Computer Assisted Music Instruction: 
Toward a More Viable Philosophy of Software Architecture

Joseph Lukasik
College of Music
The University of Colorado
Campus Box 301
Boulder, Colorado 80309 USA

Abstract: The purpose of this paper is to examine certain shortcomings in the design philosophy of many computer assisted music instruction programs presently available, and suggest ways these programs might be improved by incorporating contemporary pedagogical theories into future software architecture. Recent accessibility of object-oriented programming languages and expert systems make possible the creation of an authoring system which could produce software that provides a high level of interaction with the student and greater flexibility for the instructor. One possible system is described in the body of this paper.

Of the many major advances in software technology over the last decade in such areas as artificial intelligence and object oriented programming, surprisingly little has filtered down into commercially available computer assisted music instruction (CAMI). College level music programs in the U.S. have not begun to integrate CAMI into their curricula in any large scale fashion because instructors tend to find the programs rigid, and sometimes contrary to their own pedagogical beliefs. The preconception of the computer as drill master or a "magic black box" which mechanically dispenses a "correct" or "sorry, your answer is incorrect" as the only response to student input is, unfortunately, largely a result of software currently in use. Barton Barrie, in his 1987 overview, Computer Software in Music Education: A Guide1, observes that the drill conformation is the most common variety of music instruction software in spite of more interactive forms. One might have expected the nature of CAMI to have changed since the publication of Barrie's book; however, the results of a recent survey conducted by the University of Missouri-Kansas City Conservatory of Music2 show that a majority of college and university music programs in the U.S. involved with CAMI make use of the drill type of software.

One could generalize the emphasis of contemporary CAMI in the following manner: to elevate the student user's level of skill in a specific task, for example interval identification, through the use of drill and practice techniques. The word skill in this sense can be taken to mean "competent, expert, rapid, and accurate performance"3. Stress is placed on learning a skill at hand through practice and drill. It is hoped that this activity, when mastered, will transcend the boundaries of a drill-specific context into real-world problem solving, crystallizing assorted bits of declarative knowledge into a effective body of procedural knowledge. In many instances, such a transition is merely a happy accident at best, but it would be preferable to achieve these desired results by design.

If we begin with the notion presented in Card, Morgan, and Newell's The Psychology of Human-Computer Interaction that all cognitive behavior has in itself "a dimension of skill, so that any cognitive behavior is more or less skilled"4, in other words, learning is a skill as well as using knowledge in problem solving, two conclusions can be drawn, a) that learning factual and procedural knowledge is at least as important as learning how to learn, and b) that learning how to learn is itself procedural knowledge. These two conclusions should be considered among the basic pedagogical premises of any new music instruction software.

Different students utilize various problem solving strategies, some more effective under certain circumstances than others. Obviously, successful students use the correct strategies at the proper times. Those students who are disorganized in their manner, or confused as to which strategy might benefit them in a given situation, need attention. The new software should be dynamic in nature, possessing a sensitivity to disclose which students require special assistance and cognitive behavior modification. This can be accomplished by having the computer measure student performance against an "ideal" pattern. The "ideal" can be entered by the instructor or assembled by the software itself from actual successful performance patterns. The program should

ICMC GLASGOW 1990 PROCEEDINGS
record every decision, change of mind, query, and time lapsed between every action in order to identify a student's deficiencies. Once a serious problem is determined, the program shifts emphasis toward more rudimentary concepts, or provides a tutorial underlining the particular pain of cognitive behavior required to complete the lessons successfully. For example, if the student exhibits continued difficulty in part-writing exercises, the software may guide the student through this set of procedures to eliminate some of the difficulty:

1.) complete the soprano line
2.) complete the bass line
3.) write the final cadence
4.) write prominent middle cadences
5.) identify tonic harmonies
6.) identify dominant harmonies
7.) identify passing harmonies

etc.

As a first step in defining a software ideal, certain needs of the users must be weighed. Since the needs of students tend to vary on an individual basis, instructors might not assign the same program to all of their students. In addition, some students may require supplementary work with other software to round out their deficiencies. This can create a great deal of extra work and frustration for students and faculty alike if a new user interface must be encountered along with each new program. Commonality among software user interfaces should be a goal of software developers especially now that Macintosh style user environments are becoming popular front-ends for all major computer hardware configurations.

