Scud

(Simplified Composition Development)

A Framework for the Development of Computer-Assisted Composition

Jesse Guessford

I. Purpose

"Reusable software components do not simply emerge as a by-product of software development. Extra effort must be added to generalize system components to make them reusable."[1]

The purpose of this project is to program a reusable framework for the creation of computer-assisted compositions. The framework consists of a core group of objects that are normally required with each new composition system. Too often, composers recreate these objects from scratch each time with only slight variations. The framework developed in this project will help alleviate such repetitive coding. The group of objects that are produced through this project can be modified and extended to create a large variety of objects that will fulfill different functions in many different software systems.

The framework being developed is named SCuD or Simplified Composition Development. Throughout the development of the SCuD framework, three goals will be kept in mind; ease of use, reusability, and extendibility. To show that these goals have been achieved, two separate software systems will be built from the SCuD framework, fulfilling the three goals. These two systems will highlight the ease of building a composition system with SCuD and how easily the framework can be extended to meet the composition systems needs.

There already exist many different synthesis and sound creation tools. However, there are very few ways to use different systems all at once. This framework will also provide a method for composers to create a single composition system that utilizes several different synthesis systems, notation programs, and sound creation tools. Michael

Hamman and Simon Pamment state that "while there are a number of programming libraries for sound synthesis and for generating musical patterns, there are few which emphasize the kind of software architectural flexability that allows composers to investigate experimental notions of musical structure without getting mired in endless difficulties."[2]

II. Background

A. Why Smalltalk?

The framework for the project is written in VisualWorks 7. This is a version of Smalltalk built by the Cincom Corporation[3]. There are several reasons to choose Smalltalk for this project. The first reason is the portability of the language. Smalltalk is available on the Macintosh, Windows, Linux, and several other UNIX based platforms. A framework written in this language would be portable to each of these operating systems with little change. Having a framework tied to a specific platform causes a steep reduction in the amount of usability and reusability that the system will have over its lifetime.

Another reason for the use of Smalltalk is that the language is object-oriented. Smalltalk is a completely object driven programming language. This allows a user to easily inherit and extend any piece of the framework's code. The fact that every class and class variable is an object, allows the framework to be easily extended. There are

---

[1] Ian Sommerville, *Software Engineering*, (Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1995), 417.

[2] Hamman, Michael and Simon Pamment, *A C++ Framework for Generic Programming and Composition*, http://www.shout.net/~mhamman/papers/cim2000_c++_g.htm, accessed June 2003, 1.

several features that constitute object-oriented programming languages: encapsulation (every data type is strongly associated with the functions that operate on the data type), inheritance (where classes can be defined as a refinement or a specialization of other classes), and polymorphism (the same function name can evoke different behaviors).[4]

Smalltalk also has a large amount of graphical user interface (GUI) development built into the language. The GUI objects are easy to use, with graphical palettes available for the creation of windows and many objects available for drawing and window building. The user interface is based on the Model-View-Controller paradigm. G. Krasner and Stephen T. Pope's "A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80" is an excellent source on this topic.[5]

Stephen T. Pope gives a list of criteria necessary in a programming language to create a successful framework:[6]

- Expressiveness and terseness in the programming language.
- A large and well-documented class library, including reusable high level modules.
- Appropriate analysis and design methodologies.
- A powerful, abstract programming language and a run-time compiler.
- A windowed interactive user interface 'shell' text and or menu system.
- An integrated set of development, debugging, and code management tools.
- An interface to 'foreign language' calls (e.g. C functions).
- A software framework for constructing interactive graphical user interfaces.

Smalltalk meets all of the requirements set out by Pope. In fact, Smalltalk is the language that Pope chose to create his own music framework.

---

[3] http://www.cincom.com

[4] Stephen Travis Pope, "Object-oriented Music Representation," *Organized Sound* 1, no. 1 (1996): 55-57.

[5] G. Krasner and Stephen T. Pope. "A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* 1, no. 3 (1988).

[6] Pope, "Object-oriented," 58-59.

B. What is software reuse?

Software reuse is the re-implementation of already existing ideas into a new piece of software. Ian Sommerville points out that engineers design components based on smaller components that are "tried and true," such as nuts and bolts, as well as larger scale items like engines and condensers.[7] Software reuse is similar in that programmers are reusing "tried and true" pieces of software to design and build new software packages. Software reuse is an attempt by programmers to create objects that can be used again in a new system.

Patrick A. V. Halle in "Software Components and Reuse – Getting More Out of Your Code" states that there are four ways to reuse software:[8]

1) Reuse of Ideas
   (The publication of algorithms, methods and techniques to others.)
2) Vertical Reuse
   (A major source of reusable code provides an abstraction above the level of hardware. This is commonly found in libraries.)
3) Horizontal Reuse
   (Within the level of abstraction, an example of this is the unix pipe |.)
4) Total Reuse
   (Entire packages are reused with little or no modifications.)

The four items above refer to how the existing code will be re-implemented. There are also four different levels refering to the amount of code that is re-used. The four levels of reuse are application system, sub-system, module or object, and function.[9]

The benefits of software reuse have been well documented. The following is a list of five benefits of reuse in software development given by Agresti and McGarry[10]:

[7] Sommerville, 396.

[8] Patrick A. V. Hall, "Software Components and Reuse – Getting More Out of Your Code," in *Tutorial – Software Reuse: Emerging Technology*, ed. Will Tracz (Washington D.C.: Computer Society Press, 1988), 13.

[9] Sommerville, 396-397.

1) Productivity – The use of preexisting objects causes a reduction in effort.
2) Reliability – The use of previously proven components injects reliability into the new system.
3) Consistency – The use of some components in many places reduces the possibility of bad design.
4) Manageability – The use of well-understood components makes the code easier to read and modify.
5) Standardization – The use of standard components are in place early to help developers.

These are general benefits to any programmer regardless of the system. For the composer creating a computer-assisted composition, the chief benefits lie in items one through three of Agresti and McGarry's list. By utilizing previously written pieces of software, the composer can spend more time in creating the new elements of the program. With this gain in time also comes a gain in reliability. The composer does not need to retest the already designed portions of his program for bugs or for design flaws.

Another way to look at these objects is through an approach by T. Biggerstaff and C. Richter in "Reusability Framework, Assessment and Directions."[11] In this paper, the authors divide reusable technology into two categories: composition technology and generation technology. Composition technology is atomic and ideally unchanged. Composition technology contains passive elements on which external agents operate.[12] Generation technologies, on the other hand, occur when patterns and routines from one program are woven into a new program. Often the new program shows little semblance to the old.[13] The SCuD framework consists of a set of composition technologies. The

---

[10] William W. Agresti and Frank E. McGarry, "The Minnowbrook Workshop on Software Reuse: A Summary Report," in *Tutorial: Software Reuse: Emerging Technology*, ed. Will Tracz (Washington D.C.: Computer Society Press, 1998), 34.

[11] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment and Direction," in *Tutorial: Software Reuse: Emerging Technology*, ed. Will Tracz (Washington D.C.: Computer Society Press, 1998).

[12] Ibid., 3.

[13] Biggerstaff , 4.

ideal user of this framework will not reuse just methods but whole objects and families of objects.

Biggerstaff and Richter point out that there are limitations to software reuse[14]:

1. Lack of a method to represent software design
2. Lack of clear and obvious method (strategy) for software reuse
   a. Multi-organizational problem
   b. Requires large number of components before there is a pay-off
3. "Not invented here", using technology designed by someone else or in another area
4. Initial capitalization, effective reuse requires a large initial commitment. The libraries of a used reusable system must be well stocked before they reach the payoff point.

The biggest limitation that will plague this project is item three. Composers tend to shy away from using software that was not "invented here." For the composer to use this framework, they will have to fight this tendency. The framework attempts to help by creating an easy method to add objects not covered by the system. With the addition of these expanded objects the system can be closer to "invented here" than an off the shelf program.

C. What is a Framework?

A framework is defined by William W. Agresti and Frank E. McGarry as a system that "employs knowledge gained through previous experience."[15] So, a framework is a system that is created through the repeated creation of similar systems. A framework provides a set of objects that work for a given problem within a specific domain. New programs that fall within this domain are able to reuse and to extend the framework to meet the needs of the new program. The important concept is that the entire framework does not need to be rewritten.

[14] Ibid., 7.

There are two types of frameworks that can be developed: a white-box framework and a black-box framework. The white-box framework focuses on reuse by inheritance. This is accomplished through the implementation of new classes that are extensions of objects already within the framework.[16] A black-box framework focuses on object composition. "New functionality is obtained by assembling or composing objects to get more complex functionality."[17] The SCuD system will be a white-box framework; new objects will be designed by inheriting the qualities of existing objects. The drawback to this type of framework is that a certain amount of knowledge about the inner-workings of the framework will be necessary for the user to create new objects. The benefit of this type of composition technology is that the user can easily extend the framework. Any aspect of the framework's construction can be altered to meet the needs of the program and the programmer.

D. Previous Music Frameworks

Pope points out that several music description languages have been developed, starting soon after the first object-oriented environments became practical.[18] These previous music frameworks will be grouped by programming language and discussed further. Each of the systems will be evaluated against SCuD in terms of the systems ease of use, reusability, and extendibility.

---

[15] Agresti, 33.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1995), 18-19.

[17] Ibid., 19.

[18] Pope, "Object-oriented," 58.

