Cognitive Issues in Computer Music Programming

Hiroki NISHINO
Graduate School for Integrative
Sciences and Engineering
National University of Singapore
g0901876@nus.edu.sg

ABSTRACT
Programming Languages are the oldest 'new interface for music expression' in computer music history. Both composers and researchers in computer music still have considerable interests in computer music programming environments. However, while many researchers focus on such issues as efficiency, new paradigm, or new features in computer music programming, cognitive aspects of computer music programming has been rarely discussed. Such 'cognitive issues' are of importance when design or usability in computer music programming must be considered. By contextualizing computer music programming in the psychology of programming, it is made possible to borrow the technical terms and theoretical framework from the previous research in the field, which would be helpful to clarify the problems related to cognitive ergonomics and also beneficial to design a new programming environment with better usability in computer music.

Keywords
Computer music, programming language, the psychology of programming, usability

1. INTRODUCTION
Computer Music languages have been playing significant roles in musical creation since the birth of computer music in its history. Computer music programming is also very interesting in that computer music is at least one of the first fields, where a programming language was designed for artists as end-users, even when people hardly had access to computers. Even the design of *Music V*, one of the earliest computer music languages developed at Bell Telephone Laboratories, was enough `comprehensible` for musicians of that time without professional skills in computing, as seen in [14]. Since then, programming languages for musicians has been one of the main interests both from researchers and artists to explore the possibility of new territories in computer music.

Yet, the cognitive aspects of computer music programming have rarely been discussed in computer music community. The usability issues are seldom justified by the previous research in the psychology of programming and mostly supported only by the programming concepts or rather practical experience.

Such a lack in contextualization of the cognitive aspects of computer music programming can be significant obstacles for further research in usability issues.

By borrowing the technical terms and the theories from the previous research in the psychology of programming, the problems in computer music programming can be clarified so that the future research can be more beneficial to improve the designs of programming languages and environments for better usability in computer music programming activity.

2. RELATED WORK
In this section, we briefly describe the previous research in the psychology of programming so to contextualize computer music programming by the related work in the later section.

2.1 What is a Computer Program?

2.1.1 The surface structure and the deep structure
From a psychological point of view, *the surface structure* and *the deep structure* of a computer program must be distinguished. While the surface structure is about *textual structure* or how *surface units* are arranged in a program, the deep structure is based on the relations and the abstraction in a program, such as control flow, data flow and hierarchical organization of goal and sub-goals. A computer program is multi-dimensional in that it contains different types of deep structures.

2.1.2 Mental model
Mental model is a traditional approach in HCI to explain the understanding and reasoning by users about the system. Halsz and Moran's paper on mental models of a simple calculator is one of the traditional examples [12]. Mental model approach is also extended to programming languages. Détienne describes *"learning a programming language consists, therefore, in acquiring not only the syntax of language but the rules of operation of the virtual machine underlying it"* [9, p.17].

2.2 Program Design

2.2.1 Problem domain and computing domain
Program design has been studied mostly as problem-solving activity and considered to be composed of three phases; a programmer has to understanding a problem first. Then, research and development of the solution is conducted. Finally, he codes the solution. However, in the real world situation of programming design, programmers go back and forth between these phases as well as other design activities do.

2.2.2 Ill-defined problem
Program design activity is generally considered 'ill-defined', the characteristic of which is "*one that addresses complex issues and thus cannot easily be described in a concise, complete manner*" [18]. The goal of an ill-defined problem is often vague and some constraints and criteria may not be recognized at the beginning. For instance, a programmer may be able to notice that some specification is missing only after he started the design and in the course of implementing the solution for the specification, new constraints may be added to other parts of the problem.

Furthermore, there can be several different solutions for one ill-defined problem and there is hardly an objective true-or-false evaluation. Instead, the solutions can be evaluated by good/bad or appropriate/inappropriate assessments.

2.2.3 Software design activity

We briefly describe three different theoretical approaches to explain software design activity, i.e. knowledge-centered approach, strategy-centered approach and organization-centered approach. Detailed explanations can be found in [9, 13].

Knowledge-centered approaches focus on hierarchically organized knowledge stored in memory and programming activity is considered as activation of schemas; programmers utilize available schemas and combining them to solve the programming problems.

Strategy-centered approach focuses on the strategies that programmers take to solve the problem. For instance, in a problem that consists of hierarchically ordered sub-problems and sub-sub-problems, a programmer may work on top-down or bottom-up. The programmer may work on the end part of program first, then goes back to the beginning (forward vs backward development), or work bread-first or depth-first in hierarchical organization of sub-problems.