Almost all of the software available today does not allow instructors to input their own examples, but instead offer pre-programmed or randomly generated questions which progress too quickly in their degree of difficulty, or are unmusical as in the case of randomly generated examples. Instructional software must be designed to allow the instructor to author all questions, giving the instructor complete control over the quantity and degree of difficulty of the material presented. Dr. Jill developed for use at the University of Colorado at Boulder, is a HyperCard stack which functions both as an authoring tool and instructional drill program. Though it follows the drill courseware formats, it is capable of evaluating a student's progress, and navigating the student on to more remedial exercises and tutorial material if the semester so wishes. Most importantly, its modular architecture allows the instructor to input a large number of exercises in any sequence. The sequence can be changed easily with simple Macintosh cut and paste procedures. In Day 11 of each exercise consists of a single screen frame or card in which resides a complete authoring tool for recording the exercise question via M.I.D.I input, along with text input for the answer. Once the instructor enters material for the exercise, the authoring tool is hidden from the student's view, the card is copied (except for the exercise material) and pasted after itself easily for another exercise to be inserted. Each card contains two simple M.I.D.I. sequencers, one for the teacher's example, one for the student's response, and a simple algorithm for evaluating the difference between the two. Both sequencers are capable of reproducing the minutest subtleties of musical performance. [Software example. Day 11 for the Macintosh will be demonstrated!]

In many schools, the content of car training classes often follows the content of the written skills classes closely. As with many theory textbooks, the sequence of topics discussed and the manner in which concepts are emphasized are vital to an understanding of the "larger picture" these texts are trying to present. More often than not, the manner of presentation in educational software works against the conceptual sequence established in the texts. Since it is impossible for software manufacturers to intuise what texts will be used at a given institution, the software architecture should allow latitude in determining the conceptual sequence as the instructor's discretion.

On-line tutorials are important components of any educational software if the software is to be truly effective. Because pictographs and dictionary definitions are not enough to help students conceptualize their way through a problem, care must be taken that levels of interaction are present. The tutorial in Set Logic, developed at the University of Colorado suggests one approach. Students having difficulty finding the normal order of a pitch class set could call up a screen from Set Logic, and, at their own speed, work their way to an answer along with the computer. Since finding normal order is a multi-task procedure, students have the opportunity to check their work
at each step of the process, and find the source of difficulty. [Software example, 
set logic for the 
Macintosh will be demonstrated]

One possible configuration for new CAMI software is embodied in the proposed Orpheus project to be developed at the University of Colorado's College of Music. Orpheus will be a 

media-driven authoring system capable of producing interactively interactive coursework for music theory. Perhaps the most appealing aspect of Orpheus is that authors will no longer be burdened with learning programming techniques in order to create coursework capable of intelligent interaction, as is the case with earlier authoring systems5.

Although there are many excellent overviews of object-oriented programming languages (Flurry, Krasner, and Lieberman)3, a review of some basic principles of object-oriented languages (OOPAs) is useful in understanding the advantages they would give the software engineer who might program 

an interactive computer assisted teaching system. With traditional programming languages, the act of programming is done by writing down a list of procedures necessary to complete a task. The procedures are carried out in a static order specified by the programmer. The data needed to carry 

out the procedures is kept apart from the procedures themselves. However, with OOPAs tasks can be 

approached as a collection of sub-tasks which can be performed in any order depending on the 

circumstances. For example, the task of harmonizing a melody in four parts can be divided into 

the sub-tasks described earlier in this paper as the components of a part-writing tutorial. These 

sub-tasks are called objects. They carry with them all the information needed to function as well 

as instructions defining how they should interact with other objects; data and procedure are 

integrated. This is perhaps the most important characteristic of OOPAs. In order for an object to 

act, it must receive a command called a message from another object. An object receiving a 

message might choose to execute a task, send another message, or do nothing. Even messages can 

be considered objects with their own modes of behavior.

