# ON CONCEPTUAL MISFITS IN COMPUTER MUSIC PROGRAMMING

*Hiroki NISHINO*
NUS Graduate School for Integrative Sciences & Engineering
National University of Singapore

**ABSTRACT**

How usability problems in a domain-specific language(DSL) can be assessed is one of the largely unexplored issues in user-centered design. It has been frequently discussed that abstraction is an essential skill in software design yet inappropriate abstraction can cause significant usability problems. Similarly, the misfits between the users' conceptualization and the representations implemented within a system are also considered to cause usability problems. We apply such perspectives to this issue of usability problems in DSL design, with an additional consideration on the abstraction layers. We describe an example of conceptual misfits in computer music programming, partly based on our previous publication. Such an assessment can be beneficial for user-centered design of DSLs.

## 1. INTRODUCTION

While the interests in research of computer music programming languages have been mostly from researchers and practitioners of computer music, it is also recently drawing attention as a research topic in the context of human-computer interaction(HCI) [1, 3]. End-user programming activity is becoming more popular than ever in many professional domains.

Some end-user programmers can be categorized as *expert end-users*, who are expert in their own problem domain but still novice in computing [3]. Blackwell and his colleagues discuss that such expert end-users should be distinguished from both expert and novice programmers and the research on first year computer science students or the research on 'natural' programming languages by studying kids before learning any other language *"may not be directly relevant to needs of expert end-user programmers"* [3]. Computer music programming is ideal for such research in expert end-user programming in that computer musicians can be considered as novice programmers but with their own expertise in computer music. Moreover, the expertise knowledge that computer musicians possess is highly organized as an academic research field and is shared by computer musicians, when compared to more general end-user programming activity.

However, how usability problems in a DSL can be assessed is still largely unexplored yet is an issue of significant importance towards user-centered design. In this paper, we discuss this issue of usability problems in computer music programing language design. We combine several perspectives provided by the previous research with an additional consideration on the abstraction layers both in the software design and the users' conceptualization, to assess how usability problems can be attributed to conceptual misfits between expertise knowledge and language design.

The discussion includes the materials that overlap with our previous publication [12], yet some additional and detailed discussions are also described.

## 2. RELATED WORK

In this section, we briefly review some previous research related to our discussion in the later section. Blackwell and his colleagues discuss how inappropriate abstraction in software design can cause serious usability problems from many aspects in [4]. As Blackwell describes in the paper, *"even where developers are well motivated and sympathetic to user concerns, incompatible abstractions are a constant challenge to user centered design"*; thus, avoiding such 'incompatible abstractions' is one of major concerns in software design in general.

Such a view that problematic abstractions in software design can cause significant issues in a computer system is not limited to the context of end-user computing

and widely discussed as point of interest in software engineering in general. Lee's *Computing Needs Time* [11] provides a suggestive perspective to discuss DSL design, even though what he argues is not in the context of HCI, but mainly in one of Cyber-Physical Systems.

Lee attributes the loss of repeatability and predictability in today's computer systems to the abstraction layers of computing, since the passage of time is abstracted away in such abstraction layers. This view, that significant problems in software design can find root in the abstraction layers in system design that take away the controllability and the accessibility that are necessary for some tasks to be implemented in the higher level layers, is also applicable to DSL design, since a DSL involves higher level abstractions for its own application domain than general purpose programming languages.

Blandford and her colleagues are developing usability evaluation framework called CASSM (Concept-based Analysis of Surface and Structural Misfits, *"the purpose of which is in the identification of misfits between the way the user thinks and the representation implemented within the system"* [6]. As Blandford discusses in [7], *"the majority of HCI research and design has ignored this issue"* of this notion of misfits, yet such conceptual misfits that cause usability problems can be considered to provide good opportunity for user-centered design and redesign.

Blandford describes two types of conceptual misfits as following. *"Some misfits are surface-level - for example, users may work with concepts that are not directly represented within the system; conversely, users may be required to discover and utilize system concepts that are irrelevant to their conceptual models. Other misfits are structural, emerging only when the user manipulates the structures of some representation and finds that changes that are conceptually simple, in practice difficult to achieve"*. [8]

## 3. CONCEPTUAL MISFITS IN A DSL DESIGN

Programmers are considered *"to use knowledge from at least two domains, the application (or problem) domain and the computing domain, between which they establish a mapping"* [10, p.22]. The knowledge from the application domain is also utilized in comprehension of the existing programs [10, pp.75-103] and software maintenance can be conceptualized *"as interlinking comprehension and modification"* [14]; thus, facilitating mapping between these two domains is of significant importance for improving the usability in many different phases of programming activity. Hence, the conceptual misfits between the representations of the expertise knowledge (in problem domain) and the programming language design (in computing domain) is a significant factor to consider the usability of DSLs.

We combine the three perspectives described in the previous sections so as to discuss such conceptual misfits in DSL design. We consider the following situations as the causes of such conceptual misfits; a) The abstraction layers in DSL can make some entity in a lower-level less accessible. Yet, the counterpart of this entity in the problem domain is involved in problem solving activity. b) An entity in the users' conceptualization, which is involved in problem solving, does not have a counterpart in the DSL design. c) An entity in the DSL design does not have a counterpart in the users' conceptualization, but must be involved in problem solving.