Along with the main goals of the SCuD framework, Pope gives the following list of items that each framework should contain:[19]

1. Abstract models of basic musical magnitudes.
2. Sound objects represented as functions of time, grains, or other abstractions.
3. Flexible grain size of 'Events' in terms of 'Notes', 'Grains', 'Elements', or 'Texture'.
4. Flexible hierarchical event trees for 'parts' or 'tracks'
5. Separation of 'data' from 'interpretaion' (what vs. how).
6. Abstractions for the description of 'middle level' musical structures (e.g. chords, clusters, and trills).
7. Annotation of events supporting the creation of hierarchies.
8. Graphical annotation (including common practice notation).
9. Description of sound synthesis and processing models such as sound file mixing or DSP.
10. Converters for many common formats (i.e. midi, music V).
11. Parsing live performance into some rendition of the representation and of interpreting it in real-time.

Of these eleven items, some will create a system that is counter to the goals set out in the development of SCuD. Items 9 and 11 will both increase the complexity of the system and will also cause reusability issues. Both of these items will require extensive knowledge of the system hardware and software. Because of this the ability to port the software environment to another system will be decreased sharply.

Of the remaining items, the SCuD system achieves all but the first. Currently, there are very few basic musical magnitudes built into the system. This element of the framework will be discussed further in the conclusion of this paper.

i) C++ Frameworks

There are two frameworks that will be discussed designed in C++. Some of these frameworks have only been proposed, while others have been fully implemented. Michael Hamman and Simon Pamment wrote the first of these frameworks. The framework discussed in the paper from 2000 could not be found, however much of the

[19] Ibid.

design seems to have become Michael Hamman's *Orpheus*.[20] However, this software environment is no longer a framework, but a software program. The original purpose of the framework is given as:

> "The goal of this programming framework is to provide a small but extendible set of interfaces for developing composition and synthesis programs ranging from the production scores to real-time sound synthesis applications. The design of the framework emphasizes ease of application development without sacrificing representational flexibility."[21]

The goals set out for the design of this framework are similar to that of SCuD. This design statement follows closely the elements listed by Pope for a successful music framework. But, this is also where this framework design could have difficulty. With the expansion of the framework into real-time synthesis the scope of the framework greatly increases and the ability for the user to understand the code and the code's chance or reuse both decrease.

In this system three basic object types are proposed: algorithm object, signal generator, and renderer.[22] The difference between this system and SCuD is that SCuD is based off of a single object. This makes the system easy to use and easily extendible.

The second C++ music framework is Synthesis Toolkit (STK) written by Perry Cook. The framework can be downloaded at http://www-ccrma.stanford.edu/software/stk/download.html. The system is described as

> "a collection of roughly 60 (as of May, 1996) classes in C++, designed for the rapid creation and connection of music synthesis and audio processing systems. Primary attention has been paid to cross-platform functionality, ease of use, instructional code examples, and real-time control."[23]

---

[20] http://www.shout.net/~mhamman/projects/orpheus.htm.
[21] Hamman, 1.
[22] Ibid., 3.
[23] Perry R. Cook, *Synthesis Toolkit in C++ Version 1.0, May 1996, SIGGRAPH 1996,* http://ccrma-www.stanford.edu/software/stk/papers/stksiggraph96.pdf, accessed June 2003, abstract.

The first problem with this statement is that a system that with over 60 classes becomes difficult to understand and use. As systems grow they usually expand, therefore by 2003 the number of classes has increased to 83. Also, there are over 700 files included in the download of this system. In contrast to STK, a download of SCuD will contain one file, including documentation, and have around 30 classes. The lower number of objects allows SCuD to be more understandable and extendable for the average user. STK has tried to stay portable, as of July of 2003, there was a compiled binary for Windows, Linux, and Macintosh OS X.

This system, like the system written by Michael Hamman, divides all of the objects into three catagories: unit generators, algorithms, and control signal and user interface handlers. Once again, the SCuD system is based on one basic object, making the system easier and more extendible.

At the framework's core, the system is designed for the manipulation of synthesis algorithms.[24] SCuD is not intended to perform this task at all. The purpose of SCuD is to control outside synthesis engines and sound producers through manipulation of a score. Not, as the Synthesis Toolkit is designed, to manipulate sound synthesis directly.

ii) Java Frameworks

Two different music frameworks written in Java will be discussed. Peter Hanappe developed the first of these frameworks. The Varèse system was built out of the work Hanappe did for his Ph.d. dissertation. The system can be downloaded at http://www.ircam.fr/equipes/repmus/Varese/index.html. The Varèse system is written in Java, however a Scheme interpreter is embedded into the system for user interaction. So,

[24] Ibid., 1.

to use this system, the user will be required to learn two languages, Java and Scheme. Hanappe justifies his use of two languages by stating that:

> "The scheme interpreter is implemented on top of the Java platform, one single object system and one single memory strategy is used in the environment. This promotes a transparent use of functional objects throughout the system. In particular, functional objects are used extensively to describe complex behaviors and relations."[25]

Hanappe asserts that the use of the two languages is transparent and will promote the extendibility. I feel that a single language, in the case of SCuD Smalltalk, used for the development of the framework will provide the same amount of extendibility while cutting the languages necessary in half, and ultimately increasing the ease of use for the framework.

Additionally, the system currently only works in Linux and SGI based IRIX operating systems.[26] Documentation for the system is also a problem. The only documentation currently available is Hanappe's dissertation.[27]

JMusic is written by Andrew Sorensen and Andrew Brown and can be downloaded at http://jmusic.ci.qut.edu.au. There are several different compiled versions of the system and if a compiled version is not available there are java compilers for almost all operating systems. They describe the system as a computer-assisted composition environment designed to 'assist the compositional process by providing an open but partially structured environment for musical exploration."[28] They further

---

[25] Peter Hanappe, "Design and Implementation of an Integrated Environment for Music Composition and Synthesis," Ph.D. diss. University of Paris, 1999, 2.

[26] *The Varèse Environment*, http://www.ircam.fr/equipes/repmus/Varese/index.html, accessed July 2003.

[27] Ibid.

[28] Andrew Sorensen and Andrew Brown, *jMusic: Music Composition in Java. An Introduction to jMusic*, http://jmusic.ci.qut.edu.au/jmtutorial/t2.html, accessed June 2003.

describe the project as "a library that is simple enough for newbie programmers but sophisticated enough to enable composers to accomplish real work."[29]

Upon download the user is confronted with more than 180 files in the compiled version, compared to the one in the SCuD system. However, most of these files are synthesis instrument files and do not have a direct effect on the system. Here, like STK, the goal of the framework is different than SCuD. jMusic is attempting to create a framework built around synthesis, not the score based approach used by SCuD.

iii) Smalltalk Frameworks

The DMIX system, by Daniel Oppenheim, was started as a proposal for a framework in C++ called *P-G-G*. DMIX is implemented in Smalltalk, however the system can not be download as of July 2003.[30]

The P-G-G system was unique in that "rather than linking the graphics to sound parameters they are linked to control parameters."[31] In this system the composer would be able to represent each sound with a graphic image and alter that image overtime, therefore altering the sound. Oppenheim listed some of the possible advantages of *P-G-G*:[32]

1. Sound parameters presented as musical data and in a comprehensive fashion.
2. Composer no longer obliged to 'translate' his musical ideas into a rigid syntax of a music interpreter language.
3. Composer receives both audio and visual data.
4. 'Context-minded' – all elements of the music may be viewed along with their relationships to each other.
5. Powerful composition tools provided. New ones can be added.
6. Generality and extensibility.

---

[29] Ibid.

[30] *Dmix*, http://www.research.ibm.com/mathsci/cmc/dmix.htm, accessed July 2003.

[31] Daniel Vincent Oppenheim, *The P-G-G Environment for Music Composition – A Proposal*, http://www.research.ibm.com/mathsci/cmc/papers/pgg87.pdf, accessed June 2003, 2.

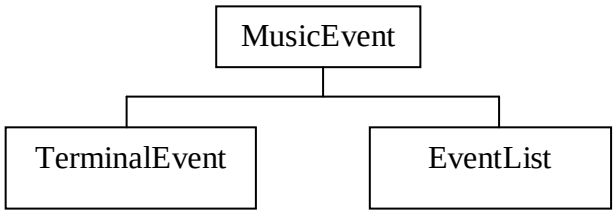
[32] Ibid., 7.

7. Modification of the music through the graphics speeds up the composition process.
8. Ability to embed subroutines into the graphics.
9. A high degree of interactivity encourages experimentation.
10. User-friendly.

Unfortunately, since this environment was just a proposal, a working version of this framework was not available for analysis.

However, this proposal did spawn the *DMIX* framework. "[The] main motivation [of the framework] was to design an easy-to-use and yet flexible environment that has an uniform user-interface, that is easily extendable, and that is independent of my synthesis hardware."[33] In this statement, Oppenheim touches on many of the same goals as the SCuD system, ease of use, extendable, and reusable. Another similarity between the two frameworks is that both are based on a single object. In the case of *DMIX* the object is MusicEvent.[34] The rest of the objects are subclasses of this abstract class. The inheritance structure of MusicEvent can be seen in Figure 1.

Figure 1 Inheritance Tree of MusicEvent[35]



This split of MusicEvent mimics the structure created in SCuD. The objects that create the heirarchy structure, in this case EventList, are separated from the sound events, TerminalEvent.

[33] Daniel V. Oppenheim, *DMIX: An Environment for Composition*, http://www.research.ibm.com/mathsci/cmc/papers/dmix98.pdf, accessed July 2003, 1.
[34] Ibid., 3.
[35] Ibid.

With the similarities between the systems, it would be interesting to further explore this system. However, since the system can not be downloaded currently, a more detailed look at *DMIX* is impossible.

