UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Compositional and analytic applications of automated music notation via object-oriented programming

Permalink
https://escholarship.org/uc/item/3kk9b4rv

Author
Trevino, Jeffrey Robert

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Compositional and Analytic Applications of Automated Music Notation via Object-oriented Programming

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Music

by

Jeffrey Robert Trevino

Committee in charge:

Professor Rand Steiger, Chair
Professor Amy Alexander
Professor Charles Curtis
Professor Sheldon Nodelman
Professor Miller Puckette

2013

Copyright
Jeffrey Robert Trevino, 2013
All rights reserved.

The dissertation of Jeffrey Robert Trevino is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California, San Diego

2013

DEDICATION

To Mom and Dad.

EPIGRAPH

Extraordinary aesthetic success
based on extraordinary technology
is a cruel deceit.
—-Iannis Xenakis

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

Acknowledgements . . . xii

Vita . . . xiii

Abstract of the Dissertation . . . xiv

Chapter 1 A Contextualized History of Object-oriented Musical Notation . 1
- 1.1 What is Object-oriented Programming (OOP)? . . . 1
  - 1.1.1 Elements of OOP . . . 1
  - 1.1.2 A Nosebleed History of OOP . . . 6
- 1.2 Object-oriented Notation for Composers . . . 12
  - 1.2.1 Composition as Notation . . . 12
  - 1.2.2 Generative Task as an Analytic Framework . . . 13
  - 1.2.3 Computational Models of Music/Composition . . . 14
  - 1.2.4 Computational Models of Notation . . . 16
  - 1.2.5 Object-oriented Systems . . . 17
  - 1.2.6 Graphical Object-oriented Programming Systems . 19
- 1.3 Design Values for Automated Notation Systems, Illustrated with the Abjad API for Formalized Score Control . . 23
  - 1.3.1 The Abjad API for Formalized Score Control . . . 23
  - 1.3.2 Design Recommendations . . . 27

Chapter 2 Computational Modeling as Analysis . . . 38
- 2.1 The Conflation of Analysis and Composition Reveals and Posits Construction . . . 38
  - 2.1.1 Formalization Reveals Metaphor . . . 38
- 2.2 Reverse Engineering as Analysis: Two Case Studies in Formalized Score Control as Analysis . . . 40
  - 2.2.1 *Cantus in Memory of Benjamin Britten* (1977-1980) by Arvo Pärt . . . 40
  - 2.2.2 *Windungen* (1976) by Iannis Xenakis . . . 57

2.3 Revealed Strengths and Weaknesses of Formalized Score Control 81

Chapter 3 Automated Notation for the Analysis of Recorded Music 84
3.1 Background 84
3.2 Methodology for Representing Amplitude and Onset Time as Notation 85
3.3 Conclusion and Future Directions for Research 98

Chapter 4 Compositional Applications 100
4.1 Algorithmic Tendencies, 2004—2008 101
4.1.1 *Substitute Judgment* (2004) for Solo Percussionist . . 102
4.1.2 *Binary Experiment for James Tenney* (2005) for Four Contrabasses 104
4.1.3 *Mobile* (2005) for Tenor Saxophone 106
4.1.4 *Zoetropes* (2005—6) for Bass Clarinet, Cello, and Percussion 108
4.1.5 *Unit for Convenience and Better Living 003* (2006) for Solo Bass Clarinet 110
4.1.6 *Mexican Apple Soda (Consumer Affect Simulation I.1)* (2006) for Contrabass and Chamber Ensemble . . . 111
4.1.7 *Mexican Apple Soda Paraphrase* (2007) for Contrabass and Live Electronics 112
4.1.8 *Perfection Factory* (2008) for Two Percussonists . . . 112
4.2 Installation and Visual Music, 2009—2010 114
4.2.1 *Algorithmically Generated Trees* (2009) 114
4.2.2 *Blooms* (2010) 116
4.3 Computer-assisted Works, 2010—2013 119
4.3.1 *Being Pollen* (2010—2011) for Solo Percussion 119
4.3.2 +/- (2011—2012) for Twenty French Horns 121
4.3.3 *The World All Around* (2013) for Harp, Clarinet, and Piano 124
4.4 Conclusion 128

