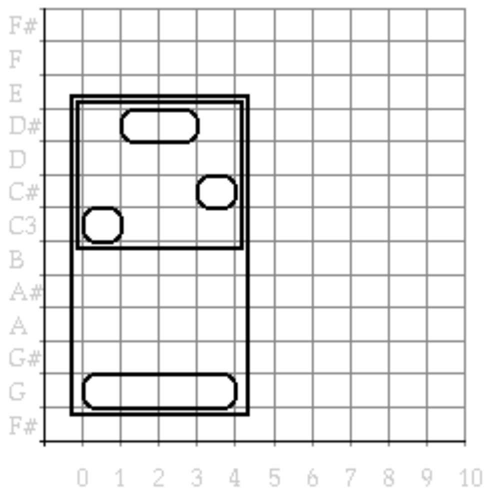

Figure ?. A. Make two windows

```
(draw nil :window-title "draw window B" :view-position (make-point 250 40)) =>
```

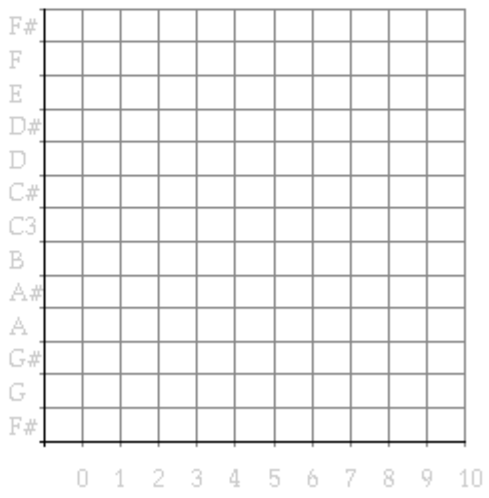

Figure ?. B. Make two windows

In order to switch, in a programmed way (running the code you can of course select one or the other window by clicking on it), between the one and the other draw-window we define the following function:

```
(defun second-draw-window ()

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

(let ((window (second (windows :class 'draw-window))))

(when window

(window-select window)

window)))
```

In the next figures we can see how we can use our newly defined edit operations.

```
(second-draw-window)

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

(front-draw-window) =>
```

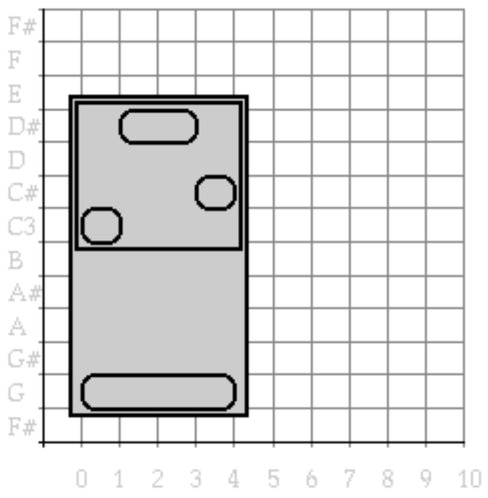

Figure ?. A. Edit opereration on windows

```
(do-front-window #'copy)
```

```
(front-draw-window) =>
```

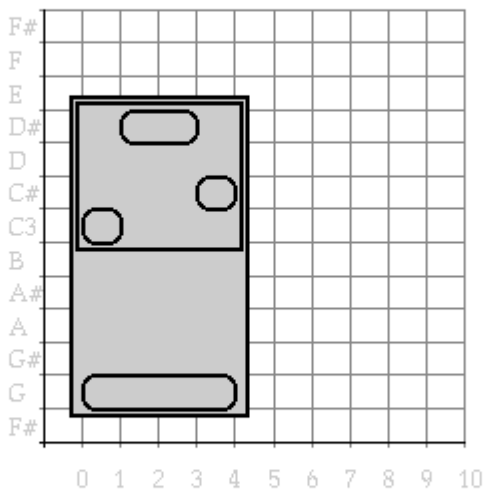

Figure ?. B. Edit opereration on windows

```
(second-draw-window)

(do-front-window #'paste)

(front-draw-window) =>
```

