CENTER FOR COMPUTER RESEARCH IN MUSIC AND ACOUSTICS
OCTOBER 1993

Department of Music
Report No. STAN-M-85

SLAPPABILITY:

A NEW METAPHOR FOR HUMAN COMPUTER INTERACTION

Daniel V. Oppenheim

CCRMA
DEPARTMENT OF MUSIC
Stanford University
Stanford, California 94305

Slappability:

A New Metaphor for Human Computer Interaction

Daniel V. Oppenheim

Center for Computer Research in
Music and Acoustics (CCRMA)

Stanford University 1993

Slappability:

A New Metaphor for Human Computer Interaction

Daniel V. Oppenheim

Computer Music Center
IBM T. J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

Music@watson.ibm.com

Foreword[1]

Under current technology, as applications evolve into more complex systems they also become harder to use. Multiple views and representations, direct graphic manipulation, configure graphic interfaces, and extensibility via user programming are techniques widely recognized as improving the human computer interaction (HCI). However, there is no uniform methodology for using multiple views or, more important, for being able to use different underlying representations and move between them within a single application ([Honing 92], for a general discussion of music representations see [Dannenberg, 93] and [Wiggins, 93]). Moreover, user-environments often take a unidimensional approach that enables either direct-graphic-manipulation or user programming—the former is typically chosen for non-technical users and the latter for 'computer experts'.

The thesis put forward in this paper is that a blending and unification of the above techniques within a single user-environment is highly desirable. This should enable the design of more comprehensive and complex software systems that are yet more natural to use, quick to learn, easy to extend, and accessible to a more diverse community of users.

Slappability is an object-oriented mechanism that helps bring out many of these capabilities. I developed it at the Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, between 1992 and 1993. This paper explains Slappability, provides specific examples from its implementation within the DMIX environment, and discuss its potential applicability to human computer interaction. The reader should bare in mind that the ideal place to implement Slappability would be within the framework of an operating system rather than within specific application.

1. Motivation: Harnessing the Complexity

My recent work at CCRMA has focused on the design and implementation of software to support composing and performing music, namely, the DMIX environment [Oppenheim, 1993], written in ParcPlace Smalltalk-80. A major concern has been to support the user's creativity, or at least to minimize the system's interference with his/her natural flow of ideas. As a result, DMIX has evolved into a large and complex environment. It includes many different components which are not grouped together in any other system: a programmable algorithmic input language (Quill [Oppenheim, 90]), MAX-like real-time interactive modules, high-level graphic and real-time editors, numerous high-level tools such as Modifiers, Extractors, and Enumerators, score tracking capabilities, and more. The rationale for this diversity in tools, and hence overall complexity, is that each such tool has a different appeal to different people and that its usefulness changes as musical conditions evolve. The overall goal is to enable composers to move easily between such diverse tools and representations, thus aiding them in expressing their creativity.

Unfortunately, the ever evolving environment eventually became overwhelmingly complex. In an effort to make it more natural to use, yet without sacrificing functionality, system complexity, or extendibility, I finally came up with the idea of *Slappability*. Slappability has been fully incorporated into the DMIX environment. My evaluation of Slappability is based on my observations of composers using the system and on my experiences in teaching composition using the DMIX system at CCRMA, Laboratorio de Investgación y Procucción Musical (LIPM) in

[1] This work was carried out at the Center for Computer Research in Music and Technology (CCRMA), Stanford University.

Argentina, Bar Ilan University in Israel, and the National Chaio Tung University in Taiwan. Users background was varied and included non musicians, seasoned instrumental composers with little prior experience in computer music, as well as experienced computer musicians. The ease with which they performed complex tasks that otherwise require expert programing skills, and the short time in which novices could harness the system towards their creative goals, illustrates the merits of Slappability.

2. *Slappability*

Drag and drop allows the user to select a source object by clicking. A target is selected by dragging the source and releasing the mouse button. This mechanism merely initiates a transfer of data between the source and target objects. Typically, the source pastes data into a global clipboard so that the target may retrieve it. This mechanism is a convenient shortcut that, in its very essence, is limited to simple operations involving data transfer, such as Copy and Paste.