The final two frameworks for computer-assisted composition in Smalltalk are MODE and Siren, both written by Stephen T. Pope. Both of these frameworks are powerful systems that have many capabilities designed into the framework. The MODE framework has the following main components:[36]

- A language for the representation of musical parameters, sounds, and event lists.
- Objects for the "middle level" musical structures.
- Real-time midi, sound I/O, and DSP scheduling classes.
- A user interface framework.
- Several built in end user tools

There are several differences between MODE and SCuD apparent from this list. First, MODE attempts to provide a system in which changes to sounds and events will effect a real-time change in the aural result. In contrast, the SCuD system does not even attempt to create any real-time objects. Instead, the purpose of SCuD is to provide a system to create score files. These score files are then used by other software systems. There are no end user programs included in this framework, only a GUI interface. The final difference is that MODE was originally written in Smalltalk-80, a variety of Smalltalk that is different than VisualWorks. MODE is no longer available and has been replaced by SIREN.

The SIREN system was designed with the following features:[37]

---

[36] Stephen Travis Pope, *The Musical Object Development Environment: MODE (Ten Years of Music software in Smalltalk*, http://www.create.ucsb.edu/~stp/publs.html#MODE, accessed February 2003, 1.

[37] Stephen Travis Pope, *The Siren Music / Sound Package for Squeak Smalltalk*, http://www.create.ucsb.edu/~stp/publs.html#MODE, accessed February 2003, 1.

- The Smoke music representation system (music magnitudes, events, event lists, generators, functions, and sounds).
- Voices, Schedulers, and I/O Drivers.
- User interface components for music applications.
- Several built-in applications (editors and browsers for siren objects).

The part of Siren that is most like SCuD is the Smoke music representation system. Pope explains that Smoke has two related music input languages: a compact binary interchange format and a group of concrete data structures.[38] Similar to SMOKE, SCuD attempts to create a representation system that uses both a binary storage method and a collection of concrete objects.

The specifics of the Smoke system are given by Pope:[39]

- Abstract models of musical quantities (scalar, duration).
- Instrument / note abstractions.
- Sound functions, granular descriptions, and other non-note oriented descriptions.
- Event, control, and sampled sound description levels.
- Hierarchical event tree structures.
- Separation of data from interpretation.
- Abstraction for "middle-level" music structures (chords, clusters).
- Annotation including common practice notation.
- Description of sampled sound synthesis processing models.
- Possibility for building converters for many common formats.
- Possibility for parsing live performance into Smoke and for interpreting it in real time

The list above is an impressive list of functionality contained within the SMOKE system. The Smoke system far outreaches the scope of SCuD. However, this list of functionality adds complications for the user. One of the major goals of the SCuD framework project is to provide the user with simple, easy to use objects. With the added functionality of SMOKE listed above, comes a larger learning curve for the user to understand and extend the basic framework.

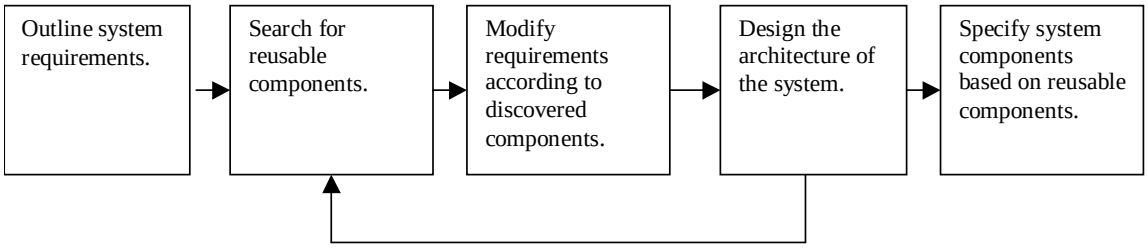
[38] Ibid.

III. Objects

A. Methodology

Figure 2 is a diagram of the design model that was followed for the creation of this framework.

Figure 2 Main Methodology[40]



The first step of this methodology was to define the system requirements. The following is a list of the requirements defined for this framework:

- Easily extendable objects for the creation of new events.
- Easily designed GUI that can be easily modified for new objects.
- Storage of objects within a tree structure.
- Basic set of composition tools (magic square creation, matrix formation, random number generators).
- Basic musical structures defined (chords, melodies).

With the requirements of the system established, the next step was to search for components (objects).

There were two ideas that I had in mind during this initial search for reusable objects. First, each object has to perform a well-defined function.[41] Second, the internal state of each object must be encapsulated. Encapsulation requires that only operations on an

---

[39] Ibid.

[40] Sommerville, 399.

[41] Tony Williams, "Reusable Components for Evolving Systems," in *Fifth International Conference on Software Reuse Held in Victoria, British Columbia, Canada 2–5 of June, 1998*, ed. Premkumar Deuanbu and Jeffery Poulin, (Los Alamitos, CA: IEEE Computer Press Society, 1998), 14.

object can change the internal state of the object.[42] If an object is not encapsulated, other object types directly modify the internal state of the object. The system requirements were then modified to work with these newly designed components.

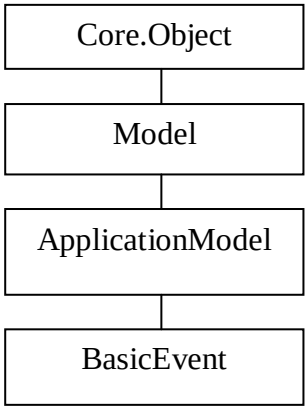
For the architectural design phase of this project, four main ideas were addressed in order to keep the system reusable. Any user of the system will be required to find the components, understand the components, modify the components, and compose new components based on existing components.[43] If the user is capable of easily performing these tasks, the framework will be successful due to reduction in effort and the elimination of errors.

B. Objects

The highest level object made for the SCuD framework is BasicEvent. The inheritance diagram for this object is given in Figure 3. An inheritance diagram shows the lineage of an object's definition. The objects at the bottom of the diagram will inherit all of the instance variables and methods contained in all of the objects connected to that object. The instance variables on an object are the location in which an instance of that object will hold specific information. The methods associated with an object are the algorithms performing specific functions on the instance variables of that object. For Example, all the classes found below Core.Object, in Figure 3, are considered sub-classes of Core.Object.

[42] Gamma, 11.
[43] Biggerstaff, 5.

Figure 3 Inheritance Diagram for BasicEvent



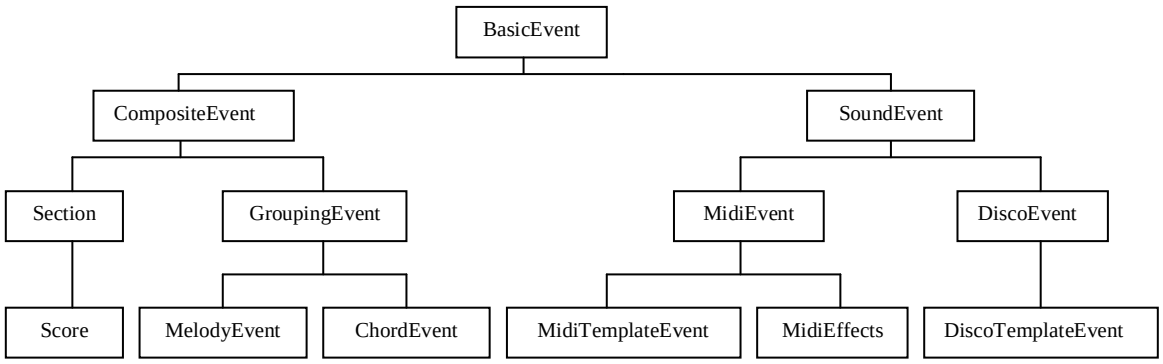
The BasicEvent object becomes the parent of a large portion of the other objects in SCuD. When the framework needs to be extended, the programmer will need to have knowledge of this object. Of the objects listed in Figure 3, this is the only object created for this framework.

Table 1 Object Definition for BasicEvent

BasicEvent
Instance Variables startTime duration parent
Instance Creation Methods startTime:duration: Instance Variable Set/Retrieve Methods startTime startTime: duration duration: parent parent: Other Methods TimeSpan <= printOn: eventCollect: type addStartTime: addDuration:

Table 1 shows that an instance of BasicEvent will hold the instance variables: startTime and duration. Both of these variables contain an instance of FixedPoint. Fixed point numbers are used because some of the synthesis systems in SCuD require fixed point calculations, removing any chances of a rounding error. This object contains a fraction, which is translated into a number with a fixed number of decimal places. Additionally, several methods are included in BasicEvent to access and modify these instance variables, thus maintaining the encapsulation of the object. BasicEvent will also hold the parent instance variable, which will be explained later.

Figure 4 Inheritance Diagram for BasicEvent



The inheritance diagram for BasicEvent found in Figure 4 is complete for the current system. However, for the user to extend the system, they will be creating sub-classes of BasicEvent.

The left side of this inheritance diagram contains storage events. These objects are needed to create the storage tree which is a structure in memory containing many instances of different objects. The tree structure is comprised of an object containing instances of other objects that are sub-classes of BasicEvent and called children. These children also have access to the original object, or parent. The right side of Figure 4 contains the sound representation events included in SCuD. These events hold

information specific to the creation of a sound. Currently in SCuD, there are two types of sound events: Disco and Midi. The Disco sound objects will create a score for use with the Disco system, designed at the Computer Music Project, Experimental Music Studios at University Of Illinois.[44] The midi objects will create and output a midi file. The SoundEvent object is where users of this framework will probably be focusing when extending the functionality of SCuD. This will be done by creating new types of sound objects.

The CompositeEvent object adds an additional instance variable, eventCollection. This variable is an instance of SortedCollection, a standard Smalltalk object that holds a set of objects in an order determined by <=. This is where the elements of the storage tree are deposited. Each subclass of CompositeEvent uses this instance variable to store instances of other objects that are themselves sub-classes of BasicEvent. CompositeEvent adds a new child by placing the child in the eventCollection instance variable. The parent instance variable, found in Table 1, of that child is then pointed to this storage object. This pattern is based on the composite pattern found in *Design Patterns: Elements of Reusable Object-Oriented Software*.[45]

Figure 5 is an example of an instance of Score. The tree structure starts with an instance of Score and places other storage and sound events within the eventCollection instance variable contained within Score.