All three situations can cause conceptual misfits and involve 'incompatible abstractions' in the abstraction layers applied to the DSL design. We describe a simple case study in the next section of such an approach.

## 4. A CASE STUDY IN COMPUTER MUSIC

We describe a simple case study in computer music, taking *"Single-Sample Feedback"* in SuperCollider [15] as the example of such conceptual misfits. Single-sample feedback is a very simple task as a concept but difficult to achieve in SuperCollider. This usability problem is lying in the representations implemented within SuperCollider. Thus, it can be considered a good example of a structural misfit in computer programming.

### 4.1. Musical Time-Scale and Synthesis Framework

Roads classify musical time-scales in computer music into 9 different scales in [13, p.3]. Sorting from the longer time-scale to shorter ones, Roads's musical time-scales are classified as *infinite*, *supra*, *macro*, *meso*, *sound object*, *micro*, *sample*, *subsample* and *infinitesimal*.

We briefly describe only three of them, since the other time-scales are not involved in the following discussion. In Roads's definition, sound-object time-scale corresponds to *"the traditional concept of note"*, microsound time-scale to *"sound particles on a time scale that extends down to the threshold of auditory perception (measured in*

*thousandths of a second or milliseconds)"*, such as grains in granular synthesis. Sample time-scale corresponds to each sample in digital signal. [13]

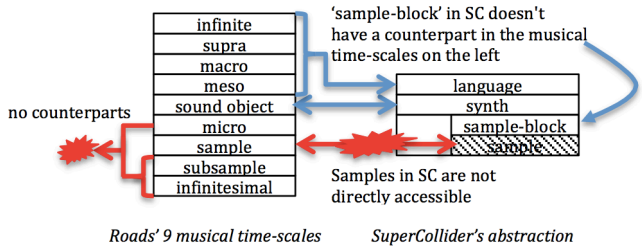


**Figure 1**. Roads's 9 musical time-scales [13, p.3] & the abstracion layers in SuperCollider

The abstraction applied to SuperCollider's synthesis framework are quite typical as a computer music programming language. A *synth* object in SuperCollider correspond to the traditional concept of note. SuperCollider perform DSP, not sample-by-sample, but by the blocks of samples as in many other computer music environments. Thus, the representations of musical time-scales in SuperCollider's synthesis framework can be considered to be conceptually organized as *synth* - *sample-block* - *sample*.

Figure 1 shows musical-times scale by Roads (on the left) as the users' conceptualization in the expertise knowledge and in SuperCollider (on the right) as the representations implemented within the system in its abstraction layers.

### 4.2. Single-Sample Feedback in SuperCollider

Single-sample Feedback is one of the elementary techniques involved in many different signal processing techiniques from the simplest IIR filtering to more advanced techniques such as physical modeling synthesis. Yet, many computer music programming languages involve difficulty in programming single-sample feedback; it is even impossible in some environments.

While many computer music programming languages provide built-in objects for synthesis techniques that involve single-sample feedback, such a solution can be sometimes even almost meaningless when the purpose of end-user programming activity is in *exploratory design* or *exploratory understanding* [5, pp.103-104]. For instance, if a user wants to experiment a new physical model for synthesis that involves single-sample feedback by programming, ready-made built-in objects hardly help such programming activity.

```
out(i) = (1 - abs(c)) * in(i) + c * out(i-1)
where    out(i)   is the current output sample
         in (i)   is the current input sample
         out(i-1) is the last output sample
         c        is the given filter coefficient.
```

**Figure 2**. One Pole Filter Algorithm

SuperCollider also has a usability problem in single-sample feedback. We take an example of one-pole filter, the algorithm of which is shown in Figure 2. While it is still possible to implement in SuperCollider, it involves unnecessary difficulty in programming. As the code shown in Figure 3, the code is a lot more complicated, compared to the simple algorithm in Figure 2.

How this usability problem can be assessed? In perspective of conceptual misfits, as shown Figure 1, each sample in SuperCollider is directly inaccessible while the conceptualization of the one pole filter algorithm involves the direct access in sample time-scale. This conceptual misfit in accessibility causes one of the usability problems in the code that the user must write the incoming signal once into a buffer to access each sample and then write it back to another buffer again after applying one-pole filter algorithm.

```
01:{
02://blksize is the signal block size for 1 dsp cycle
03:var blksize = Server.local.options.blockSize;
04:var coef    = -0.95;
05:var obuf    = LocalBuf(blksize);
06:var ibuf    = LocalBuf(blksize);
07:var in, out, lastout, insig;
09://write input (white noise) to the buffer
10:insig = WhiteNoise.ar(1);
11:insig = insig * 0.5;
12:BufWr.ar(insig, ibuf, Phasor.ar(0,1,0,blksize));
13://access each sample via built-in objects.
14:blksize.do {
15:    arg i; //the loop index.
16:    in  = BufRd.kr(1, ibuf, i);
17:    out = ((1- abs(coef) * in)) + (coef * lastout);
18:    BufWr.kr(out, obuf, i);
19:    lastout = out;
20:};
21://recover a sample block from the buffer
22:BufRd.ar(1, obuf, Phasor.ar(0,1,0,blksize));
23:}.play
```