Appendix A Code Examples 131
A.1 Abjad Interface to Mike Solomon's LilyPond Woodwind Diagrams As a Function, Implemented with Basic String Functions 132
A.2 Abjad Interface to Mike Solomon's LilyPond Woodwind Diagrams As a Function, Implemented with Abjad Scheme Functions 133

A.3 Abjad Interface to Mike Solomon's LilyPond Woodwind Diagrams As the WoodwindDiagram Class, Inheriting from Abjad's AbjadObject Abstract Class 134
A.4 Processing Code for *Algorithmically Generated Trees* 143
A.5 Catalogue of Possible Entrances Into and Exits from Clarinet Multiphonics 155
A.6 Clarinet Solo Material Based on Multiphonic Catalogue . . 160
A.7 Prepared Piano Part for *The World All Around* 165
A.8 Harp Part for *The World All Around* 171
A.9 Formatted Score for *The World All Around* 174

Appendix B Score Examples 176
B.1 Arvo Pärt's *Cantus in Memory of Benjamin Britten* (1977—80) for Bell and String Orchestra, as Rendered with the Abjad API for Formalized Score Control 177
B.2 Iannis Xenakis's *Windugen* (1976) for Twelve Cellos, as Rendered with the Abjad API for Formalized Score Control 183
B.3 Glenn Gould's Performances of the First Movement of Webern's op. 27 Piano Variations 191
B.4 *The World All Around* (2013) for Prepared Piano, Eb Clarinet, and Harp 196

LIST OF FIGURES

Figure 1.1: Abjad creates notation by scripting the LilyPond typesetting program. . . . 24
Figure 1.2: Rest-delimited notes and chords. . . . 28
Figure 1.3: Code to slur groups of rest-delimited notes and chords in PWGL/ENP. . . . 28
Figure 1.4: Slurred groups of rest-delimited notes and chords. . . . 29
Figure 1.5: A multiphonic notation, including a woodwind diagram. . . . 33
Figure 1.6: Graphic overrides change the appearance of a woodwind diagram. 35

Figure 2.1: A font function enables custom typefaces. . . . 41
Figure 2.2: Modeling the bell and string staffs and their names. . . . 42
Figure 2.3: Adding string staffs to a score. . . . 43
Figure 2.4: Modeling the bell part. . . . 44
Figure 2.5: Modeling pitch as a series of scalar descents. . . . 45
Figure 2.6: Modeling the pitches: a switch system for choosing arpeggio notes. . . . 46
Figure 2.7: Applying the arpeggio notes to the scalar descents. . . . 47
Figure 2.8: Recursively generating most of the string parts' rhythms. . . . 48
Figure 2.9: Splitting durations cyclically by the duration of one bar. . . . 48
Figure 2.10: Manual composition of pitches and rhythms after generation. . . 50
Figure 2.11: Splitting and finishing the viola part. . . . 51
Figure 2.12: The cello and contrabass pitches and rhythms composed to completion. . . . 52
Figure 2.13: Placing previously generated pitches and rhythms into measures. 53
Figure 2.14: Adding dynamic markings to parts via measure indexes. . . . 54
Figure 2.15: Adding technical and expressive markings to parts via measure indexes. . . . 55
Figure 2.16: Defining and using a custom technical marking. . . . 56
Figure 2.17: Adding rehearsal marks. . . . 56
Figure 2.18: Document layout and formatting. . . . 57
Figure 2.19: Modeling the rotation of material through the score. . . . 59
Figure 2.20: A function that returns a matrix of cyclic tuples to specify which staffs the rotating music should be written on. . . . 60
Figure 2.21: Modeling Xenakis's bookended rotations. . . . 60
Figure 2.22: From single staff to rotating staffs. . . . 61
Figure 2.23: Repitching the staff as it rotates. . . . 62
Figure 2.24: Modeling the low-level typographical habits in the rotation section. . . . 63
Figure 2.25: Function for creating a staff of random pitches. . . . 64
Figure 2.26: The final stage of typographical adjustment for the rotation section. . . . 65