[44] Hans G. Kaper, Sever Tipei, and Jeff M. Wright, "DISCO: an Object-oriented System for Music Composition and Sound Design," in *Proceedings of the 2000 International Computer Music Association in Berlin, Germany August 2000,* (San Francisco, CA: International Computer Music Association, 2000).

[45] Gamma, 163-174.

Figure 5 Example of a Score (Storage Tree Diagram)

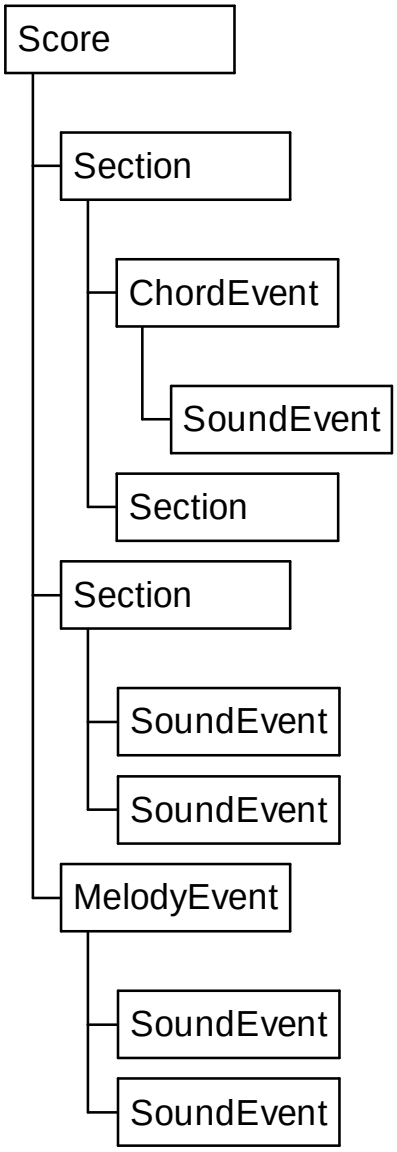


Table 2 lists each of the structural objects found in Figures 4 and 5 along with the object's instance variables and a brief explanation of their purpose and use.

Table 2 Storage Objects

Object	**User Assigned Instance Variables**	**Explanation**
BasicEvent	startTime duration	Base object, details in Table 1.
CompositeEvent	eventCollection	The base object for this subset of storage objects.

Object	User Assigned Instance Variables	Explanation
Section		This is an object that can store several different types of events. The events can be chords, melodies, sounds, or other sections.
Score		This is the top level of the storage tree. This object contains the functions needed to save and export the tree structure. This object has overwritten the startTime variable, so it is always 0.
GroupingEvent	soundTemplate	This is an abstract object. The objects that are sub-classes of this object can contain instances of SoundEvent only. These instances of SoundEvent can contain all the needed information, or they can contain only part of the information. The additional information is stored in the soundTemplate instance variable, which contains an instance of a GroupingTemplate. This information is used for all the incomplete SoundEvents contained within that GroupingEvent.
MelodyEvent		This object is similar to a section, except that it can only contain SoundEvents.
ChordEvent		All the SoundEvents stored in this object have the same startTime and duration values.

These are the objects that comprise the storage objects used by this framework. These objects can be used as is or extended to meet the specific requirements of a composition.

The representations of sounds are defined using a different inheritance diagram. However, the tree begins at the same place as the storage objects, BasicEvent. The base object for the representation of sounds is SoundEvent, a subclass of BasicEvent. The object contains all the base functionallity that any instance of a sound representation would require. Table 3 shows the definition of SoundEvent.

Table 3 Object Definition for SoundEvent

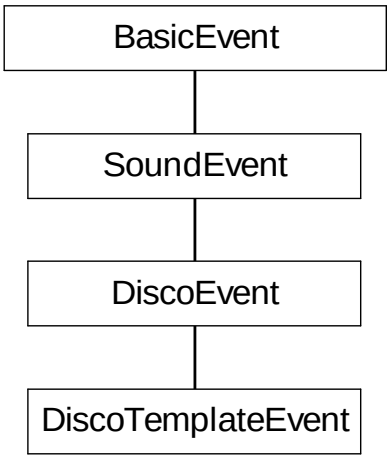
SoundEvent

Instance Variables
eventType
Instance Variable Set/Retrieve Methods
eventType
eventType:
Template Methods
template
templateType
Other Methods
printOn:
eventCollect:

This class is abstract. An instance of this object should not be created. If a user desires to expand the functionality of SCuD, new sound representations will be sub-classes of SoundEvent.

DiscoEvent contains all the needed information for the representation of a sound within a Disco score. The DiscoTemplateEvent contains the variables needed for the creation of a template sound used in MelodyEvent and ChordEvent. Figure 6 shows the inheritance tree for DiscoEvent.

Figure 6 Inheritance Diagram for DiscoEvent



All of the instance variables needed for both classes are stored in DiscoEvent. This is because in the Graphical User Interface an instance of DiscoEvent will be transformed into a DiscoTemplateEvent.

Table 4 DiscoEvent and DiscoTemplateEvent Object Definitions

DiscoEvent
Instance Variables loudness partialCollection numberPartials enharmonic overallEnvelope taper fundamental
Instance Creation Methods startTime: duration: Instance Variable Set/Retrieve Methods loudness loudness: partialCollection numberPartials numberPartials: enharmonic: enharmonic overallEnvelope: overallEnvelope taper: taper fundamental: fundamental Partial Creation Methods fundamental: numberPartials: enharmonic: overallEnvelope: taper: Other Methods printOn: addPartial: removePartialAt: removePartialNumber: changeId: trimPartials returnNewEvent numberPartials partialAt: partialNumber: □template □type

The instance variables of DiscoEvent are designed to hold a variety of information about a sound that will later be synthesized by the DISCO program. Some of these instance

variable are utilized in only the DiscoTemplateEvent, while others are used in both classes.

Table 5 DiscoEvent and DiscoTemplateEvent Instance Variables

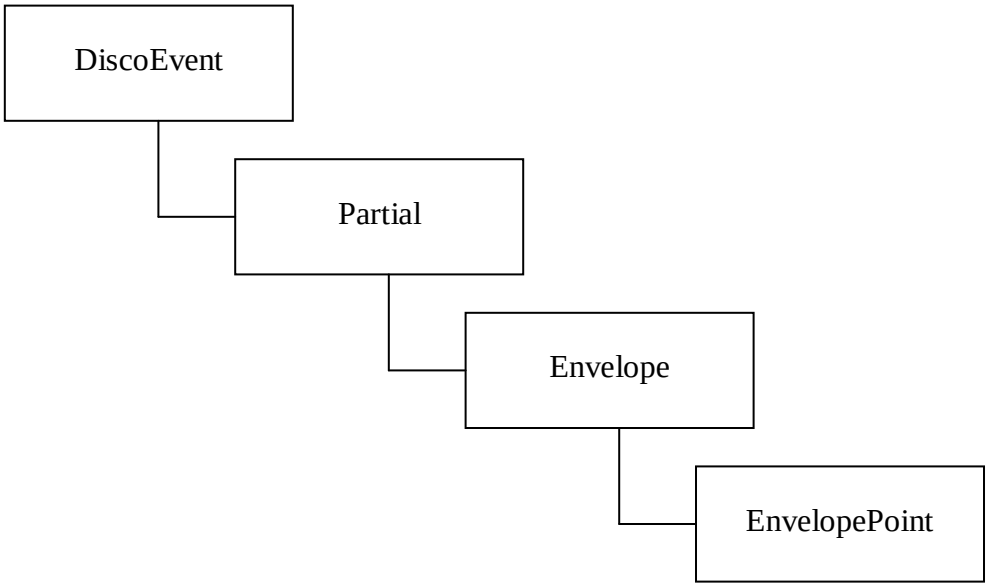
Instance Variable	Object Contained	Purpose
startTime	FixedPoint	Inherited from BasicEvent, this holds the start time of the event in seconds.
duration	FixedPoint	Inherited from BasicEvent, this holds the duration of the event in seconds.
eventType	Symbol	Inherited from SoundEvent, this holds a symbol that represents either a single or template event. This is unique for each sub-class.
partialCollection	SortedCollection of Partials	A SortedCollection storing the partials needed to create the sound. The collection is sorted by the frequency of the partial (low to high). This instance variable is used only in DiscoEvent.
loudness	FixedPoint	Stores a value from 1.0 – 256.0 that represents the loudness of the sound in sones.
numberPartials	Integer	This variable contains the number of partials found in the sound. For DiscoEvent, this number is determined from the size of partialCollection.
fundamental	FixedPoint	Stores the lowest frequency of the sound. This is the frequency of the lowest partial.
enharmonic	FixedPoint	Stores the amount that the frequencies of the sounds partials differ from the expected harmonic result.
overallEnvelope	Envelope	Stores the overall amplitude envelope for the sound.
taper	FixedPoint	Stores the rate of decay of the amplitude as the partial number increases.

The instance variable partialCollection from Table 5 is overwritten in DiscoTemplateEvent. This class does not store a collection of partials. The final five instance variables in Table 5 -- numberPartials, fundamental, enharmonic, overallEnvelope, and taper -- are used to create a collection of Partials in DiscoEvent. Several of these variables require the modification of the sound in the same method. Two methods were created for this purpose, fundamental: numberPartials: enharmonic:, and

overallEnvelope: taper:. The first method creates the partials that are added to partialCollection. The second method scales the overallEnvelope to create an amplitude envelope for each partial. For DiscoEventTemplate, these values are stored and are used later in this same manner to create a complete representation of the sound.