Slappability takes this mechanism a crucial step further. Once the source and target have been selected, a direct communication between the two objects is initiated. The target object receives a single message with the source object as an argument:

```
target getSlappedWith: source
```

It is now entirely up to the source and target objects to negotiate a response. Different objects respond in different ways and since the two objects themselves are now interacting directly, a world of endlessly rich possibilities becomes available. In DMIX, for example, Slapping text onto Graphics enables non-technical users to extend graphic manipulation in ways that would otherwise require skilled programming (see the example in section 4.2); carefully worked out compositions can be Slapped onto interactive Max-like objects that allow the user to 'perform' them and add expressive nuances, and much more (extensive examples follow).

The method `getSlappedWith:` is implemented in Class Object, ensuring some default action that will always take place. Every subclass may override this method in order to implement more appropriate behaviors. There are several additional bells and whistles, such as using menus when several Slap-behaviors make sense, passing the screen points at which the user clicked as an argument, or using a double-dispatch message passing technique—but these additions do not change the basic concept.

Three basic responses might take place: the source or target objects might be modified, they could both participate in modifying a third object, or a new object might be created. There are often several plausible responses, in which case a pop-up menu would offer different alternatives.

Slappability seems to have a magical effect on users: they quickly take advantage of the system's complexity and frequently jump between different tools and representations, or find ways of transforming their musical materials into tools, and vice versa. In fact, it apparently takes users less time to become comfortable in using DMIX than is the case with other high-end environments that implement only a single facet (i.e. either programming, or graphic editors, or real-time interaction, etc.).

3. Examples in DMIX

The first two examples will demonstrate the basic concepts of Slappability in some detail. The examples that follow illustrate less obvious possibilities.

3.1. Modifying music objects via Slappability

Figure 1-a is a DMIX piano roll view of the Bach Preluded no. 1 in C major. Time is represented on the X axis and pitch on the Y axis. The length of a note is proportionate to the length of its rectangle.

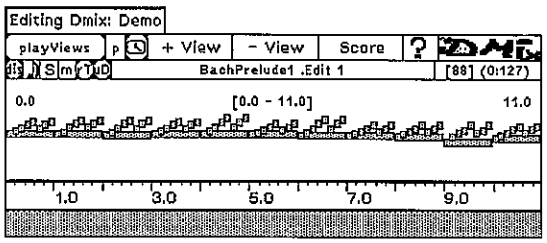

Figure 1-a: a graphic view of the Bach Prelude no. 1

Figure 1-b is a standard DMIX sine Function. Note the default window settings of the Function (second line in the label above the view): the X axis is interpreted as time and is set from 0 to 1

second, the Y axis is set to accommodate MIDI values of 0 to 127.

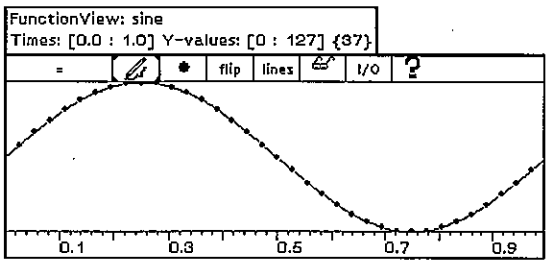

Figure 1-b: standard DMIX sine Function

The Slapping mechanism is initiated by keying COMMAND-S within the Function's view. Here the Function is dragged and Slapped onto the Bach Prelude.

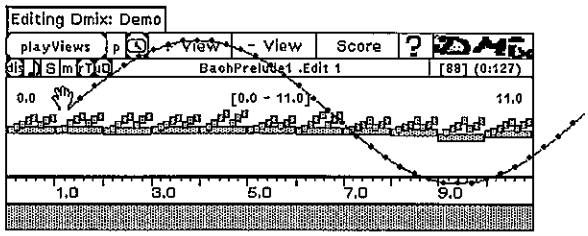

Figure 1-c: Slapping the sine Function onto the Bach Prelude

Unlike a drag and drop, a dialog now takes place between the Function and the music-object representing the Bach Prelude. The Function finds out the duration of the Prelude and adjust its X axis accordingly (note the new window setting in Figure 2-b). Then, the music-object requests the Function to apply itself on its pitch-parameter (a default). The result of this process is seen in the following Figure:

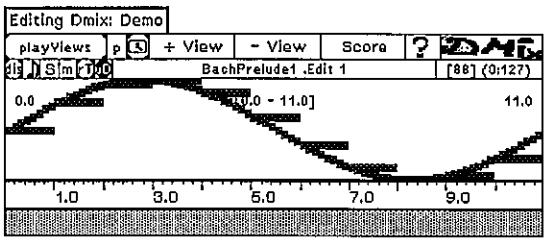

Figure 1-d: The result (by default pitch is affected)

3.2. Additional Slap options: modifying tempo

Note that there are many different ways in which a function might affect the Prelude. For example, it might be used to set any other parameter, such as velocity, duration, or MIDI channel number. It might also be used to scale a parameter, to modify time, or to modify tempo. A completely different intent might be to first transform the function into a music object and then merge or insert it into the Prelude. In the current implementation, a SHIFT-Slap pops-up a hierarchical menu that offers the non default alternatives:

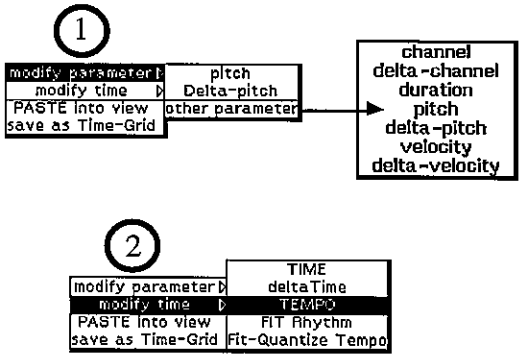

Figure 2-a: other Slap options become available by SHIFT-Slapping

The following example demonstrates how the same Function can now be used to further modify the tempo of the already modified Prelude. Modifying tempo requires a scaling of the delta-times between notes. To facilitate this, the Function's mode is set from '=' to '/' (division). The Function's Y axis is also changed and set to the range of 0.5 to 2 so changes in tempo are set between half and double that of the original. Note that the Function's X axis is already set between 0 and 11 seconds—this was changed by the music object during the dialog that took place in the previous Slap.

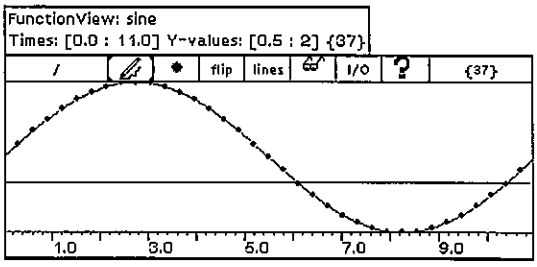

Figure 2-b: setting the function to enable tempo modification

The Function is SHIFT-Slapped onto the graphic view, and the option: 'modify time: TEMPO' is

selected from the hierarchical menu shown in Figure 2-a. The result is seen in figure 2-c:

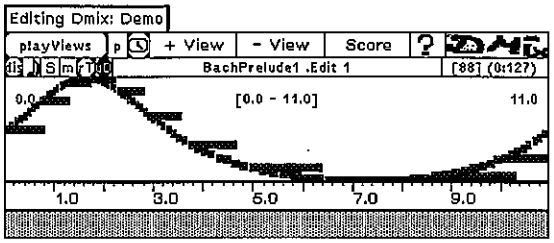

Figure 2-c: result of modifying tempo

3.3. *Transmogrification*

In the previous examples a Function was Slapped onto the Bach Prelude and modified the music. Slapping a music object onto a Function has a completely different effect. Now, the music object is modifying the Function. The Function's data (collection of x-y points) are replaced with new points that are collected from the Slapping music object. In the following example a 4.6 second phrase from of the Bach Prelude (a) Slapped onto an arbitrary Function (b). As a result, each x-y point in the Function's data now corresponds to a begin-time / pitch of a note in the corresponding phrase (c). The pitches are represented in the Y axis, and the onset times, or rhythms, on the X axis:

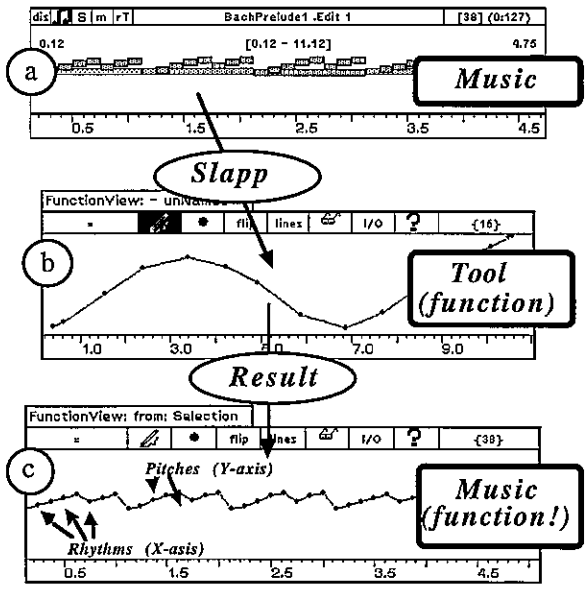

Figure 3: Slapping a music object onto a Function

Two subtle, yet extremely important, points should be noted. Firstly, the Function is now a representation of the music, though some information was lost (i.e. velocity and MIDI channel). In fact, the technique demonstrated above could be used to represent the music as a collection of functions, one for each parameter (pitch, velocity, etc.). In other words, the user is thus able to actually modify the underlying music representation he/she is using. Secondly, whereas a user would naturally think of the piano roll notation as a representation of music, and of the Function as representing an abstract, high-level, mathematical tool, Slappability helps blur these conceptual differences. Now the user may think of the Function in musical terms, such as a phrase from the Prelude, or as a collection of the rhythms, or as gestural or dynamic information.

This property of *Slappability* that enables the transformation of one object, or representation, into another, seems particularly useful and I call *Transmogrification*. In the following example a Jazzy improvisation is Slapped onto a Function (a), the resulting rhythms are extracted and become the Function's data (b). The new Function, or rather the Jazzy rhythm, is then Slapped onto Bach's Prelude in C major (c) with the result being a rhythmically Jazzed-up Prelude (d)—the onset time of each note in the Prelude is shifted to the closest onset time in the Jazzy improvisation, often forming interesting chords.

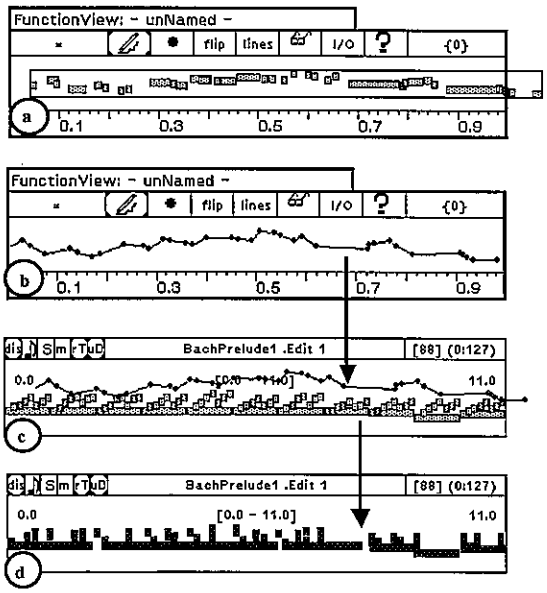

Figure 4: Transformations between music and tools

3.4. Creating new objects

Slapping can also produce new objects. *Slapping* one Function onto another can create a new Function that is the resultant of some mathematical operation between them. In Figure 5 a sine-wave Function is Slapped onto an up-ramp Function (a). In (b) the Functions are multiplied by each other and in (c) they are added. Tools can thus easily be personalized either by *transmogrifying* some other object into a tool, or by creating new tools based on existing tools.