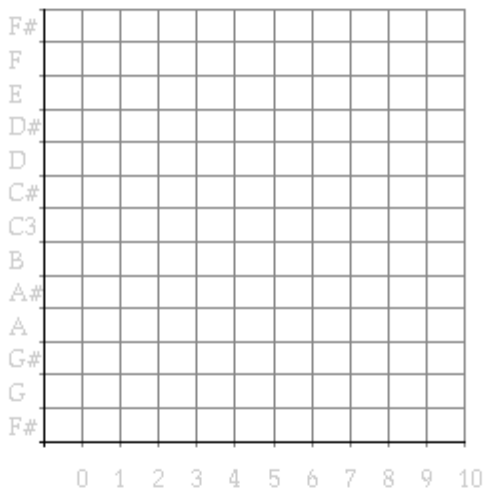

Figure ?. C. Edit opereration on windows

If you start combining these methods with transformation you will soon find out that our idea of copying did not really work. We copied a reference co musical-object to the scrap and, consequently in to the other window. When (destructively) changing the musical object in window A, it will also change in window B.

```
(second-draw-window)

(front-window-operation #'transpose 3)

(front-draw-window) =>
```

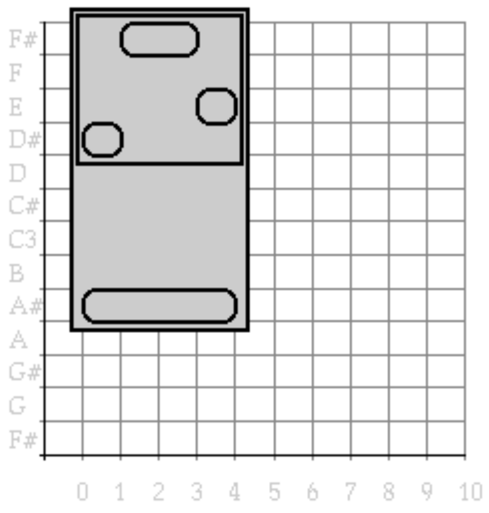

Figure ?. A. Unforeseen side-effects

```
(second-draw-window)

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

(front-draw-window) =>
```

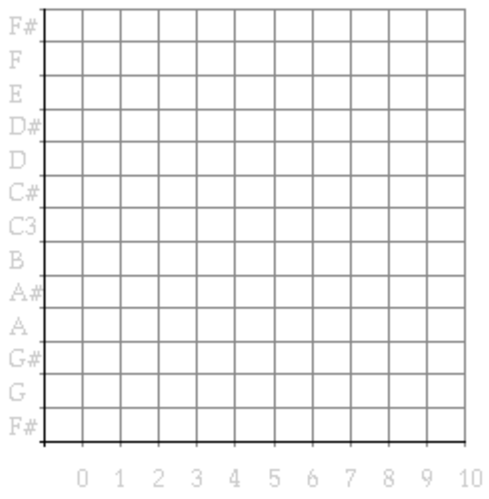

Figure ?. B. Unforeseen side-effects

To prevent this from happening we will copy the object before putting it on the clipboard.

```
(defmethod copy ((window selection-mixin))

"Set scrap to current selection in window"

(put-scrap :musical-object (copy-object (selection window))))


(defmethod cut ((window selection-mixin))

"Set scrap to current selection and remove selection from window"

(put-scrap :musical-object (copy-object (selection window)))

(clear-selection window))
```

But, also when pasting in different windows, each possibly transforming this object again, we need to make a copy from the object when obtainign it from the scrap (which actually is a bug in the clipboard system: it is not an object that is copes, but just refered to.copy but refering that is done).[13]

```
(defmethod paste ((window selection-mixin))

"Paste musical-object from scrap in window"

(replace-selection window (copy-object (get-scrap :musical-object))))
```

Open musical objects

We will integrate the edit operation on musical object a little bit further in the Macintosh environment by providing a open-selection musical-object. We add it as “Open Selection” menut item under our Music menu, and make it available as the action to be done when doublke-clicking on a msucial-object: it will open a new window (of the same class and related title) containing only the selection. Note, that for now, `open- object` copies musical objects, as such updates on the master window will not affect the contents of the newly opened window (for the same reason described for clipboards). [14]

```
(defmethod open-selection ((window selection-mixin))

(open-object (selection window)

:window-class (class-of window)

:window-title (format nil "Part of ~A" (window-title window))

:view-position (add-points (make-point 20 20) (view-position window))))
```

```
(defmethod open-object (object &rest args)

(when object (apply #'draw (copy-object object) args)))



(defmethod mouse-click ((window selection-mixin) time pitch (modifiers (eql nil))
(count (eql 2)))

(select-cover window time pitch)

(open-selection window))



(add-music-command "Open Selection" #'open-selection #\O)
```

[to do: add #’selection als enable-p function: adapt `install- file- type- sub- menu`]

[to do: als de auto domain schaling gemaakt is moet openen van een deel zinnig zijn: uitgezoomed plaatje, nu foot note: how would you design this?]

draw rests

rusten verbieden in P en collectie (around setf en after init) en s mag niet alleen uit een rust bestaan dan makkelijk te tekenen

After creating a compound-musical-object all its elements will have a proper backpointer to it. As an example of the use of this construct consider the drawing of pauses. Uptill now these were simply ignored in drawing. But when a pause is part of a sequential structure, a white box with the height of the enclosing box would make a good graphical representation, and it would make the draw program more complete. footnote A rest in a parallel structure, or on the top level does not make much sense anyhow and could be ignored.