There are several additional objects needed to create a DiscoEvent. These objects form the hierarchy diagram found in Figure 7.

Figure 7 Hierarchy Diagram for DiscoEvent



Each of the objects in Figure 7 contains an instance variable that holds a collection of the item shown immediately below. For instance, DiscoEvent contains an instance variable partialCollection, which holds a SortedCollection of instances of Partial.

The definition of the Partial object is shown in Table 6.

Table 6 Object Definition for Partial

Partial
Instance Variables idNumber frequency staticParameters dynamicParameters

Instance Creation Method idNumber: setDynamicParameter setStaticParameter Instance Variable Set/Retrieve Methods idNumber frequency frequency: staticParameter: staticParameter: as: dynamicParameter: dynamicParameter: as: Other Methods changeID maxAmp copyDynamicFrom: copyStaticFrom: <=

The instance variables hold a variety of information that is necessary to define an instance of a Partial.

Table 7 Partial Instance Variables

Instance Variable	**Object Contained**	**Purpose**
idNumber	Integer	On export, each partial in the sound needs a unique ID number between 1 and 9999.
frequency	FixedPoint	The frequency of the partial.
staticParameters	Dictionary of Keys and FixedPoints	This dictionary contains sound elements that do not change over the duration of the sound. Each change is stored as an instance of FixedPoint. For more information see Table 8.
dynamicParameters	Dictionary of Keys and Envelopes	This dictionary contains sound elements that change over the duration of the sound. Each of these changes is stored as an instance of Envelope. For further explanation, see Table 9.

Each of the two dictionaries contained within Partial contain different values that affect the resulting sound. Tables 8 list the keys contained within staticParameter. All objects stored along with the keys are instances of FixedPoint.

Table 8 StaticParameter Keys

#VibratoPhase
#TremoloPhase
#HallSize
#ReverbTime
#ClearReverb

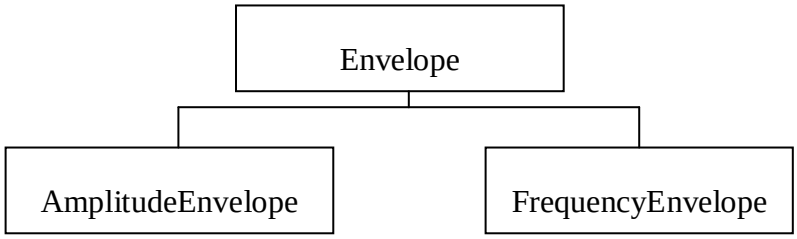
Table 9 lists all the keys contained in dynamicParameter along with the object type that is stored with each key.

Table 9 Dynamic Parameter Keys and Objects

Key	**Object Stored**
#Amplitude	AmplitudeEnvelope
#Loudness	AmplitudeEnvelope
#Frequency	FrequencyEnvelope
#Pan	Envelope
#VibratoAmplitude	Envelope
#VibratoRate	Envelope
#TremoloAmplitude	Envelope
#TremoloRate	Envelope
#ReverbMix	Envelope
#FrequencyTransient	Envelope
#AmplitudeTransient	Envelope
#RandomEffects	Envelope
#TimbreID	Envelope

All of the keys in Table 9 store instances of Envelope or an Envelope subclass.

Figure 8 Inheritance Diagram for Envelope



The basic Envelope class contains definitions for most of the functionality needed to represent an envelope shape.

Table 10 Object Definition for Envelope

Envelope
Instance Variable pointCollection
Instance Creation Methods initialize initializeBasic initializePan Instance Variable Set/Retrieve Methods pointCollection pointCollection: Other Methods addPoint: createPointAt: findPointAfter: includes: includesFirstAndLast removePointAt: rescaleX: rescaleY: shortenEnvelope shortenEnvelope: copy clear duration exponentialInterplationAt: forTime linearInterpolationAt: forTime maxPeak pointAt: size valueAt: getPointAt: isPoint:

The only instance variable in Envelope is pointCollection. This variable contains an instance of SortedCollection, itself containing instances of EnvelopePoint. Class Envelope contains most of the basic functionality needed. Each subclass adds additional functionality that is specific to the type of envelope the class represents.

For example, an amplitude envelope needs to begin and end with a “y” value of zero. To do this, AmplitudeEnvelope overrides a few of the methods from Envelope:

clear, copy, and addPoint. In addition, the method initializeAmplitude is added so that the proper envelope shape will be automatically stored in pointCollection.

FrequencyEnvelope adds three additional instance varialbles: upperYBound, upperFrequencyRange, lowerFrequencyRange. The upper and lowerFrequencyRange variables contain the upper and lower bounds of the synthesis system in hertz, represented as an instance of FixedPoint. The upperYBound represents a scaling factor that the envelope is multiplied by. In addition to these instance variables, there are several methods added to access the new variables to maintain the objects encapsulation. There are also methods to convert from a value in hertz to an appropriate value between zero and one: asFrequency: and asIntrument:.

The EnvelopePoint class is the final the object in Figure 7. The instance variable pointCollection in an Envelope object is a SortedCollection containing instances of EnvelopePoint.

Table 11 EnvelopePoint Definition

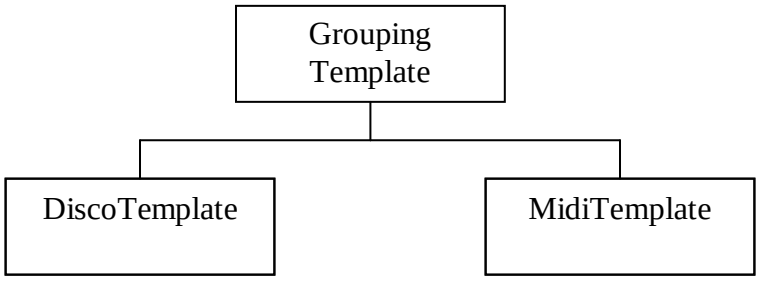
EnvelopePoint
Instance Variables x y path envelope

Instance Creation Methods x: y: path: initialize Instance Variable Set/Retrieve Methods x x: y y: path path: Other Methods printOn: precision copy

An instance of this object contains the x and y location of a point on the envelope, along with the path type to the next point. The envelope instance variable is similar to the parent variable in BasicEvent. This variable stores a reference back to the Envelope in which the EnvelopePoint is stored.

For the DiscoTemplateEvent object to work with a grouping event, the missing information about the sound must be recalled. To do this, the basic information about the building blocks of the sound is stored. In this way, the user needs to input the basic sound information only once for all the sound objects in a GroupingEvent. The Grouping Template object hierarchy is designed to fill in this missing data. For the sound event types contained within this framework, the GroupingTemplate inheritance tree looks like Figure 9.

Figure 9 Inheritance diagram for GroupingTemplate



The definition of the GroupingTemplate class is defined in Table 12.

Table 12 GroupingTemplate Definition

GroupingTemplate
Instance Variables
Instance Creation Methods Initialize Other Methods soundType

This class is an abstract class; all of the methods in this class need to be overwritten by the subclasses of GroupingTemplate class. To examine the way a GroupingTemplate is used, a sub-class of GroupingTemplate needs to be investigated.

For this purpose, the DiscoTemplate object will be examined.

Table 13 DiscoTemplate Definition

DiscoTemplate
Instance Variables partialTemplate
Instance Creation Methods Initialize Instance Variable Set/Retrieve Methods partialTemplate partialTemplate: Other Methods soundType

In this class, the methods defined in GroupingTemplate are overwritten, and two new methods are written. These methods access the newly defined instance variable in DiscoTemplate. The partialTemplate instance variable stores an instance of Partial, storing the basic information need to create additional partials for any sounds stored in the GroupingEvent.

Once the score is complete, the sounds need to be stored in files of various types.

This is accomplished through an inheritance tree of print objects. The way that these different printers work are based on the Observer Pattern as found in *Design Patterns,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.[46] The observer pattern consists of an inheritance tree of objects that act on another inheritance tree of objects. This allows both trees to extend independently.

These different print methods are stored in their own inheritance diagram. The base of this tree is the EventPrint class.

Table 14 Object Definition for EventPrint

EventPrint
Instance Variables fileStream
Instance Creation Methods printScore: printScore: with: Instance Variable Set/Retrieve Methods fileStream: fileStream Other Methods baseFileName baseFileWindowHeader

There are two methods used to create instances of an EventPrinter: printScore: and printScore:with:. The difference between the two methods is that the first will prompt the user to enter a file name, while the second has a filename passed to the method. Some of the instance methods of the object should not be overwritten by the objects' subclasses. These methods are printScore:, fileStream, and fileStream:. The printScore: method is used when the file name is not known. This method prompts the user for a file name and then sends the name along with the score to the printScore:with: method. The fileStream methods access the instance variable that is created to hold the outgoing or incoming file.

[46] Ibid., 331-344.

The methods that are overwritten by subclasses of EventPrint are printScore:with:, baseFileName, and baseFileWindowHeader. The baseFileName and baseFileWindowHeader methods are used to create the dialog box in the printScore: method. These methods place text appropriate to the object in the dialog box. The printScore: with: method begins the process of exporting the score to the appropriate file.

EventPrint objects traverse the score storage tree collecting the type of events, i.e. MidiEvents, that the printer requires. The collection is done via the eventCollect: method. This method uses double-dispatching. The receiving object sends a method back to the sender, with information about the object type, i.e. CompositeEvent. As the printer is traversing the score, it converts all usable objects into a single object type that is stored in OrderedCollection.

A list of the subclasses of EventPrint included in this framework are shown in Table 15.