Figure 2.27: The final rotation function. 66
Figure 2.28: Utility functions enable rotation. 66
Figure 2.29: Modeling the score's first rotation with the rotation function. . . 67
Figure 2.30: Modeling a rotation with multiple simultaneous pitches. 68
Figure 2.31: Modeling the tutti section with randomly selected sixteenth notes. 69
Figure 2.32: The tutti section as single function. 70
Figure 2.33: The first section of the score as a single encapsulation. 70
Figure 2.34: Weighted probability choice functions for the first random walk section. 71
Figure 2.35: Conditional checks to determine a clef change. 72
Figure 2.36: Applying automatic clef changes to an expression. 73
Figure 2.37: Adding random walk notes to the score. 74
Figure 2.38: Querying and fusing trill spanners. 75
Figure 2.39: Functions for adding drones and random walks. 76
Figure 2.40: Functions for adding the random walk gesture to score, framed by drones as specified. 77
Figure 2.41: The random walk section as a single function. 78
Figure 2.42: Imposing metric hierarchy by fusing chains of small durations. . 79
Figure 2.43: Imposing metric hierarchy on the entire score. 80
Figure 2.44: Creating the score object. 81

Figure 3.1: Reading recording data from file in Python. 87
Figure 3.2: Processing recording data in Python. 88
Figure 3.3: Reading pitch data from the score file. 89
Figure 3.4: Converting the score to Abjad's pitch representation. 90
Figure 3.5: The final file parsing function. 90
Figure 3.6: Coloring noteheads according to amplitude. 91
Figure 3.7: Quantizing performance events with Abjad. 92
Figure 3.8: Pitching each performance according to the score data. 93
Figure 3.9: Typographical manipulations to split one staff to a piano staff. . 94
Figure 3.10: Splitting a performance to a piano staff. 95
Figure 3.11: Final encapsulations, including format and layout of the Score object. 96
Figure 3.12: The final automatic notation function. 97

Figure 4.1: Relative Durations of Materials in *Substitute Judgment* (2004). . . 102
Figure 4.2: Division and Formal Disposition of Materials in *Substitute Judgment* (2004). 103
Figure 4.3: Section A of *Binary Experiment for James Tenney* (2005) for four contrabasses. 105
Figure 4.4: Section B of *Binary Experiment for James Tenney* (2005) for four contrabasses. 106

Figure 4.5: Section A of *Mobile* (2005) for tenor saxophone. 107
Figure 4.6: Section B of *Mobile* (2005) for tenor saxophone. 108
Figure 4.7: Bass clarinet solo from *Zoetropes* (2005—6). 109
Figure 4.8: Materials in *Unit for Convenience and Better Living 003* (2006). . . 110
Figure 4.9: Material E has almost entirely crowded out the other materials in *Unit for Convenience and Better Living 003* (2006). 111
Figure 4.10: Memory notation navigates between listened selection and notated pitch in *Perfection Factory* (2008) for two percussionists. . . 113
Figure 4.11: Colored noteheads indicate selected pitches in *Perfection Factory* (2008) for two percussionists. 113
Figure 4.12: Trees generated and e-mailed to the audience in *Algorithmically Geneated Trees* (2009). 115
Figure 4.13: Stills captured from the rotating motion of *Bloom I* (2010). 117
Figure 4.14: Stills captured from the rotating motion of *Bloom II* (2010). 118
Figure 4.15: Half-cosine interpolations transition from complex rhythms to pulse in *Being Pollen* (2011). 120
Figure 4.16: The second recitation pairs common notation with an instruction score *Being Pollen* (2011). 121
Figure 4.17: Screenshot from the animated notation parts created for +/- (2011—2012). Mouthpiece pops are indicated by points that scroll from right to left along a midline, to be performed when they cross the vertical line at the left boundary. The minimal aesthetic of the interface is inspired by early video games, such as Pong (1972). 123

ACKNOWLEDGEMENTS

The dissertation you're reading wouldn't exist without the invaluable advice of my mentors, the support of my friends and family, and the patience of my partner, Melissa. This dissertation was also supported by the generosity of the Andrew W. Mellon Foundation and the Woodrow Wilson Foundation.