Organization-centered approach corresponds to the organization of the design activity and there are two models for this approach. One is the hierarchical model, influenced by structured programming, which models programming activity as problem solving of top-down, breadth-first searching process for a solution. On the other hand, the opportunistic model is based on the empirical studies on how a programmer deviates from hierarchical model; a programmer may write the part of the solution that they think is most crucial, not in top-down, breadth-first order. Green and his colleagues describe and explain such a behavior in [10].

2.3 Program Comprehension

2.3.1 Program Text Comprehension

As in programming design activity, several different approaches exist to explain program text comprehension. The theoretical framework of program text comprehension is largely based on natural text comprehension and there are several different approaches as in the case of programming activity.

In structural approach, superstructures (or a generic structure of a program) can play a significant role in comprehension process. Rist explains that the basic structure is made of input, calculate and output [19] and structural schema on such a basic structure guides the comprehension.

Détienne tried experimental validation of a functional approach, according to which program comprehension is processed top-down by activating knowledge schemas [8]. She also described the importance of mental model approach, in which "*to understand a program means to construct a detailed model of the situation"* as *"a theoretical approach that potentially has the predictive and explanatory power to account for how the comprehension activity is determined by the task"* and unlike the other models, mental model "*reflects the entities of the problem domain and their relationships, that is to say, the problem goals and the flow of data."*[9, pp93- 103].

2.3.2 Rules of discourse

Rules of discourse also play a significant role in program comprehension. Some rules of discourse can activate program schemas as in functional model in the previous chapter. Mullen tries to explain the importance of the rules of discourse by several other factors, such as chunks, split-attention effect, analogical reasoning etc. [15]. We pick up and briefly describe some of the examples by Mullen here below.

'Chunk' is "*a collection of memory elements having strong associations with one another, but weak association with elements within other chunks*" [15]. One of the rules of discourse that programmers share is separating each meaning full groups of code each other. Grouping the parts of the program together according to how mind chunks the related elements can help understanding of the code; e.g. the code can be easily understood if blank lines separate a group of four lines, which initialize one object, from the other part of the code.

Another rule of discourse is to keep the size of functions reasonable and not to distribute them sparsely in the different files as possible. Mullen explains this by the split attention effect, which makes the information difficult to comprehend by occurrence of indirection. For an example, *if text that supports a picture is presented separately from the picture it is more difficult to comprehend/learn than if the text were displayed meaningfully upon the picture itself* [15]. In a program text, if a part of code contains a lot of function calls to very small functions that are distributed among many different locations in the code, such a part of the code can cause lots of indirection and penalty for short-term memory, resulting in the split attention effect to decrease comprehensibility of the program.

Thus, the rules of discourse that programmers share can be also endorsed by the theoretical framework and play significant role in program comprehension.

2.3.3 Cognitive Fit

Cognitive fit theory developed by Vessey is the theory on the correspondence between the task performance and the representation format. For instance, *graphical representations emphasize spatial information while tables emphasize symbolic information* [21] and then a symbolic task can be performed better with tabular representation than with graphical representation and vice verse. Thus, fit and gap between a task and the representation of information is a significant factor in comprehension.

Some study reports the effect alike also in a textual programming language. Green showed *nested conditionals favored sequence information ("Given this input, what happens?)* and Gilmore and Green found that *a more declarative programming language gave improved access to circumstantial information* ("*Given this result, what do we know about the input?"*) [11].

2.3.4 Dual-task interference

Simultaneously working on two tasks can cause the interference between the given two tasks and the performance can be relatively worse than when each task is processed one after the other, not simultaneously [17]. Such dual-task interference has been observed between many different activities.

Shaft and Vessey considered the modification task of a program as dual-task interference situation and cognitive fit between comprehension and modification [20].

2.4 End User Programming

End-user software engineering or end-user programming is even considered as '*the most common form of programming in use today*' [2] and becoming an important research topic both in HCI and software engineering community. End-users who program for everyday work may not be expert in programming but they certainly are expert in their professions. Such an end-user is called a 'domain-expert end-user' or simply 'expert end-user'.

As Blackwell describes, "*an important characteristic of end-user programming research is that end-user programmers should not be regarded as "deficient" computer programmers, but recognized as experts in their own right and in their own domain of work. They might only write programs occasionally or casually, but it is possible that they have done so for many years'*, and thus 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'* [1].

3. COGNITIVE ISSUES IN COMPUTER MUSIC PROGRAMMING ACTIVITY

In this section, we contextualize several aspects of computer music programming in the framework of the related work described in the previous chapters and also propose several interesting characteristics of computer music programming.

3.1 Program Design

3.1.1 Ill-defined problem, exploratory design, and the aesthetics of failure in computer music

Computer music programming also shares lots of characteristics with general programming activity and many problems in computer music are also ill-defined as in other programming activity. Yet the fact that the goal of a program design tasks is often composing a new computer music piece also bring some more interesting issues to be considered.