A group of objects which respond to a given message with like behavior patterns is called a 

class. Consider a real-world class of objects called road_transportation. This class would 

contain the objects car, stationwagon, truck, and bus, each of which would respond the 

same way to the message go. The response to a message is called a method. Consequently, objects 

belonging to the same class respond to the same messages and execute the same methods.

Single objects are referred to as instances of a class. Since a class is considered an object, 

an entire class can be viewed as an instance of a larger class. This hierarchical organization of 

objects provides a natural medium for another important characteristic of OOPAs, inheritance. 

Inheritance guarantees that objects inherit behavior from the super-class to which they belong, as 

seen in the relationship between the class road_transportation and its instances. It also allows the programmer's choice of having to write the procedural code for every instance in a class; 

code is only written to break the hereditary chain.

In the following example, the message evaluate instructs the object Exercises to 

calculate student's score from a model performance kept in memory. However, the object class 

Exercises does not possess a method to act directly on data, so it passes the message on to its 

two sub-classes, Student_answers and Student_response_times, which in turn send the 

message on to instances belonging to the sub-classes.

Class: Exercises

Sub-class: Student_answers

Instance: Melodic interval identification

Instance: Harmonic interval identification

Instance: Triad quality identification

Instance: Seventh chord identification

Instance: Chord progression identification

Sub-class: Student response times

Instance: Time until correct answer

Instance: Time until incorrect answer

Instance: Time until first change of mind

Instance: Time until second change of mind

Polymorphism is an extremely useful attribute of OOPAs. It allows a single message to be 

interpreted in numerous ways by different objects. An example of Polymorphism is as it might

ICMC GLASGOW 1990 PROCEEDINGS
appear in Orpheus would be the mechanism for defining the level of interaction in new courseware. The author desiring a high level of interaction between student and software selects the appropriate button on an authoring tools menu. This sends the message see_level_of_interaction(high) on to four objects, consider_answers, consider_times, record_times, and put_tutorials_on_line. Consider_answers evaluates the number and type of incorrect answers and decides whether or not to send a student to tutorials (if any are available) based on several model student performances. Consider_times might send a student to tutorials if the time spent answering questions seems inordinate. Record_times will keep a record of the time it took a student to answer correctly, incorrectly, and change his or her mind to another answer. Put_tutorials_on_line would ask for either a yes or no argument from the author and go on to link pre-existing tutorials to the courseware or provide the author with an opportunity to create tutorials as the question material is entered.

Dynamic binding is a fundamental characteristic of OOPs which can greatly simplify the task of modifying software as needs arise. It means that an object will wait to decide what method is an appropriate response to a given message until that particular object runs. It allows the ability to assign any new object to a pre-existing variable, and on a larger level, the ability for programmers to add new classes of objects to existent software. This flexibility is cost-effective and insure a prolonged future for software developed in this manner.

Past courseware systems allowed only on-screen or musical keyboard input for student response. Orpheus will provide a variety of tactile musical interfaces for student users. Musical keyboard, percussion, and wind controller M.I.D.I. (Musical Instrument Digital Interface) devices enable students not comfortable with keyboard technique to respond to some of the lessons with greater ease. An analog pitch-to-M.I.D.I. converter will act as the computer's ears to evaluate sight singing exercises, a skill building exercise required of all music students, and allow acoustic instruments, such as the violin, to interface with courseware. A M.I.D.I. time clock converter will interface an electronic baton with the computer. Students will be able to conduct music played by the computer, while the input from the baton is analyzed to determine the student's level of rhythmic and metric perception.

Orpheus is intended to run on the Macintosh environment requiring only a few pieces of onboard musical equipment which are relatively inexpensive and readily available. A goal of the Orpheus project is to make a powerful authoring medium accessible to a large number of instructors by designing it for a commercially available hardware platform.

It is hoped that some interest in interactive music instruction software is generated by this paper, and that the previously mentioned advantages of object-oriented software architecture encourage others to explore new courseware system design.

4ibid.
5Course of Action. Bloomington, MN: Authorware, Inc. 1987, is a prime example of existing authoring systems. In this case, an intermediate knowledge of programming techniques is needed to achieve effective results. Music capabilities were added to Course of Action as an afterthought, and leave much to be desired.
9Flurry, M.