VITA

2005	B. A. in Music, Science, and Technology w/distinction, Stanford University
2005-2008	Graduate Teaching Assistant, University of California, San Diego
2007	M. A. in Music, University of California, San Diego
2009	Guest Student under Walter Zimmermann, University of the Arts, Berlin, Germany
2010	Guest Lecturer, Stanford University, Berlin Campus
2010-2013	Graduate Teaching Assistant, University of California, San Diego
2013	Ph. D. in Music, University of California, San Diego

PUBLICATIONS

Trevino, J. and Andrew Allen. "Richard Boulanger and Victor Lazzarini, Eds. : The Audio Programming Book" in *Computer Music Journal*, Summer 2012, Vol. 36, No. 2, Pages 85-89.

Gurevich, M. and Jeffrey Robert Trevino. "Expression and Its Discontents: Toward an Ecology of Musical Creation." *Proceedings of the 2007 International Conference on New Interfaces for Musical Expression*, June, 2007, New York.

Trevino, J. "Jeffrey Trevino Interviews Chris Chafe" *Array (The Journal of the International Computer Music Association)*, Winter, 2006, pp. 22-31.

Caceres, J.P., Mysore, G., and Trevino, J. "SCUBA: A Self-Contained, Unified Bass Augmenter." *Proceedings of the 2005 Conference on New Interfaces for Musical Expression*, June, 2005, Vancouver, pp. 38-41.

ABSTRACT OF THE DISSERTATION

Compositional and Analytic Applications of Automated Music Notation via Object-oriented Programming

by

Jeffrey Robert Trevino

Doctor of Philosophy in Music

University of California, San Diego, 2013

Professor Rand Steiger, Chair

The current research applies the Abjad API for Formalized Score Control (Bača, Oberholtzer, and Adán, 1997—present) in compositional and analytic contexts to demonstrate specific recommendations for the design of automated notation systems, expand the system's application into computational musicology, and formalize the algorithmic tendencies of the author's compositional praxis. First, a concise review of the literature summarizes the history of object-oriented programming, proposes a framework for understanding automated notation systems, and makes recommendations for the design of object-oriented programming systems for composers. Next, two step-by-step literature examples recreate the typographical details of previously composed scores as interpreter sessions; this yields a histori-

cally informed assessment of the API's abilities and drawbacks, contributes to a body of pedagogical examples for new users, and implicitly offers an analysis of the modeled works. In a third chapter, the API is used to quantize and notate data extracted from several recorded performances of a single musical work, illustrating the ways in which traditional musical notation can be extended to visualize multidimensional data for computational musicology. Lastly, to demonstrate the efficacy of the API in the context of an individual compositional practice, the fourth and final chapter discusses the author's uses of the system as the continuation of an extant algorithmic composition practice.

Chapter 1

A Contextualized History of Object-oriented Musical Notation

1.1 What is Object-oriented Programming (OOP)?

1.1.1 Elements of OOP

Object-oriented programming (OOP) may be understood as an alternative to a previously conventional segregation of *data* — values expressed with numbers — and *functions* — procedures that process data. This older model of *procedural programming* emphasizes the way in which a program accomplishes a task, by sending data through a pipeline of processing functions, much as a recipe describes separately the ingredients and procedures necessary to create a certain dish. Procedural programming approaches a problem by first asking: what must be done, and how might these tasks be analyzed into smaller tasks sufficiently specific for the limitations of the utilized programming language? OOP, on the other hand, emphasizes the agents that take part in the process. It approaches a problem by first asking: what are the actors, agents, and objects involved in this task, and how to they communicate with each other and behave (Wirfs-Brock, Wilkerson, and Wiener 1990, p. 5)?

An Example: Implementing a Counter with Procedural and Object-oriented Programming

In object-oriented programming, data values become object *states* and the functions that process them become object *behaviors*. This means that values and procedures commonly used together have been grouped together into an object. The following example illustrates this fundamental difference between procedural and object-oriented programming.