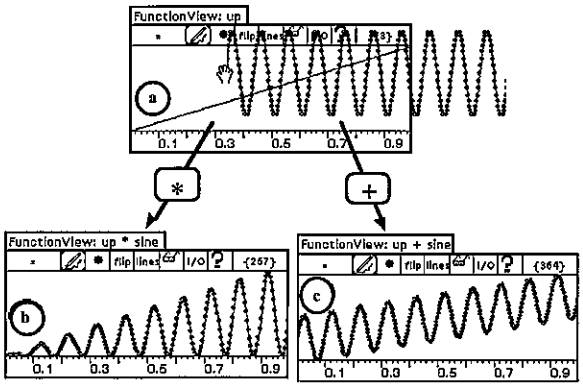

Figure 5: Slapping one Function onto another

3.5. Graphics and Algorithms

In the following example (Figure 6), music was selected in a graphic view and then Slapped onto an empty edit window of a QUILL algorithmic music generator [Oppenheim 90]. The music-event, that is 13 notes long, is transformed into an ASCII algorithmic description using the QUILL syntax. Compiling this text will regenerate the same music. However, the algorithm can now be changed to produce a musical result that cannot be obtained via graphic manipulation.

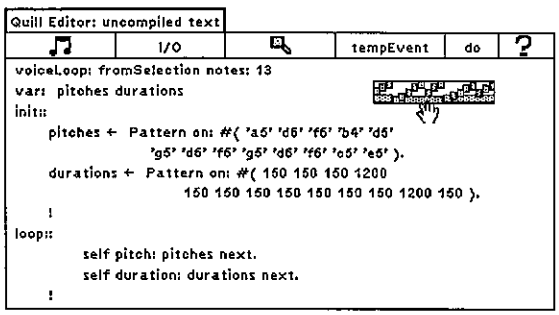

Figure 6: Converting music-events into algorithms

The voiceLoop syntax above is closely modeled after the VOICE construct in PLA [Schosttstaedt 1983] and the PART construct in Common Music [Taube, 1991]. In the above example a loop creates a voice named `'fromSelection'`. The loop will iterate 13 times (`notes: 13`), once for each note in the original phrase. Two variables are declared: `pitches` and `durations`. In an initialize block (`init::`) each variable is assigned a Pattern—a list accessing object. The list provided to each Pattern is the collection of a parameter field from each of the 13 notes. The loop (`loop::`) simply assigns the next parameter in the list each time it iterates.

The voiceLoop syntax used above is almost pure Smalltalk, and a composer that is also an experienced Smalltalk programmer could add conditionals, use Functions to modify parameters, and many other objects made available by Smalltalk and DMIX. However, a novice user will need a considerable investment in time in order to acquire the programming skills needed to work on this level. In order to make Quill accessible to non programmers, a much simpler input format was implemented. The first five notes could be entered in this manner (comments are surround by double quotes):

```
                    "list of pitches"
p:   a5   d6   f6   b4   d5

                    "list of durations"
d:   150      150      150      1200      150
```

An interesting discovery I made was that Slappability enabled me to significantly extend the power and flexibility of this simple syntax. The key to this improvement was the ability to Slap any object into Quill, where it would become available to the user. For example, the Bach Prelude in Figure 4-c could be Slapped into Quill as a motive. Similarly, the Function used to 'Jazz up' its rhythm in Figure 4-b could be Slapped and saved in Quill. Assuming this has been done, the entire syntax needed to obtain the exact same result that was obtained graphically in Figure 4 would be the following:

```
                "mark begin time (t:)"
t: [beginFunction: 'Jazzy Rhythm']
insertMotive: 'Bach Prelude'

                "mark end time"
t: [endFunction: 'Jazzy Rhythm']
```

3.6. Composition and Performance

Figure 7 shows a graphical view of already composed music being Slapped onto an ECHO—a MAX-like real-time algorithmic music processor. The ECHO accepts the music as if it were being input by a performer in real-time. The composer can now interact with the music and add expressive nuances as it is being processed.