The constraints in ill-defined problems may be vague or even unrecognized at all when the program design activity is begun. Moreover, a goal of computer music programming is mostly a new computer music piece and this program design activity is highly exploratory a lot more than general programming tasks. A Composer may completely change the goal of the programming tasks; He might begin programming tasks with a short piece for tape in mind, but during his exploration, he may completely change the original plan and start writing for piano and interactive system. Even a bug or an error that a composer encounters can change the whole goal of the programming task. Cascone describes such a creative 'failure' in [4].

Such a highly exploratory design activity in computer music programming should be considered as a significant characteristic in designing a new programming environment.

3.1.2 Two languages in one environment

As mentioned in the previous section, a programmer is assumed to have the mental models of a device. One of the special characteristics in computer music programming, especially of textual computer music programming languages, is that they often mix two different programming paradigms into one language, each of which is based on a different mental model; while the synthesis models are normally declaratively defined, the other part of computer music programming language are usually based on different paradigm, such as instrument-score style, imperative programming or object-oriented programming.

While this feature may facilitate problem-solving on most of problems in computer music programming, it also may cause difficulty if the problems lies across the boundary of both domains of two languages.

3.2 Program Comprehension

3.2.1 Program Text Comprehension

Computer program is multi-dimensional and this is also true to computer music. Interestingly, computer music programming adds one more deep structure that is not in general purpose programming – musical structure, such as phrases, structures, forms, timbre, and the like. How to deal with this musical dimension should be highlighted as a significant factor in usability of computer music programming.

For instance, chunking the group of notes in one phrase in a c-sound score file may help the comprehension of the phrases so to recover the mental representation of the score, but such chunking also significantly damage to represent the relationship between the notes in different phrases; e.g. chunking one phrase in two voices of counterpoint makes it harder to grasp vertical relationship between the notes in two melodies while the melody in one voice can be clear described.

Recovering such deep structure of music contents in a program may be a difficult task, yet improvement in programming language syntax may be potentially beneficial to help recovering the musical representation from a program text.

However, if the musical contents are generated by certain algorithms and not explicit in the program text, it can even be almost impossible to imagine the musical output of the program, since mental or situational models related to musical events can be hardly recovered only by program texts.

3.2.2 Cognitive fit and cognitive styles

Carter and his colleagues described a *cognitive style* of composers in [3], relating it to the information processing strategy that the composers take. For instance, as for one of the characteristics called global/analytic, which corresponds to the composers' composition approaches; Those composers characterized as global tend to compose plan for the pieces they are working on, whereas other type of composers characterized as intuitive, in a more improvisatory approach. Such tendencies of global/analytic cognitive styles seem to correspond to the strategy-centered approach in design activity, such as top-down/bottom-up, breadth-first /depth-first strategy.

Dannenberg refers to cognitive styles in [7], to describe his work on the Nyquist composition environment, however, some aspects of the work seem to be more suitable in the framework of cognitive fit theory, rather than cognitive style. For instance, generally speaking, the shape of an ADSR envelope is much easier to grasp when it is visualized by a graphical representation than when it is described by a textual representation such as the list of floating-point values, whereas the exact duration of the the sustain in the same envelope is more comprehensive when the actual floating-point value is explicitly shown in the list, rather than estimating the duration by looking at the graphical representation of the envelope. Such cognitive ergonomics in graphical/textual representation can be easily explained by cognitive fit theory rather than by cognitive style.

Also as Green and his colleagues described in [10], programing activities by programmers in the real-world situations can be highly opportunistic. Such opportunistic behavior can be more significant especially when computer music programming is highly exploratory as described in the previous section. Even when a composer with global cognitive style work on the certain programming tasks, his programming activity can hardly be truly top-down.

Such issues as cognitive fit and strategic approach in programming activity should be considered important for further discussions on usability analysis of computer music programming environments.

3.2.3 Dual-task interference in live-coding

Live-coding would be an extreme type of computer music programming activity. Live-coding musicians perform their music, programming on-the-fly on the stage, sometimes even writing the code from the scratch. Nilson describes "*live-coding*

can demand producing functioning code to a strict time limit, to find ways to introduce or modify code with low latency" [16] and other paper by Collins and his colleague describe "*You forget the current audio or just take too long while you prepare the next section"* [6]. While the former description corresponds to the restriction on available time for coding given to a live-coding performer, the later also corresponds to the cognitive overload.

Such a nature of live-coding would be an unusual, but interesting case of dual-task interference. Certainly, listening to music in the professional level and writing code with the strict limitation in time are quite different mental activities, both of which consume considerable cognitive resources. Furthermore, modification task of existing code is often involved in live-coding performance and such modification task alone can be also considered as dual-task [20], as described in the previous chapter; The interference between multiple tasks can occur in live-coding and it is an interesting example to be discussed.