The problem of managing of a *counter* — a value to be incremented or decremented to track some aspect of a system — arises often in the course of music programming: for example, to track the number of measures created in a program designed to make a specified number of measures, a program might repeatedly perform a measure-creating operation and increment a counter to indicate that a measure has been created, until the counter reaches the specified number of measures. To implement such a counter with procedural programming, the programmer might initialize a variable named "counter" to hold a value of "0,"

```
int counter = 0
```

perform the action to be counted, and then pass the value through an addition function to increment the counter:

```
counter += 1
```

In this way, the program records the number of measures created in the data value "counter." In this example, data and procedure exist separately in the system: the counter's value exists as a variable, named "counter," and the incrementing procedure exists as a separate function, represented by the symbols "+=".
In contrast to this procedural approach, for an object-oriented counter implementation, the programmer creates a counter object, with a value that records the object's state ("counter") and a behavior that increments this value ("inc"):

```
object Counter:
value counter

behavior inc() : counter = counter + 1
```

To interact with this object, the programmer may query the value of the counter:

```
Counter.value
```

And the programmer may increment the counter:

```
Counter.inc()
```

Data Abstraction and Encapsulation

Humans often abstract ideas and perceptions to emphasize meaningful information and suppress irrelevant detail; for example, a map necessarily contains less detail than the navigated landscape to which it corresponds. (Wirfs-Brock, Wilkerson, and Wiener 1990, p. 3). Likewise, the true detail of a natural environment does not vanish when one uses a map to navigate, and a map can be viewed as a usefully simplified model that affords purposeful interaction with a relatively more complex environment. When interaction with this simplification governs interaction with the environment itself, the map has become an *interface* to the landscape.

The creation of multiple objects with separate internal memories and behaviors in OOP creates an environment in which one object might duplicate the behavior of another object or lack a behavior that characterizes another object; for the object concept to remain meaningful in an environment of multiple, differing objects, one object's memory and behaviors must be accessible only through interaction with that object (Cardelli, Wegner, et al. 1985, p. 481). The limitation that an object's internal memory and behaviors may be accessed only via an interface to that object is called *encapsulation* (Van Roy and Haridi 2004, p. 18), and the broader methodology of segregating the construction of an object from its use is known as *data abstraction* (Abelson and Sussman 1983). Because an object's internal construction may only be accessed via the object's interface, the internal construction of an object's behaviors — the object's *implementation* — can change drastically, and, if each implementation maintains the same object interface, many implementations will behave identically; this trait is known in the literature as *polymorphism* (Van Roy and Haridi 2004, p. 18). (This object-oriented definition of polymorphism should not be confused with the literature's use of the term to classify the flexibility of a language's data type handling, as in Cardelli, Wegner, et al. 1985, pp. 472-480.) For example, the counter object above could be described as above, or in this alternate implementation:

```
object Counter:
value counter

behavior inc() : counter += 1
```

The only difference between these two `Counter` implementations is the specific formulation of the `inc()` method: in the first implementation, the addition function, in conjunction with an assignment operator, increments the counter's value, while in the second implementation, a single operator, `+=`, both increments and assigns the value. This difference remains invisible to the user: regardless of the specific implementation of the counter's methods, the interface to the object remains the same, and the programmer may increment the counter by invoking the increment method (`Counter.inc()`).

Problems of Interfaces: Affordances and Transparency

An object's interface allows and prevents certain modes of interaction with the object's internal state and behavior. Designer Don Norman proposes the idea of *affordances* to describe the way that an interface's design invites or discourages a certain mode of use (Norman 2003): boys pull on girls' pigtails, because the shape and height of pigtails affords (invites) pulling. Any technology will afford certain interactions and, consequently, applications of the technology.

The proliferation of interfaces through data abstraction also presents a tradeoff between usability and openness. As Nance and Sargent point out:

> A major consequence of the conjunction of HCI [(Human-Computer Interaction)] with other advances is an ever-increasing user relief from the requirement to have detailed knowledge of the underlying computing technology. The result has greatly expanded the population of productive users of the ubiquitous digital technology. However, a concomitant result is that, unless the user forces revealing actions, the modeling software hides how the function is performed (Nance and Sargent 2002, p. 164).