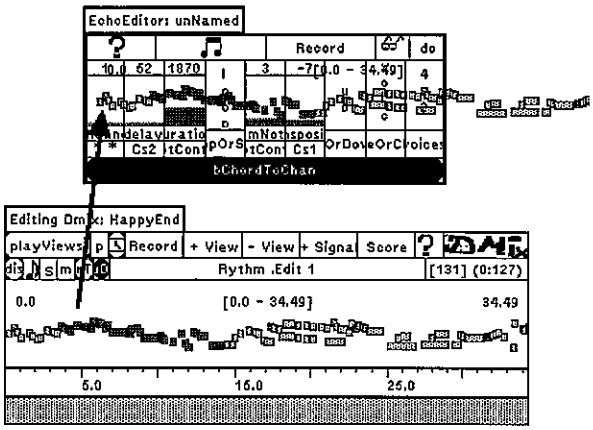

Figure 7: Performing Compositions

4. Complexity versus Usability

As programs become more complex, they also become harder to use. Examining this problem from several perspectives will demonstrate how *Slappability* should prove useful.

4.1. Menus, Buttons, and Libraries

In the evolution of commercial applications, it seems that as newer versions become more sophisticated they also become more complicated to use. For example, Microsoft Word used only about 20 menu-items in the 1984 release, but by 1992 there were over 50 [Microsoft 1992]. Similarly, the first version of Aldus SuperPaint required about 350K of disk space. Two versions later it occupied about 3.2 Megabytes, and the number of menu-items and tools increased proportionally. Unfortunately, so did the difficulties in using it (based on my experiences and those of my colleagues). The direct relation between complexity and difficulty of use may be an inevitable result of what I term a *Linear Design Paradigm*. The issue of interface is taken quite literally: for each operation that the program can support there is one menu item (or button) for the user to click on. As applications get more sophisticated their user-interface invertibly becomes more cluttered and therefore harder to use.

My premise is that no matter how many menu-options are available, a *creative* user will always want to do something NEW for which there is no menu button. Simply increasing the number of menu-items in new releases to accommodate for users 'wish lists' is counterproductive since the added complexity in the user interface hampers the efficient use of the application.

With *Slappability* I have discovered two things. Firstly, for many operations menus can be bypassed altogether, making the system much more manageable and natural to use. Secondly, I discovered unexpectedly that users who were working via *Slappability* rather than menus started doing things that I, as a designer, never thought of and for which no menu-item was available. In other words, not only was the user's creativity less impaired by the system's design, but also the system's design was enhanced by the user's actions, and that is no small matter!

4.2. Graphic manipulation Vs. Programming

There seems to be a dichotomy between two fundamental approaches to user-interface design: namely direct-graphic-manipulation and programming. Compare Microsoft Word, which is menu driven and simple to use, with EMACS or TEX, which are more flexible due to their programmable interface but require a large investment in time to take full advantage of their flexibility. Similarly, compare commercial MIDI sequencers (with graphical interfaces) to high-end, programmable, ASCII-based environments such as Common Music [Taube, 1990]. Regrettably, the application designer must compromise and choose between one approach or the other.

Through *Slappability*, however, I found ways that avoid such a compromise and bridge between the two. By typing text (source code) into any text-window and Slapping it onto a graphic editor DMIX users can achieve results that otherwise require expert programming skills. Moreover, by a careful factoring of the design I eliminated the need to know about the underlying data-structures or implementation. Thus even novices can enjoy many of the benefits of programming while working with a familiar, and reassuring, graphical interface. For example, Slapping the following

line of text onto a graphic view will cause the MIDI pitches (key numbers) to equal that of the same notes' velocity + a random value between 0 and 5:

```
event pitch:event velocity
          + (random next * 5)
```

The following text will access the currently selected notes in the graphic editor, extract their #pitch parameter, transform that into a Function , then open a graphic editor on that Function:

```
(activeSelection asEvent
        asFunction: #pitch) edit
```

4.3. Formalism and the *Creativity Flowchart*

Software design has traditionally consisted of two stages. First the domain, and then the task (the way a user works), are formalized and modeled in software. The domain is modeled as a set of data structures and primitives. The task is modeled and implicitly formalized in the overall application design and user-interface. In many domains this works well—NASA can fly astronauts to the moon in auto-pilot mode. But in domains where the user's creativity plays a vital role, such as music composition, or graphic design and animation, pre-determining how a user must work can have a profoundly negative effect on the end result. Below is an example and discussion of this potential problem, but due to the limited scope of this paper it must be kept somewhat simplistic.