Table 15 Subclasses of EventPrint

Subclass	Events Collected	Stored Object	Purpose
DiscoPrint	DiscoEvent, DiscoTemplate Event	DiscoEvent	Exports all DiscoEvents and DiscoTemplateEvents into a score file that is used by the DISCO synthesis engine. This DISCO score file is based on the I-cards of an earlier synthesis program, DIASS.
MidiPrint	MidiEvent, MidiTemplate Event	MidiEffects	Exports all MidiEvents and MidiTemplateEvents into midi file format. For printing all of the events are converted into an instance of MidiEffects.
BossPrint	All	All	Saves the entire score as a BOSS array.
BossRead	All	All	Reads in a file that was stored in BOSS array.
DiscoAsMidi Print	DiscoEvent, DiscoTemplate Event	MidiEffects	Exports all DiscoEvents and DiscoTemplateEvents in midi file format.

Subclass	Events Collected	Stored Object	Purpose
ScoreAsMidi Print	DiscoEvent, DiscoTemplate Event, MidiEvent, MidiTemplate Event	MidiEffects	Exports all events in midi file format.

The user can easily extend this list of EventPrint subclasses. Such an extension can be performed with the inclusion of new event types or for the exporting of existing types into different formats.

Two of the print objects, BossPrint and BossRead, work slightly differently from the rest. These objects are used to store the entire score. The file stores a set of binary objects that directly relate to the instance variables of the stored object. The built in Smalltalk binary storage system, or BOSS accomplishes this. These two objects contain a method readBoss:. This method converts between instances of objects and the array in which they are stored. This is the only place within the framework that all the possible object types are tested. Hence, when the framework is extended, this method needs to be modified with the addition of the new objects.

IV. Systems

A. Graphical User Interface

One computer-assisted piece of software that has been designed by using the SCuD framework is a graphical user interface (GUI) for the framework. This GUI is designed to allow the user to implement many of the features contained within SCuD with the click of a mouse button. The user has the ability to insert, manipulate, move,

and remove events from a score through an easy to use interface. The user can then save the score or export the score via one of the many event printers.

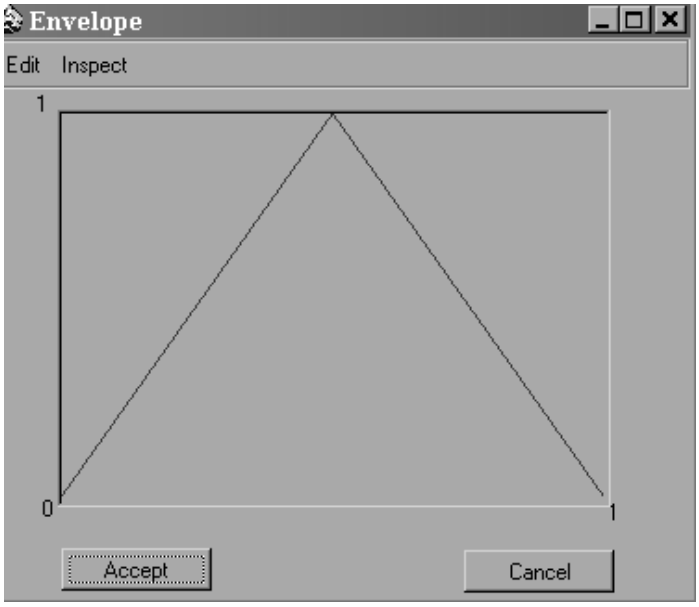
The design of the GUI is based on the Model-View-Controller paradigm. This pattern of GUI design is examined thoroughly in "Applications Programming in Smalltalk 80™: How to Use Model-View-Controller (MVC)" by Steve Burbeck[47] or Krasner and Pope's "A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80."[48] Basically, this GUI design uses a triad of objects to create the interface. The first set of objects, or the model, is examined in the Objects section of this paper. The basic building blocks of this framework become the model of the GUI interface.

A dialog window for each event is specified in that event class' definition. In most cases, this is a series of labels and fields in which the user can enter or view pertinent information. In a few cases, this simple dialog box is not enough. One of these cases is in the Envelope class. A method for drawing the envelope shape needed to be implemented and the user needs to be able to control and modify the shape.

[47] Steve Burbeck, "Applications Programming in Smalltalk 80™: How to Use Model-View-Controller (MVC)"

[48] G. Krasner and Stephen T. Pope. "A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* 1, no. 3 (1988).

Program Window 1 Envelope Window



The view holder in the center of Program Window 1 is where envelope shape is displayed.

The view for Envelope is EnvelopeView. It is a subclass of UI.View, a built-in class for use specifically with the model-view-controller paradigm. In this class, all of the methodology for the creation of the envelope shape is given. The only current problem with this view is the lack of ability for the user to visually distinguish between line segments that are linear and those that are exponential. EnvelopeView has a subclass FrequencyEnvelopeView. This class exists so that the view takes into account the change in boundaries possible in the FrequencyEnvelope class. These boundary changes have to be reflected in the scaling of the envelope shape.

The controller in this triad is EnvelopeController, with a subclass FrequencyEnvelopeController. EnvelopeController is itself a subclass of UI.Controller, another built in Smalltalk object. In this class, the functionality of the window is defined.

When the user right clicks inside of the view holder, a menu appears with the following choices:

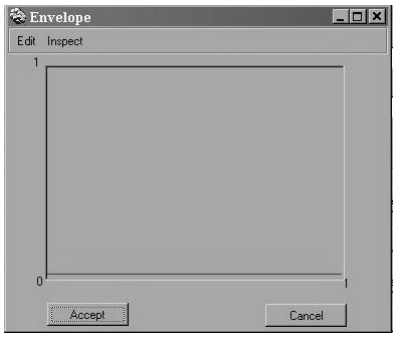
Program Window 2 EnvelopeController Menu

Add Point
Remove Point
Load Standard
    Load Basic
    Load Amplitude
    Load Frequency
    Load Pan

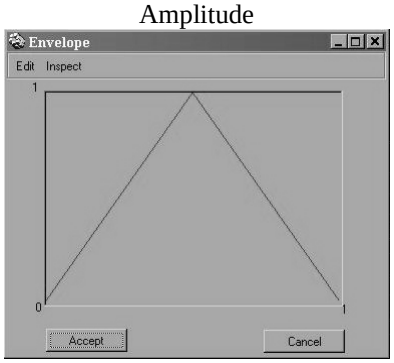
The "Load Standard" selection supplies the user with the sub-menu shown in Program Window 2. This sub-menu gives the user several choices to completely modify the envelope to one of the set shapes found in Program Window 3.

Program Window 3 Standard Envelopes

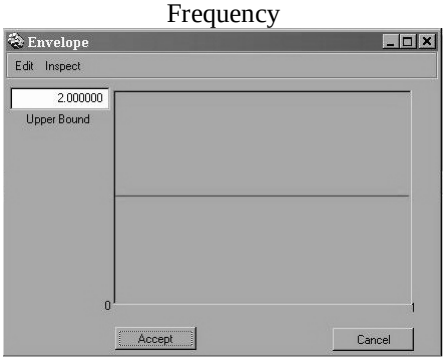
Basic



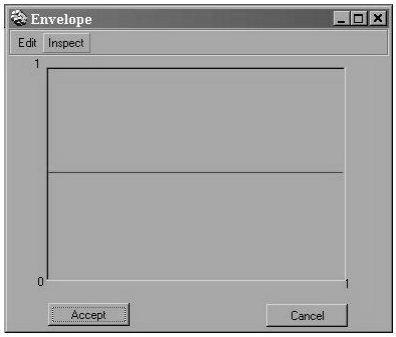
Amplitude



Frequency



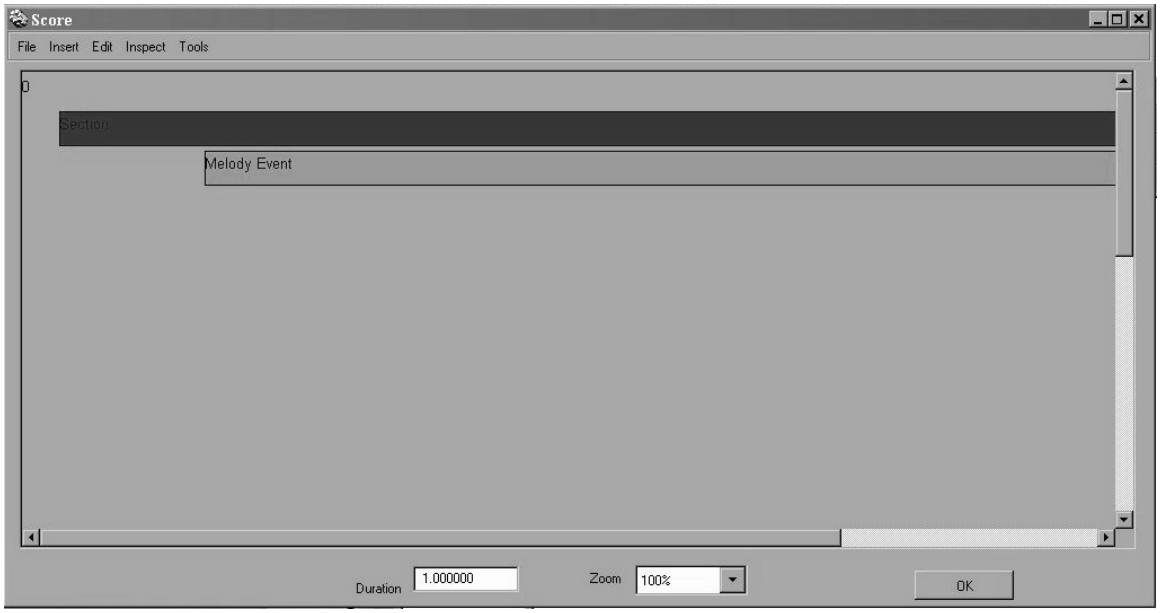
Pan



A left mouse click on any envelope point allows the user to move that point to a new location.