In the trade-off between technological transparency, on the one hand, and straightforwardness of use, on the other, object-oriented programming trades access to object internals for an interface's affordances.

Incremental Data Abstraction via Classes and Inheritance

A set of data abstractions might address most of the necessary tasks of a given application area, but the programmer will likely need to create new abstractions to

meet the challenges of new problems or propose novel solutions to established problems; because a programming language might be used to solve novel problems, languages should simplify the process of creating new abstractions (Liskov and Zilles 1974). For example, if two objects are similar, it would be useful to create one object with reference to the other, by describing only the difference between the two. OOP enables this with its system of classes and inheritance.

OOP departs fundamentally from other paradigms by abstracting the idea of a data type. (Van Roy and Haridi 2004, p. 18). OOP creates objects from templates called *classes*. A class describes a kind of object and includes *attributes*, *methods*, and *properties*. When an object is created from a class, it is an *instance* of the class and *inherits* all of the attributes, methods, and properties of the class; *inheritance* defines new abstractions as incremental extensions of existing abstractions and allows the user to create a new object by describing only the difference between the old and new objects. To say that some new class (known variously as a *derived class*, *child class*, or *subclass*) *inherits* from an existing class is to say that the new class contains its own hidden, encapsulated version of the methods, attributes, and properties of the class from which it derives (known variously as the *base class*, *superclass*, or *parent class*). Class attributes might be initialized uniformly, through the specification of a default value in the class definition itself, or individually, when the specific object is *instantiated* from the class. For example, to instantiate an object from the Note class in the Abjad API for Formalized Score Control (Bača, Oberholtzer, and Adán, 1997-present), the programmer must supply a pitch and duration attribute to the `Note()` function, which instantiates an object from the Note class:

```
>>> a_note_object = Note( ”c’”, (1,4) )
```

Named a_note_object, this object inherits all of the attributes, methods, and properties from its instantiating class:

```
note.descendants
note.duration_multiplier
note.leaf_index
note.lilypond_format
note.lineage
note.multiplied_duration
note.note_head
note.override
```

```
note.parent
note.parentage
note.preduration
note.duration
note.prolation
note.set
note.sounding_pitch
note.spanners
note.start_offset
note.stop_offset
note.storage_format
note.timespan
note.written_duration
note.written_pitch
note.written_pitch_indication_is_at_sounding_pitch

note.written_pitch_indication_is_nonsemantic
```

Every note object instantiated from the Note class inherits the same set of attributes from its instantiating class. The arguments given to the `Note()` function have supplied values for several of these attributes; for example, this specific object has a written pitch equal to middle C and a duration of one quarter note; this can be seen by querying the object's `written_pitch` and `duration` attributes:

```
>>> a_note_object.written_pitch
NamedChromaticPitch("c'")
>>> a_note_object.duration
Duration(1, 4)
```

1.1.2 A Nosebleed History of OOP

Cybernetics

During and in the decade following World War II, scientists formulated mathematical models of communication, cognition, homeostasis, and biological systems (Aspray 1985). Shortly after the end of the war, *cybernetics* — a term coined by Norbert Wiener in 1948 — emerged as an academic field organized around the study of command and control dynamics, the design and analysis of systems, and analogies between organisms and machines, including computers (Dunbar-Hester 2009). While conducting anti-aircraft weaponry research at MIT's Radiation Laboratory in

1943, Norbert Wiener, Julian Bigelow, and Arturo Rosenblueth bifurcated the analysis of human-machine systems into two paradigms: a "behavioristic" model that emphasizes the relationship between a system's inputs and outputs and a "functional" understanding that emphasizes an understanding of the internal structure and function of objects (Priestley 2011). While the authors were concerned at the time with illustrating how this "behavioristic" understanding of systems allows the uniform analysis of human-machine systems, thus enabling the unified discussion of systems with both human and machine actors and laying the groundwork for cybernetics, this dichotomy between an input-output systems model and an object-based systems model presages the computational models that underlie procedural and object-oriented programming languages respectively. The presence of this dichotomy in the founding work of cybernetics does not demonstrate a clear line of influence between object-oriented programming and cybernetics; rather, it shows that practitioners have acknowledged since the beginning of computation research that a view of programming abstractions as either participants in a processing chain or as communicating objects with encapsulated construction and behavior can lead to divergent views of systems and problems.