Let us go back in time and consider a computer system that might have aided Classical musicians to compose. The user might have gone through three stages:

stage 1. define a key;
stage 2. write (an 8 bar) melody;
stage 3. harmonize.

This is obviously oversimplified, but plausible: many Classical works could have been composed using such a system since most works of this period conform, more or less, to this sequence. But now comes an intriguing question: could Beethoven's Fifth Symphony have been conceived and/or implemented with such a system? I claim unequivocally that the answer is no: in the Symphony's first introductory measures the key is ambiguous, no melody is presented, and the motive is not harmonized—even though things become clear soon after. If Beethoven would have limited his creative thinking to the above three-stage process this profound creation would never have come to be. Whereas this hypothetical system would indeed support creativity, it would be restricted to a superficial level that shares a broad common denominator with many compositions. However, it is precisely the individuality of each composer, his or her unique way of thinking, of putting ideas together, personal techniques for working out materials, and so forth, that give birth to new creations (this is not to claim that every work of art must be innovative). I suspect that no system having a formalized preconception of how a human should use it will ever fully support *creativity* in the sense that it will encourage a 'Beethoven' to create a work as unique as the Fifth Symphony or the late Quartets; rather, such a feat would probably be accomplished in spite of the system's limitations. Current formal approaches seem oriented less towards encouraging the discovery of the new and the unique, and more towards a rehashing of the mundane and mediocre. This problem stems from the implicit need to formalize the task, and I refer to to this problem as a *creativity flowchart* syndrome: computers are formal systems; but humans that use computers are not, and should not be expected to behave as such. Object-Oriented technology offers interesting alternatives.

4.3.1. Non-formal Alternatives

Object-oriented methodologies provide a powerful paradigm that might help avoid the rigidity implied by the *creativity flowchart* syndrome and better support the unexpected ways people might decide to use systems creatively. In such applications, objects seemingly 'float' in the system with no preconception of what other objects they will interact with or when. Thus, a careful design could leave all notions of 'where to begin' and 'what to do next' to the user's discretion (for a more detailed discussion see [Oppenheim, 91]).

Regrettably, such a design still requires an equally flexible user-interface to take full advantage of its potential, and to enable unpredictable interaction with the system. Direct-graphic-manipulation using *Slappability* implements this interface in a unique and powerful way. It not only allows the user to jump quickly between presentations, but also extends the idea of multiple presentations by enabling one object-type to transform itself into another. For example, in DMIX music can be transformed into Functions (tools) as seen in Figure 4, but also into Quill algorithms (Figure 6), Filters, Patterns, and more. *Slappability* also

offers a natural way to jump between different modes of creation: improvisation, programming and algorithms, scoring, and graphics, among others. Moreover, the user can continuously switch between a top-down, bottom-up, side-in, or side-out approach. In short, the system can provide an extremely rich, unconstrained compositional environment. The user is able to express himself or herself uniquely, and in ways that seem more natural. Moreover, by and large the structure of the underlying applications gets configured by the user as he or she works, rather than be determined by the programmer.

4.4. Tools, individuality and transmogrification

Tools used by composers tend to be highly personalized. Every artist uses different tools which play an important role in developing a unique style that expresses his or her individuality. Software tools enable users to perform high-level operations of a more general nature than those obtainable through buttons or menu-items. There is a trade-off between high-level, ease of use, generality, and individuality. High-level tools are easy to use but perform a more generic action; low-level tools require configuration and hence can be made more personal, but are harder to use. Paradoxically, the tools intended to 'free' one's creativity often end up limiting it via the narrowness of the channel of common choices. For example, if books were written by hand I expect we would find much more variety in layouts or ways of attracting attention to important points. High-level formatting tools tend to produce documents that appear similar to each other (this is not necessarily a drawback) but they also tend to discourage individual expression. *Slappability* enables the creation of tools from musical materials, or from combinations of other tools, and thus encourages the building of unique tools.

5. Conclusions

The strength of *Slappability* lies in the ways in which it allows users to control complex systems with ease. With the availability of Slappability, DMIX users now welcome the complexity of the system, whereas before they were often overwhelmed by it. The fact that non technical users can achieve results that otherwise require expert users suggests that it might be ideal in fields such as music education, where programs could be devised for children to use without sacrificing the more complex musical issues.