The other important use of model-view-controller in the GUI interface can be seen with CompositeEvent and its subclasses.

Program Window 4 Score Window



As with Envelope, Score has a dialog window that contains a view holder as seen in Program Window 4. This view holder contains a visual representation of the events contained within Score. The creation of this view is accomplished through the use of two separate view classes: ScoreView and BasicEventView. ScoreView creates a GraphicsContext on which the visual representations of the events are stored. This view also creates the timeline that appears at the top of the view. In addition to which, this class keeps the event boxes, or the representations of the events, from colliding when the events overlap in time.

The visual representation of the events are created in BasicEventView. This class creates a filled box with a title, both of which are dependent upon the class. BasicEventView works on all subclasses of BasicEvent, relying only on the methods startTime, duration, displayTitle, and displayColor. The displayTitle and displayColor

methods need to be unique for each different subclass of BasicEvent. This is to ensure that each event type will appear in a different color with a different title.

There is also a controller associated with Program Window 4. ScoreController keeps track of the interaction between the user and ScoreView. Similar to EnvelopeController, a press of the right button produces a menu of possibilities:

Program Window 5 ScoreController Menu

Insert
- Section
- Melody
- Chord
- Disco Event
- Midi Event

Open Event
Remove Event

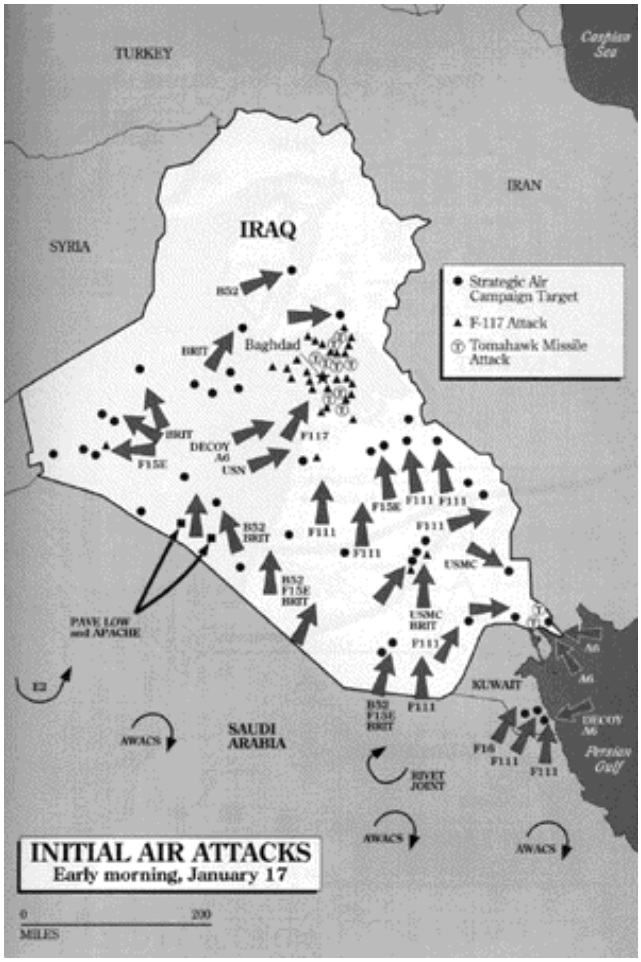
With a left mouse click the user can move events to the right or left, changing the start time of the event.

B. George I vs. the Very Bad Man from the East

The *George I vs. the Very Bad Man from the East* is a computer-assisted composition written by the author. The work was premiered in November of 2002 in the lobby of the Music Building Auditorium, University of Illinois, Urbana, Illinois using three CD players placed around the space. The maps shown in Figures 10 - 12 were placed at each CD player and in other locations around the room. The performance was repeated at the Ukrainian Art Museum in Chicago, Illinois at a concert given by the MAVerick Ensemble in February 2003. In this performance slides of each of the maps were projected on a screen in the performance space.

The system implements elements of SCuD, creating three separate instances of Score with each run of the program. The work is the sonification of three maps of Iraq and Kuwait. Two of the maps are of the Allied and Iraqi troop placements on the eve of the first Gulf War. The third map shows the locations of the first wave of bombings of Iraq by the Allied Air Command.

Figure 10 Map of Initial Air Attacks[49]



[49] http://www.pbs.org/wgbh/pages/frontline/gulf/maps/

Figure 11 Iraqi Ground Troop Placement on Eve of Gulf War[50]

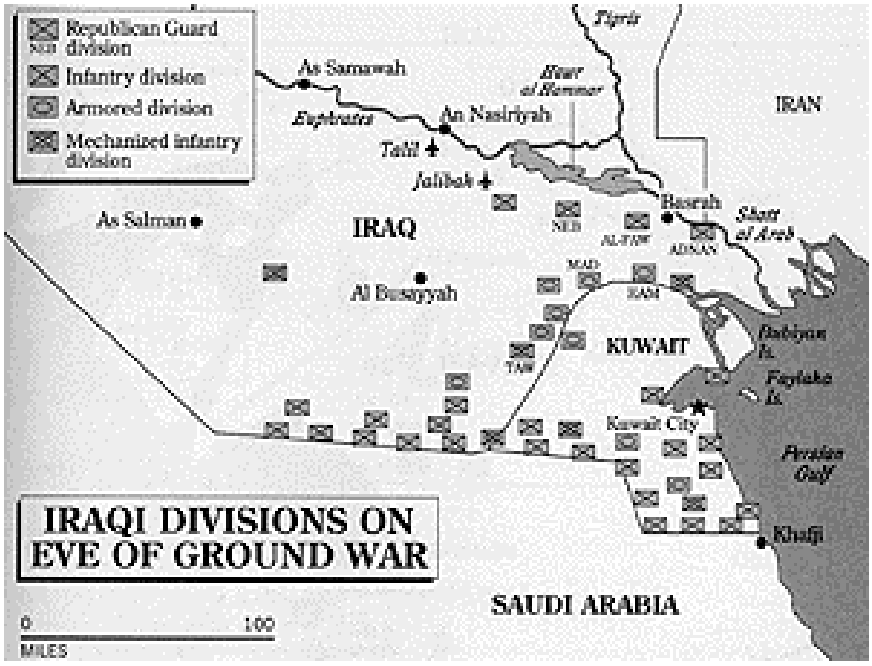
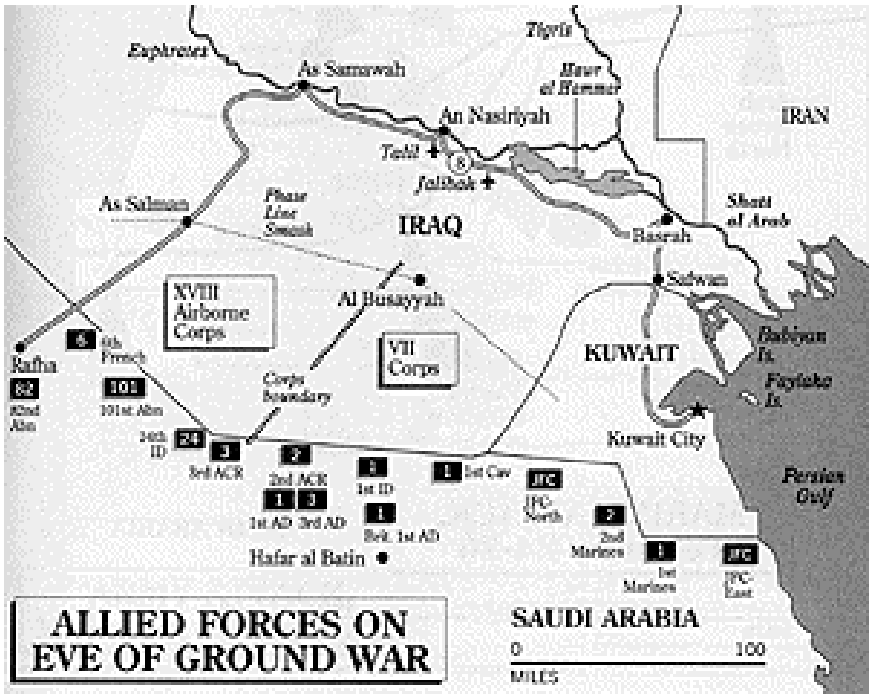


Figure 12 Allied Troop Placement on Eve of Gulf War[51]



[50] Ibid.
[51] Ibid.

Figures 10 - 12 show the maps that were used by the composer for the creation of *George I –Vs.- The Very Bad Man From the East*. Each event on the map, troop or bomb location, was transferred into a point with an "x" and "y" coordinate. Later in the program, the "x" values are transformed into start times and durations and the "y" values into frequencies.

Each of the maps has a different space in which the elements contained on the map are bounded. For the purpose of this piece, the difference in the horizontal direction was ignored. The timings of all the elements on the maps were scaled by the size of the bounding box for that map and the duration of the piece. The events corresponding to the placement of the troops found in Figures 11 and 12 were given a duration by calculating the location of the bottom left hand corner of the boxes which represent a troop. This information supplies the start time of the event, and the top right hand corner of the box supplies the duration. The events in Figure 10 were created with short durations; this is due to the nature of the symbols used on the map.

The difference of map scales was not ignored in the vertical direction. A location that was present on each of the maps was chosen to become a fixed frequency point. The frequency of this point was determined by Figure 10. Because this map covers the most landmass in the "x" direction, it will also cover the most frequency. The frequency values of each of the events are then scaled so that an event at the fixed frequency point, found on all three maps, will have the same frequency.