Simulation

The history of OOP languages begins with the simulation programming languages of the 1960s, of which Simula67 (1967), based heavily on Algol60 (1960), was the first (Van Roy and Haridi 2004, p. 489). Simulation languages seek to model the behavior of complex systems in order to enhance system performance (*system analysis*), evaluate the performance of systems (*acquisition and system acceptance*), and create artificial environments for research and entertainment (Nance and Sargent 2002, p. 162). Simulation predates computers and began as early as 1777, when Buffon estimated the value of π by dropping a needle onto strips of wood — not far afield from this initial experiment, the first computer simulations were "Monte Carlo" models, a technique for modeling complex systems that uses deterministic inputs to constrain and measure a random distribution: i.e., to estimate the value of π, circumscribe a circle inside a square, place points randomly and uniformly within the square, and

then measure the ratio of the number of points inside the circle to the total number of points to derive the value of π (Nance and Sargent 2002, p. 162). From the perspective of user interaction, a simulation language allows the user to describe the elements of a system, their attributes, their permissible logical relationships, and the time-dependent processes that govern the behavior of the system (Kiviat 1993). Simulation languages contribute several key concepts to OOP, as well as to computer science more broadly: Simula (1965) proposed that a section of code represent a "quasi-independent" process; SIMSCRIPT II (1968) introduced an *entity/attribute/set* concept, by which entities could be both members of sets and unique objects with their own attributes; and Simula67 extended this model by introducing the key concepts of data types, inheritance, encapsulation, and message passing between entities (Nance and Sargent 2002, p. 167).

Structured Programming

In the 1960s, *structured programming*, in which sequentially specified and grouped operations describe the order in which the program processes data values, became an industry best practice, and professionals warned against programming habits that decoupled the sequence of execution from the order in which procedures have been specified; as Edsger W. Dijkstra recommends in his famous letter to the editor, "Goto Statement Considered Harmful":

> ...[W]e should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible (Dijkstra 1968).

Dijkstra continues to decry the use of the "goto" statement, a programming device that allows programmers to leap to specified line of code in the written program, thus specifying a sequence of code execution that differs substantially from the written, visual order of commands and makes the analysis and evaluation of programs more difficult. Beyond the specific goal of revising contemporary programming habits, this call for the elimination of a "gap" between the static nature of code and the dynamic nature of the data processing it enables sets the conceptual

stage for an object-oriented approach that emphasizes the cooperation of variously static or dynamic objects that interact with one another, while enabling programmers to conceptualize a task as a sequence of processing functions that act upon data values.

Other Influences

Other research areas and trends contributed to the formation of object-oriented paradigms. Knowledge representation languages (such as KRL, KEE, FRL, and UNITS) for Artificial Intelligence engaged discrete state simulations with a knowledge theory based on Minsky's concept of "frames," while ACTORS and FLAVORS (both 1981) developed message passing and multiple inheritance respectively (Stefik and D. Bobrow 1985). Several object systems were added to the LISP language widely used in AI between the late 70s and late 80s (D. G. Bobrow et al. 1986), and the self-defining organization of LISP inspired the definition of object-generating classes as objects themselves in Smalltalk during the 1970s (Kay 1996, p. 575). As more efficient time-sharing mainframes defined the metaphors of personal computing in the 1960s, computers modeled users as interacting agents with states and behaviors, and *recursive design* allowed operating systems to model themselves, creating a number of *virtual machines* that each encapsulated the computation abilities of the mainframe computer itself (Creasy 1981). Metaphors of time sharing and the "master"/"instance" data model of Ivan Sutherland's pioneering drawing program and interface SKETCHPAD (Sutherland 1964) both influenced the development of Smalltalk in the 1970s (Kay 1996, p. 575) — although no more than speculation, Smalltalk creator Alan Kay was a professional jazz guitarist before entering college, and musical abstractions such as scales and chords may also have influenced the development of OOP (Kay 1996, p. 579).