We only have to define a boundary method for pauses to achieve that. And because the calculation of the pitch range of the enclosing box for a rest needs accces to the enclosing object, the back-pointers come in really handy now. Remember that we can now at once define a draw-object method for any musical object, because boundary works for each of them.

```
(defmethod boundary ((object pause) onset window)

(when (element-of object)

(let ((boundary (boundary (element-of object))))

(when boundary

(destructuring-bind (left top right bottom) boundary

(list onset to (offset object onset) bottom)))))
```

```
(defmethod draw-boundary ((object pause) boundary window)

(draw-rectangle boundar window 1))
```

hoe oinderscheid maken met noten ??

enlarge boundaries eruit halen

Score system
Sound

steeds aan de lezer vragen dan aan eind helemaal bouwen defclass sound duration file pitc sound? same? loudness name after int probe-file transpose/mirror stretch play csound cut/copy/paste

Mop protocol

abstract class, at make-instance time, no need for mop

```
(defclass abstract-class (standard-class)())


(defmethod make-instance :around ((object abstract-class) &rest args)

(error "Cannot instantiate abstract class ~A" (class-name object)))


(defclass draw-window-mixin ()()

(:metaclass abstract-class))


(make-instance 'draw-window-mixin)

=>

> Error: Cannot instantiate abstract class draw-window-mixin

> While executing: #<standard-method make-instance :around (abstract-class)>
```

mix checking, at defclass time

```
(defclass inheritance-checker (meta-class)

(:before-when-present etc slots to be filled from initargs ??))
```

```
(defmethod check ((object meta-class) tree) t)

(defmethod check ((object inheritance-checker) tree)
(get-init-args object
:before-when-present
:should-be-before
:after-when-present
:should-be-after
:should-be-with
:may-not-be-with))

; reasons meegeven

(defmethod check-inheritance ((object inheritance-checker) list tree)
(loop for class in list
unless (check class tree)
do (error "Inheritance violation in class ~A, detected by ~A"
(name-of object) (name-of class))))

(defmethod initialize-instance :around ((object inheritance-checker) &rest args)
(let ((class (call-next-method)))
(check-inheritance class (inheritance-list class) (inheritance-tree class))
class))
```

should-have methods instatiation time but needs class, object may not be made

```
(defclass method-checker (meta-class)
(needed-methods))

(defmethod make-instance :before ((object method-checker) &rest args)
(unless (check-methods object)
(error "Cannot instantiate unfinished class ~A" (name-of object))))
```

```
(defmethod check ((object inheritance-checker) tree)

(check all needed methods present?))
```

vb, visible-structure needs calc-boundaries

auto mixin

```
(defclass auto-mixin (meta-class)

(conditions))
```

```
(defclass allow-auto-mixin (meta-class)

)
```

dan voor aanmaken allow-auto-mixins even alle auto-mixins proberen, als conditie dan toevoegen aan direct-superclasses, altijd vooraan

mop cocktail methods

als specializer and or not dan in compute applicable methods die ook terug geven of, als truc anonieme classen aanmaken die substitueren in method die classen automaties in mixen, in die specifieke gevallen around en zelf invocatie doen compute applicable methods doen

init keyword class adder

meta class instance init-mixer before make-instance van het object zelf, classes bijmixen (als uitbreiding van boven?)

one instance class

voor formats bv MTX en CSOUND gebruik in ??, eerst via table/resource noot allocatie, only 16 midi poort

als mop gebruik abstracte classes

Full code listing

problems in presentation order classes/objects load restrictions too much freedom as ADT class + methods, but not multi multimethods problem no real solution but browsers, progr environment conceptual order, logical order of domain good exercise present final version plus comment style/headers etc so much redefinitions, not much code

It is difficult to find a good order to present object oriented code, and it is often argued that this is no longer needed, given the existence of code browsers and other tools in the programming environment, and although these tools are indispensable, it is worth while to find a nice order, and add some comments at the level of groups of related programs etc. Other programmers that have to read your code, and even yourself (some time after you wrote it), will be thankful for a good listing. That is why we present in the appendix all code developed in this chapter. It is re-ordered and only the last versions of functions are kept. The listing contains some details that were omitted in the text (on handling pitch names and low-level byte manipulations for writing MIDI files). In the real world the code would have been split in a couple of files - but we present them here as one. Separation lines and comments at different levels indicate the structural units within the program.