3.3 End User Programming

When placed in the thread of end-user programming, computer music programming is one of the most major, historical domains of expert end-user programming. Computer musicians are clearly an example of expert end-users, in that they have strong expertise in music domain but much less in computing. As described in [5], even in those days when the non-experts, who are not computer scientists, hardly had the access to computers, programming languages for computer music was being developed and composers with less expertise in programming had been invited to compose his musical pieces, using those tools and languages for computer music compositions. Furthermore, computer music programming is also an exceptional field even as expert end-user programming in that it already has a considerably long history and there are many end-users with the domain-expertise in music, a lot of who are educated in academic education of their expertise or with professional experience for many years.

Computer music programming as expert end-user programming activity also seems to be an ideal situation when we consider one of the traditional criticisms made to some of psychological studies on programming activity that the problem size is too small and far from the real world situations in which the programmers work in software industry. The problem in computer music is usually fairly small but still deals with the practical problems in their expertise domain of music.

4. CONCLUSION

Cognitive aspects of computer music programming have been rarely discussed in computer music community. Yet, by borrowing the theoretical framework and technical terms mainly from the psychology of programming, it can be made clear what kind of issues are in common with general programming activity and what are special characteristics in computer music programming. Such a contextualization can help clarifying the problems in computer music, to improve the design and the research on programming language design. Furthermore, computer music is likely to be very interesting as a topic in the psychology of programming, as Blackwell describes in [1].

Characteristics of computer music programming seem to be interesting and also beneficial to study on the usability of programming language design. For instance, how to utilize the expertise in music domain for cognitive ergonomics of programming languages is an interesting issue and the nature of creative activity with open-ended goals in computer music programming is also an interesting subject when we consider how programming environments should support exploratory design activity.

5. ACKNOWLEDGEMENT

This work was supported by project grant NRF2007IDM-IDM002-069 from the Interactivity and Digital Media Project Office, Media Development Authority, Singapore.

6. REFERENCES

[1] Blackwell A. and Collin, N. The programming language as a musical instrument, *Proc of PPIG05* (2005)

[2] Burnett, M. et al, End-user software engineering. *Communications of ACM, Vol. 47(9)* (2004)

[3] Carter, J. et al. An Analysis of Interviews with Composers From A Cognitive Styles Perspective. *Proc of ICMC'09*, (2009)

[4] Cascone, K, The Aesthetics of Failure: "Post-Digital" Tendencies in Contemporary Computer Music, *Computer Music Journal*, *Vol. 24(4)* (2000)

[5] Chowning, J. Fifty Years of Computer Music: Ideas of the Past Speak to the Future. *Proc of ICMC'09* (2009)

[6] Collins, N. et al. Live coding in laptop performance, *Organized Sound, Vol. 8 (3)* (2003)

[7] Danneberg D.,The Nyquist Composition Environment: Supporting Textual Programming With A Task-Oriented User Interface, Proc of ICMC'08, (2008)

[8] Détienne, F. Programming Understading and Knowledge Organizaiton, *Cognitive Ergonomics: Understanding, Learning and Designing Human-Computer Interaction*, pp.245-256. (1990)

[9] Détienne, F. Software Design - Cognitive Aspects. *Springer Verlag* (2001)

[10] Green, T.R.G et al. Parsing and Gnisrap *Proc of Empirical Studies of Programmers 2nd Workshop* (1987)

[11] Green, T.R.G and Petre, M. When Visual Programs are Harder to Read than Textual Programs, Proc of ECCE6, (1992)

[12] Halasz F. and Moran T.P. Mental models and problem solving in using a calculator. *Proc of CHI83* (1983)

[13] Hoc J.-M et al. Psychology of Programming, *Academic press* (1990)

[14] Matthews M.V. et al. The Technology of Computer Music. *The MIT Press* (1969)

[15] Mullen, T. Writing Code for Other People: Cognitive Psychology and the fundamental of good software design principle. *Proc of OOPSLA'09* (2009)

[16] Nilson, C. Live coding practice. *Proc of NIME'07* (2007)

[17] Pashler, H., Dual-Task Interference in Simple Tasks: Data and Theory. *Psychological Bulletin Vol. 116* (1994)

[18] Reed D. The use of ill-defined problems for developing problem-solving and empirical skills in CS1. *Journal of Computing Sciences in Collges, Vol.18 (1)* (2002)

[19] Rist, R. Plans in Programming: Definition, Demonstration, Development, *Empirical Studies of Programmers 1st Workshop*, 1986

[20] Shaft, T. and Vessey, I. The role of cognitive fit in the relationship between software comprehension and modification. *MIS Quarterly, Vol.30 (1)* (2006)

[21] Vessey, I. Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision Sciences, Vol. 22(2)* (1991)