Slappability: A New Metaphor for Human Computer Interaction

Daniel V. Oppenheim

Center for Computer Research in
Music and Acoustics (CCRMA)
Department of Music
Stanford University, Stanford, CA 94305
Dan@CCRMA.Stanford.Edu

Under current technology, as applications evolve into more complex systems they also become harder to use; user-interfaces are often unidimensional: either direct-graphic-manipulation or programmable. *Slappability* is a new object-oriented approach to software and user-interface design, developed by the author specifically to aid in the design of large, sophisticated systems that are more natural to use, quick to learn, and easy to extend. It integrates diverse approaches such as graphics and programming, or composition and performance, and provide a simple and consistent user interface. *Slappability* has been incorporated into the DMIX environment for composing and performing music. The ease in which users perform complex tasks that otherwise require expert programing skills, and the short time in which novices harness the system towards creative goals, strongly indicates the merits of *Slappability*.

In this presentation I will define *Slappability*, provide specific examples from its implementation in the DMIX environment, and discuss its potential applicability to human computer interaction. Implementation details are provided at the end of the paper.

1. Motivation: Harnessing the Complexity

For the past several years, my work has focused on the design and implementation of software to support the composition and performance of music, namely, the DMIX environment [Oppenheim, 1993]. A major concern of mine has been to support the user's creativity, or at least to minimize the system's interference with his 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 existing application: 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, score tracking, 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*, and have been experimenting with it during the past year.

2. *Slappability*

To the user *Slappability* seems no different than the familiar drag-and-drop metaphor: he clicks and selects a source object, drags it above a target object, and releases the mouse. Indeed, *Slappability* may be thought of as an extension of drag-and-drop, the main difference being the endlessly rich possibilities that can result. 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 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.

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 it does with other high-end environments that implement only a single facet (i.e. either programming, or graphic editors, or real-time interaction, etc.).

2.1. Slappability versus Drag-And-Drop

Whereas the common drag-and-drop mechanism initiates only a transfer of data between the source and target objects (typically via a global clipboard), *Slappability* permits more sophisticated inter-object communications. For example, one object could in some way transform the other (e.g., in Figure 1 where a Function is Slapped onto a graphic view, or in Figure 3, when Slapping music into a Quill window), the two objects could generate new objects and remain unchanged (e.g., Slapping one Function onto another, Figure 2), or they might establish a link to enable potential future communication (e.g., such as connecting an AD port to a Filter).

3. Examples in DMIX

3.1. Transmogrification

Figure 1: Transformations between music and tools

One property of *Slappability* that seems particularly useful is the ability to transform one object into another, a property I will term *Transmogrification*. In DMIX music can be transformed from its graphical representation into Modifiers—high level tools such as a Function—and vice versa. In Figure 1, for example, a Jazzy improvisation was Slapped onto a Function (a), as a result the rhythms were extracted and became the Function's data (b). The new Function, or rather the Jazzy rhythm, was then Slapped onto Bach's Prelude in C major (c) with the result being a rhythmically Jazzed-up Prelude (d).

3.2. Creating new objects

Slapping can also produce new objects. *Slapping* one Function onto another can create a new

Function that is the result of some mathematical operation between them. In Figure 2 a sine-wave Function is Slapped onto an up-ramp Function (a). In (b) the Functions are multiplied to 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 2: Slapping one Function onto another

3.3. Graphics and Algorithms

In the following example (Figure 3), music was selected in a graphic view and then Slapped onto an empty edit window of a QUILL algorithmic music generator. The music-event gets transformed into an ASCII algorithmic description using the QUILL syntax. Compiling this text will then regenerate the same music. However, the algorithm can now be edited and change the music in ways that cannot be performed graphically.

Figure 3: Converting music-events into algorithms

3.4. Composition and Performance

Figure 4 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 4: Performing Compositions

4. Complexity versus Usability

As programs become more complex, they also become harder to use. Examining this problem from several perspective will demonstrate how *Slappability* might 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. Similarly, the first version of Aldus Super-paint required about 350K of disk space. Two version 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 hardship 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 gets 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 'whish lists' is counterproductive since the added complexity in the user interface hampers the efficient usage of the application.

With *Slappability* I have discovered two things. First, for many operations menus can be bypassed altogether, making the system much more manageable and natural to use. In addition I unexpectedly discovered 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: direct-graphic-manipulation versus programming. Compare Microsoft Word, which is menu driven, 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 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 all the MIDI velocities to equal the MIDI key-number (pitch) + 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 and 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 takes 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 I provide an example and discussion of this potential problem, that due to the limited scope of this paper 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 go through three stages:

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

This is obviously oversimplified, yet it seems plausible that 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 (things do become clear soon after). If Beethoven would have limited his creative thinking to the above three-stage process this profound creation would not have come to be. Whereas this hypothetical system would indeed support creativity, it would do so on a superficial level that shares a broad common denominator with many compositions. However, it is precisely the individuality of each composer, the 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 with 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 it 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' up to the user (for a more detailed discussion see [Oppenheim, 91]).

Regrettably, such a design still requires an equally flexible user-interface to take full advantage of this 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 quickly jump 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 1, but also into Quill algorithms (figure 3), Filters, Patterns, and more. *Slappability* also offers a natural way to jump between different modes of creation: improvisation, programming and algorithms, scoring, 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, 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 being determined by the programmer.

4.4. Tools, Individuality and *transmogrification*

Tools used by composers tend to be highly personalized. Every artist uses different tools that play an important role in developing a unique style that expresses his 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 a narrow 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 might also 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. The Implementation

The underlying mechanism that implements *Slappability* is simple: The user selects a source object by clicking, and by releasing he selects a target object. Next is the most important phase: the target receives a single message with the source 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. For example, if a music-object is Slapped onto a graphic view it might get pasted in, but if the same music-object is Slapped onto a Function, some aspect may be extracted (pitch, duration, delta-time, etc.) and become that Function's data (see Figure 1). Similarly, if a Function is Slapped on to a music-object it will modify some aspect (e.g., pitch, time, etc., in Figure 1); but if one Function is Slapped onto another, then some mathematical operation will take place between them (*, /, +, -) creating a new Function (see figure 2).

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 where the user clicked as an argument, or using a double-dispatch message passing technique—but these additions do not change the basic concept.

6. Conclusions

The strength of *Slappability* lies in the ways it allows users to control complex systems with ease. Paradoxically, DMIX users now welcome the complexity of the system, whereas before they were often overwhelmed by it.

Like copy-and-paste or drag-and-drop, *Slappability* is 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.

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 try to better support users in their creative work.

Acknowledgements

I would like to thank Bill Schottstaedt for pointing out that an easy way to move between the different representations in DMIX would be very useful. I would also like to thank the many composers and students at CCRMA who gave important feedback and much encouragement.

References

- [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.
- [Taube, 1990]. "COMMON MUSIC - A Music Composition language in Common Lisp and CLOS". CCRMA report STAN-M-63, Stanford University.