Conclusion

[to do: beter uitwerken recap en gebruik vertellen]What have we got after all this writing and rewriting of code? We have designed a data representation for musical objects that proved quite flexible. We wrote tools for printing, drawing, playing and outputting them in different formats. We have defined a couple of transformations on them. The modularity is of a very fine grain - most programs are a couple of lines long, all take less then 10 lines of code. Though the code achieves quite some complex behavior, e.g. the nested drawing of boxes around compound musical objects or the writing of standard MIDI files, it still is easy to oversee and maintain because of that modularity. However, it has to be stressed that most programs are still bare skeletons of what a complete computer music system needs. But they are very good skeletons, and easy to fill-in. The constructions used and explained are the basic, object oriented constructs of CLOS. However, even these constructs (take e.g. the method combination for around methods) surpass with ease the power found in other object oriented languages. The design of programming language environments for the time based arts is so complicated in itself that programming language constructs that help in managing the complexity become a crucial issue NIL

. We hope to have shown how the constructs of CLOS can help achieving this.

Definitions made

`collection (compound-musical-object)` *`class`*

A group of musical objects, starting at specified times

`nested-mixin (draw-window-mixin)` *`class`*

A mixin to draw the structure in a nested fashion

`named-mixin (draw-window-mixin)` *`class`*

A mixin to print name tags of objects

`midi-awareness-mixin (draw-window-mixin)` *`class`*

A mixin to show distinction between MIDI and free notes

`compound-musical-object (musical-object named-musical-object-mixin)` *`class`*

A named group of musical objects ordered in time

`named-musical-object-mixin nil` *`class`*

A named object

`midi-note-mixin nil` *class*

A note mixin which will guarantee an integer pitch number

`midi-note (midi-note-mixin note)` *class*

A note which will guarantee an integer pitch number

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

Apply the operation tho the object and its onset

`mapc-musical-object (object operation &rest args)` *method*

Apply the operation to this object

`mapc-musical-object-onset (object onset operation &rest args)` *method*

Apply the operation to this object, and to each sub-..-sub-object and their onsets

`get-factor nil` *function*

Ask user to type a factor and return it as positive number

`get-interval nil` *function*

Ask user to type an interval in semitones and return it as number

`get-pitch nil` *function*

Ask user to type an pitch name

`(setf pitch) (value object)` *method*

Call the primary method only with a pitch that is kept within [0,127]

`(setf pitch) (value object)` *method*

Call the primary method with a pitch number

`(setf pitch) (value object)` *method*

Call the primary method with a pitch that is kept within [0,127]

`window-can-do-operation (window operation &optional item)` *method*

Can paste when a musical-object is on the clipboard

`clear-selection (window)` *method*

Clear selection from window

`transpose (object interval)` *method*

Don’t transpose when this would move the pitch outside the MIDI range

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

Draw a graphical piano-roll representation of a musical object

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

Draw a graphical representation of the musical object itself

`margin (object window)` *method*

Leave space in a boundary box for the name

`named-margin (object window)` *method*

Leave space in a boundary box for the name

`combine-event-lists (object onset self parts)` *method*

Merge a list of sorted agenda's into one sorted agenda

`merge-events (&rest events)` *function*

Merge a list of sorted agenda's into one sorted agenda

`multf (args)` *macro*

Multiply a generalized variable

`stretch (object factor)` *method*

Multiply the duration of a musical object by a factor

`paste (window)` *method*

Paste musical-object from scrap in window

`mirror-selection (window)` *method*

Reflect the musical object selected in the window around a center

`clear (window)` *method*

Remove current selection from window

`replace-selection (window object)` *method*

Replace selection in window with musical-object

`all-selected? (window)` *method*

Return T if selection is the whole musical object, false otherwise

`no-selection? (window)` *method*

Return T if there is no selection, false otherwise

`object-note-list (object onset)` *method*

Return a basis event list consisting of one onet-part pair

`collection (&rest agenda)` *function*

Return a collection of musical objects starting at specified onsets

`show (object)` *method*

Return a list describing a compound musical object with its components `note-list (object onset)` *method*