**Figure 3**. One Pole Filter Example in SuperCollider

The other gap found in Figure 1 is that there is no counterpart in the conceptualization of musical time-scales to sample-block in SuperCollider. This gap is likely the cause of this usability problem (the user must understand the concept of 'sample-block' and write a loop to process each sample one by one). It should be also noted that this concept of 'sample-block' is normally hidden to the user. The user is expected to have the knowledge how

SuperCollider's synthesis framework is implemented to write this code while it does not appear in many other programs.

```
01:{
02:var coef    = -0.95;
03:var in, out, lastout;
04://apply one-pole filter to input (white noise)
05:in = WhiteNoise.ar(1);
06:in = in * 0.5;
07:out = ((1- abs(coef) * in)) + (coef * lastout);
08:lastout = out;
09:out;
10:}.play
```

**Figure 4**. A Proposed Redesign

As above, we assessed this usability problem in SuperCollider as caused by two conceptual misfits. One is the lack of accessibility to samples and the other is the incompatible abstraction of 'sample block' in SuperCollider. If these two conceptual misfits are the causes of usability problems, recovering the accessibility to each sample and removal of the 'sample-block' abstraction should improve the usability in SuperCollider. One experimental redesign for this problem is to perform DSP sample-by-sample and make each sample directly accessible. Figure 4 shows a sample code of this proposed design. Clearly, the code is much terser and simpler than the original.

### 4.3. Potential Usability Problems

The other conceptual misfits that can be suggested from Figure 1 is that there is no counterpart to micro time-scale in SuperCollider. As shown, there is no counterpart of micro time-scale in the representation of time-scale in SuperCollider's abstraction layers. This misfit is not limited to the design of SuperCollider, but frequently seen in many computer music programming languages. As expected from this misfit, how to model microsound synthesis techniques, which is conceptualized as a technique in this micro time-scale, is one of the major concerns in synthesis framework design and implementation. Some previous works discuss this difficulty as in [2, 9].

## 5. CONCLUSION

We have described our approach that considers the abstraction layers both in the user's conceptualization and the language design, combining three perspectives proposed by the previous research. Even though our approach is still quite simple and just assesses conceptual misfits based on the abstraction layers, such an assessment can be a beneficial "broad-brush" analysis for the usability problems in language design, which can be considered important for the further improvement of usability in DSLs.

## 6. REFERENCES


[1] Aarron, S., Blackwell, A.F., Hoadley, R. & Regan, T. "A Principled Approach to Developing New Languages for Live Coding," in *Proc of NIME'11*, 2011

[2] Bencina, R. "Implementing Granular Synthesis", *Audio Anecdotes III*, A K Peters, Ltd., 2007

[3] Blackwell, A.F., & Collins, N. "The Programming Language as Musical Instrument", in *Proc of PPIG'05*, 2005

[4] Blackwell, A.F., Church, C. & Green, T.R.G. "The Abstract is 'an Enemy': Alternative Perspectives to Computational Thinking", in *Proc of PPIG'08*, 2008

[5] Blackwell, A.F. & Green, T.R.G., "Notational Systems - the Cognitive Dimensions of Notation Framework", in *HCI Models, Theories and Frameworks: Toward a Multidisciplinary Science*, Morgan Kaufmann, 2003

[6] Blandford, A., Green, T.R.G., Furniss, D. & Makri, S. "Evaluating System Utility and Conceptual Fit Using CASSM", in *Intl Journal of Human-Computer Studies*, Vol.66, pp.393-409

[7] Blandford, A. & Green, T.R.G. "Ontological Sketch Models of Schedule Organisers", in *Proc of a Workshop on Understanding Work and Designing Artefacts*, 1998

[8] Blandford, A., Green, T.R.G. & Connell I. "Formalising an Understanding of User-System Misfits", in *Proc of EHCI-DSVIS'04*, 2004

[9] Brandt, E. "Implementing Temporal Constructors for Music Programming", in *Proc of ICMC'01*, 2001

[10] Détienne, F. *Software Design - Cognitive Aspects*, Springer Verlag, 2001

[11] Lee, E. "Computing Needs Time", *Communications of the ACM*, Vol.52, No.5, 2009

[12] Nishino, H. "Misfits in Abstractions: Towards User Centered Design in Domain-Specific Languages for End-User Programming", in *Proc of SPLASH'11*, 2011

[13] Roads, C. *Microsound*, The MIT Press, 2004

[14] Shaft, T.M. & Vessey, I. "The Role of Cognitive Fit in the Relationship between Software Comprehension and Modification", *MIS Quarterly*, Vol.30, No.1, pp29-55, 2006

[15] Wilson, S. et al. *The SuperCollider Book*, The MIT Press, 2011