To determine other sound parameters, each of the elements in the maps were grouped by type. Many of the groupings already appear on the maps. For instance, all of the "Tomahawk Missile Attacks" found in Figure 10 were programmed to create similar

sounds. The sound events in this group contain reverb, transients, and panning effects. Not only are the types of modifications constant within the grouping of events, but the values of these modifications are also constant.

Because of the use of randomness during the creation of the score, this piece could be considered a manifold work. In short, this is a work that can take many forms. With each run of the program, a new aural result is created, even though it has come from the same computer-assisted composition. For a more in depth look at manifolds, please refer to "Manifold Compositions - a (Super)computer-assisted Composition Experiment in Progress" by Sever Tipei.[52]

V. Future Developments

One of the needed additions to SCuD is a set of magnitudes for pitch. Magnitude is the super-class of all the numbers in Smalltalk. Sub-classes of Magnitude share several common characteristics. The values can be sorted and transformed through simple mathematical operations.

Another set of magnitudes would cover rhythm. These magnitudes would provide an alternative to always working in minutes and seconds. Instead, the user could enter a quarter note or a half note. Later, during the export process, the sound object would use the magnitude to convert such notes to minutes and seconds. This set of magnitudes would also provide simple mathematical operations on the values. For example, a quarter note plus a quarter note equals a half note.

---

[52] Sever Tipei, "Manifold Compositions - a (Super)computer-assisted Composition Experiment in Progress" ,in *Proceedings* of the *1989 International Computer Music Conference held at the Ohio State*

These magnitudes would be similar to Smalltalk magnitudes in that they "represent partially or fully ordered scalar or vector quantities with numerical or symbolic values."[53] Other similarities would occur between these new magnitudes and the magnitudes found in the SMOKE system. In that system:

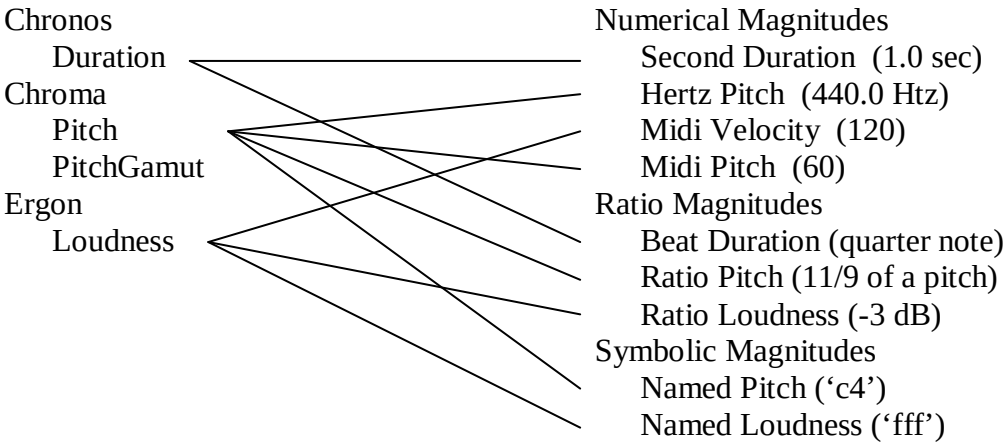
> "The class of a music magnitude depends on the 'type' of its value (e.g. a floating point number or a string), while its species will denote what it represents. Only the species will be visible to the user."[54]

Pope also states:

> "Some of the magnitudes depend on what they stand for (the representation class, and some of it on how they are stored (the implementation class). These two aspects are the objet's species and their class."[55]

My list of magnitudes would be similar to the list provide by Pope in his article "Object-oriented Music Representation"

Table 16 SMOKE Music Magnitudes[56]



The column on the left of Table 16 is the representation of the magnitude and the right column is the species of the magnitude. The lines between the two columns show how

---

University, Columbus, Ohio; November 1989, (San Francisco, CA: International Computer Music Association, 1989), 324-327.

[53] Pope, "Object-oriented," 60.

[54] Ibid., 61.

the two different representation methods come together to form the different magnitudes. Note that one representation can be shown in several different species.

In order for these magnitudes to be useful, the tempo markings and time signatures of the score would need to be known. This new set of objects would be stored in Score and contain information about the tempo and time signature changes within the piece. With this set of objects in place, the duration magnitudes could return a value in seconds for an event that had a duration of a quarter note.

I feel that the SCuD framework has achieved the three goals that it set out to accomplish. Through careful choice of programming language and design of the framework, especially the fact the framework deals only with scores, SCuD can easily be reused on different operating systems and by different users. Because of the design of the objects and the creation of a single inheritance tree all pointing to BasicEvent, the system can be extended with knowledge of only a few objects. Also, by keeping the design and the objects simple and providing documentation and examples for each object, the user will find that framework is easily used to create composition systems of their own.

[55] Ibid., 60-61.
[56] Ibid., 61.

Bibliography

Agresti, William W. and Frank E. McGarry. "The Minnowbrook Workshop on Software Reuse: A Summer Report." In *Tutorial: Software Reuse: Emerging Technology*. Edited by Will Tracz. Washington D.C.: Computer Society Press, 1988.

Biggerstaff, T. and C. Richter. "Reusability Framework Assessment and Directions." In *Tutorial: Software Reuse: Emerging Technology*. Edited by Will Tracz. Washington D.C.: Computer Society Press, 1988.

Burbeck, Steve. "Applications Programming in Smalltalk 80™: How to Use Model-View-Controller (MVC)."

Cook, Perry R. *Synthesis Toolkit in C++ Version 1.0: May 1996, SIGGRAPH 1996*. http://ccrma-www.stanford.edu/software/stk/Papers/stksiggraph96.pdf. Accessed June 2003.

DMIX. http://www.research.ibm.com/mathsci/cmc/dmix.htm. Accessed July 2003.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1995.

Hall, Patrick A. V. "Software Components and Reuse – Getting More Out of Your Code." In *Tutorial: Software Reuse: Emerging Technology*. Edited by Will Tracz. Washington D.C.: Computer Society Press, 1988.

Hamman, Michael and Simon Pamment. *A C++ Framework for Generic Programming and Composition*. http://www.shout.net/~mhamman/papers/cim2000_c++_g.htm. Accessed June 2003.

Hanappe, Peter. "Design and Implementation of an Integrated Environment for Music Composition and Synthesis." Ph.D. diss. University of Paris. 1999.

Kaper, Hans G., Sever Tipei, and Jeff M. Wright. "DISCO: an Object-oriented System for Music Composition and Sound Design." in *Proceedings of the 2000 International Computer Music Association in Berlin, Germany August 2000*. 340-343. San Francisco, CA: International Computer Music Association. 2000.

Krasner, G. and S. T. Pope. "A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming*. 1, no. 3 (1988).

Monroe, R. T. and D. Garlan. "Style-Based Reuse for Software Architectures." In *Fourth International Conference on Software Reuse held in Orlando, Florida 23-26 April 1996.* Edited by Murali Sitaraman, 84-93. Los Alamitos, CA: IEEE Computer Society Press, 1996.

Oppenheim, Daniel Vincent. *DMIX: An Environment for Composition*. http://www.research.ibm.com/mathsci/cmc/papers/dmix89.pdf. Accessed July 2003.

_____. *The P-G-G Environment for Music Composition – A Proposal*. http://www.research/ibm.com/mathsci/cmc/papers/pgg87.pdf. Accessed June 2003.

Pope, Stephen Travis. *The Musical Object Development Environment: MODE (Ten Years of Music Software in Smalltalk).* http://www.create.ucsb.edu/~stp/publs.html#MODE. Accessed February 2002.

_____. "Object-oriented Music Representation." *Organized Sound*. 1, no. 1 (1996).

_____. *The SIREN Music / Sound Package for Squeak Smalltalk.* http://www.create.ucsb.edu/~stp/publs.html#MODE. Accessed February 2002.

Smaragdakis, Yannis and Don Batory. "Implementing Reusable Object-Oriented Components." In *Fifth International Conference on Software Reuse held in Victoria, BC, Canada 2-5 June 1998.* Edited by Premkumar Deuanbu and Jeffry Poulin, 36-45. Los Alamitos, CA: IEEE Computer Society Press, 1998.

Sommerville, Ian. *Software Engineering*. Fifth Edition, Reading MA: Addison-Wesley Publishing Company, Inc., 1995.

Sorensen, Andrew and Andrew Brown. *jMusic: Music Composition in Java. An Introduction to jMusic*. http://jmusic.ci.qut.edu.au/jmtutorial/t2.html. Accessed June 2003.

Tipei, Sever. "Manifold Compositions - a (Super)computer-assisted Composition Experiment in Progress" In *Proceedings of the 1989 International Computer Music Conference held at the Ohio State University, Columbus, Ohio, November 1989*. 324-327. San Francisco, CA: International Computer Music Association, 1989.

The Varèse Environment. http://www.ircam.fr/equipes/repmus/Varese/index.html. Accessed July 2003.

Walker William F. *A Computer Participant in Musical Improvisation*. http://www.shout.net/~walker/papers/walker-chi97.pdf. Accessed 2003

_____. "A Conversation-based Framework for Musical Improvisation." Ph.D. diss., University of Illinois, 1994.

Walker, William, Kurt Hebel, Salvatore Martirano, and Carla Scarletti. *ImprovisationBuilder: Improvisation as Conversation*. http://www.shout.net/~walker/papers/walker-icmc92.pdf. Accessed June 2003.

Williams, Tony. "Reusable Components for Evolving Systems." In *Fifth International Conference on Software Reuse held in Victoria, BC, Canada 2-5 June 1998*. Edited by Premkumar Deuanbu and Jeffry Poulin, 12-16. Los Alamitos, CA: IEEE Computer Society Press, 1998.

http://www.pbs.org/wgbh/pages/frontline/gulf/maps/