Return a list of onset-event pairs of all parts of a musical object

`event-list (object onset object-event-list-method)` *method*

Return a list of onset-event pairs of all parts of a musical object

`c (name &rest agenda)` *function*

Return a named collection

`p (name &rest elements)` *function*

Return a named parallel musical object

`s (name &rest elements)` *function*

Return a named sequential musical object

`sorted-note-list (object onset)` *method*

Return a note list: (onset note) pairs, with onsets in ascending order

`note (&rest attributes)` *function*

Return a note object with specified attributes

`example-collection nil` *function*

Return a simple collection of 5 notes

`event-merge (list1 list2)` *function*

Return a sorted event list of all events of both arguments

`replace-object (orginal replacement)` *method*

Return object with replacement, adjusting backpointer accordingly

`collection? (object)` *method*

Return t if the object is an instantiation of class collection, nil otherwise

`duration (object)` *method*

Return the duration of a collection

`second-draw-window nil` *function*

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

`all-class-subclasses (class)` *function*

Return the tree of classes inheriting from this one

`class-subclasses (class)` *function*

Return the tree of classes inheriting from this one

`clip (min value max)` *function*

Return the value, clipped within [min,max]

`named-example nil` *`function`*

Returns a simple musical object with named structure

`retrograde (object)` *`method`*

Reverse all components of the object in time

`cut (window)` *`method`*

Set scrap to current selection and remove selection from window

`copy (window)` *`method`*

Set scrap to current selection in window

`(setf duration) (value object)` *`method`*

Set the duration of a compound musical object by stretching its components

`(setf duration) (value object)` *`method`*

Set the duration of a compound musical object by stretching its components

`(setf onsets) (onsets object &optional onset)` *`method`*

Set the onset times of a collection, relative to the onset of the object

`sort-event-list (event-list)` *`function`*

Sort a list of onset-object pairs in order of increasing onsets

`stretch-selection (window &optional factor)` *`method`*

Stretch the duration of the musical object selected in the window by a factor

`musical-element nil` *`class`*

T

`musical-object (musical-element)` *`class`*

The abstract root class of all musical objects

`nested-margin (object window)` *`method`*

The margins of a nested compound boundary box

`draw-window (midi-awareness-mixin named-mixin mouse-select-mixin nested-mixin structured-mixin visible-selection-mixin piano-roll-contents-draw-window)` *`class`*

The window class named pianoroll

`transpose-selection (window &optional interval)` *`method`*

Transpose the musical object selected in the window by an interval

`make-instance (object &rest args)` *`method`*

`abstract-class (standard-class)` *`class`* `draw-boundary (object boundary window)` *method*

`boundary (object onset window)` *method*

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

`open-object (object &rest args)` *method*

`open-selection (window)` *method*

`new-draw-window (window &rest args)` *method*

`initialize-instance (object &rest args &key scrap-type)` *method*

`canon-selected (window)` *method*

`repeat-selected (window &optional n)` *method*

`transform-selection (window transform &rest args)` *method*

`clear-object (object)` *method*

`(setf elements) (elements object)` *method*

`(setf elements) (elements object)` *method*

`add-margin (&rest margins)` *function*

`same? (object1 object2)` *method*

`make-type-predicate (name class)` *function*

`make-constructor (name class)` *function*

`our-class (standard-class)` *class*

`window-change-class (object &optional class)` *method*

`get-window-cocktail nil` *function*

`select-direct-subclasses (class)` *function*

`make-cocktail-instance (ingredients &rest args)` *function*

`define-cocktail-class (ingredients)` *function*

`unmidi-selection (window)` *method*

`midi-selection (window)` *method*

`change-note-class (object class)` *method*

`intonation-example nil` *function*

`update-instance-for-different-class (old new &rest initargs)` *method*

`(setf pitch) (value object)` *method*

`midi-note? (object)` *function*

`midi-note (&rest args)` *function*

`named (name)` *function*

`named-constructor (object)` *method*

`enlarge-rectangle (box margins)` *function* `onsets (object &optional onset)` *method*

`clipboard (scrap-handler)` *class*

scrap-handler of for musical-objects

`musical-object-clipboard (clipboard)` *class*

scrap-handler of type :musical-object

Literature references made

(Pope, 1992)

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.

“Dannenberg, Desain and Honing (in Press)”

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

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 combination

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]

pretty printing

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

setf methods

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]