Like copy-and-paste or drag-and-drop, Slappability is a but a mechanism. It becomes especially useful as a metaphor for users if the underlying application is flexible, modular, extensible, and supports object-transformations. It should ideally be located in an operating system, where it could enable the sharing of applications in novel ways. For example, editing a sound file in a music program, defining a digital filter in an unrelated application such as Mathematica, and then Slapping the sound file onto Mathematica to have it processed (see Figure 8). I believe this would have a beneficial impact both in the way applications would be designed as well as on the overall user environment, user interface, and general human computer interaction.

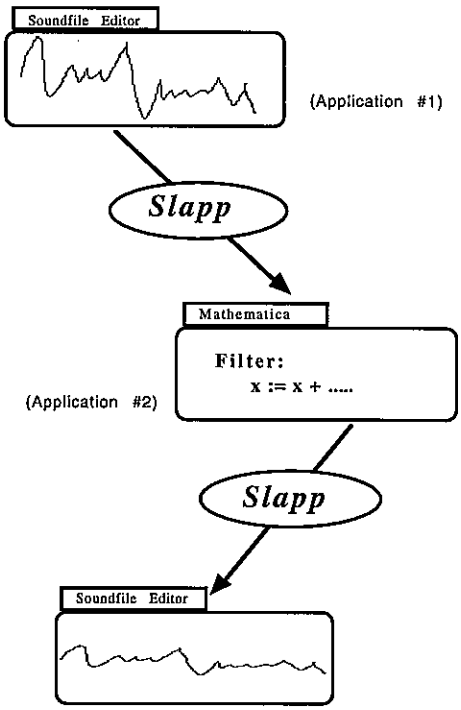

Figure 8: using Slappability between applications

Creativity is a fundamental component of human thinking and behavior, and this is by no means limited in its domain to the arts. Hence the problem of supporting creativity is general and relevant to all domains of human computer interaction. Slappability is one way to improve the support of users in their creative work.

Acknowledgements

This work could not have come to be without the continuous support and encouragement from everyone at CCRMA. It was Bill Schottstaedt that pointed out that an easy way to move between the numerous representations in DMIX would be useful—an innocent remark without which Slappability would not have come to be. I am especially grateful to the numerous composers and students who used DMIX and gave precious feedback and much encouragement, including Joanne Carey, Eitan Avitsur, Kui Dong, Jan Vandenheede, Celso Aguiar, Peter Bramble, and many many others.

References

[Dannenberg, 1993] "Music Representation Issues, Techniques, and Systems." Computer Music Journal Vol. 17(3):20-30. MIT Press.

[Honing, 92] "Issues in the Representation of Time and Structure in Music." in Desain P. and Honning H. "Music, Mind and Machine: Studies in Computer Music, Music Cognition and Artificial Intelligence." Amsterdam : Thesis Publishers. Also in Proceedings of the ICMC 1990, Glasgow, Scotland.

[Oppenheim, 1990] "QUILL: An Interpreter for Creating Music-Objects Within the DMIX Environment", Proceedings of the ICMC, Montreal, Canada.

[Oppenheim, 1991] "Towards a Better Software Design for Supporting Creative Musical Activity (CMA)", Proceedings of the ICMC, Montreal, Canada.

[Oppenheim, 1993] "DMIX—A Multi Faceted Environment for Composing and Performing Computer Music: its Design, Philosophy, and Implementation", Proceedings of the SEAMUS Conference, Austin, Texas; also in proceedings of the Arts and Technology Symposium, Connecticut College, Connecticut.

[Schosttstaedt 1983] "PLA: A Composer's Idea of a Language." Computer Music Journal 13(3) :49-55.

[Taube, 1991]. "COMMON MUSIC: A Music Composition language in Common Lisp and CLOS." Computer Music Journal 15(2):21-32.

[Wiggins, Miranda, Smail, and Harris, 1993] "A Framework for the Evaluation of Music Representation Systems." Computer Music Journal 17(3):31-42.

[Microsoft 1992] Microsoft Word manual, Microsoft Corporation.