Smalltalk

Smalltalk (1980) is the first widely used object-oriented programming language (Sammet 1991). Created through research directed by Alan Kay at Xerox during the 1970s, many syntactically divergent versions of the language throughout the

decade adhered to the same core principles of recursive design:

1. Everything is an *object*.
2. Objects communicate by sending and receiving *messages*.
3. Objects have their own memory.
4. Every object is an instance of a *class*, which is itself an object.
5. The class holds the shared *behavior* for its instances.
6. Classes are organized into an *inheritance hierarchy*.

Evoking Dijkstra's "conceptual gap," the creators of Smalltalk introduced their language to the public as the result of design concerned explicitly with elegant discourse between human and computational models of concepts:

> We have chosen to concentrate on two principle areas of research: a language of description (programming language) that serves as an interface between the models in the human mind and those in computing hardware, and a language of interaction (user interface) that matches the human communication system to that of the computer (D. Ingalls 1981).

To align the conceptual framework of intercommunicating objects with the syntax of their new language, they created an "object message" syntax to emphasize that Smalltalk's code directs the flow of communication from object to object. In Smalltalk, if `bob` is an integer, the programmer sends an addition message to `bob` to change `bob`'s value:

```
bob +4
```

If the previous value of `bob` had been 3, the new value stored in the object would be 7; if `bob` were a string instead of an integer, with a value of `"Meta"`, the new value in `bob` might be `"Meta4"` (D. H. H. Ingalls 1978).

Hybrid Languages

While Smalltalk is a pure object-oriented language — everything is an object, including classes — several *hybrid* programming languages became popular in the

1990s and 2000s. Languages such as C++, Java, Python, and Perl enable OOP but also contain built-in data types, such as integers, lists, and floats (floating-point decimal numbers), that cannot be modified by the user (J. Schwarz 1993, Gosling 2000, Van Rossum and Drake 2003, Holzner 1999). These data values must be wrapped in an object instance via inclusion in a class or object attribute or method to participate in the language's class hierarchy; however, the use of such values without OOP enables the basic conventions of structured, procedural programming. Not all of these languages were created to be hybrid languages; Python, for example, began as a completely procedural system and gained object orientation during the course of its development.

Proposed Modern Standards of OOP

New applications of a programming language can result from changes of programming style or personal preference (Sammet 1991). As OOP has gained hegemony in the programming world, authors have proposed and widely circulated "mental toolkits": sets of standards and best practices for programming that maximize the advantages of abstraction, encapsulation, and inheritance without introducing problems at later stages in code development or revision. The practice of any of these models as convention promotes a strict understanding of programming style intended to limit the perils and maximize the benefits of OOP. For example, Robert C. Martin has proposed an mnemonic rule-set for class design, SOLID (*Principles of OOP*), and Craig Larman has proposed a similar guide to assigning responsibility to objects and classes, GRASP (Larman 2002). Such guidelines, should they become standard, can substantially influence both the way in which programmers use a given programming language and the way that programmers approach and design solutions to problems.

1.2 Object-oriented Notation for Composers

1.2.1 Composition as Notation

Western music's basis in notation implies two kinds of information analysis: graphic specification divides an instruction into perceptually fused but independently specified aspects (pitch, rhythm, dynamic), and segregates event information (notes on the page of the score) from sound production information (instructions for how to play an instrument to produce sound) (Ariza 2005a, p. 83). While many environments for both notation and sound production have arisen within the last twenty years, the present study concerns itself with systems' efficacy on two fronts: the ability to elegantly express both low- and high-level compositional ideas with the aid of a graphic or text-based programming language and the ability to generate a sufficiently detailed common practice musical notation from the programmed ideas, where sufficiency is assessed with reference to the accepted vocabulary of common practice notational symbols and constructs. Contra taxonomies of computer-aided algorithmic composition (CAAC) environments that have attempted to categorize a much broader set of systems (Ariza 2005b), the present discussion redefines composition narrowly as the act of programming a computer to create a notation in the form of a document — although recent practice has shown a healthy willingness to question the technological nature and collaborative context of this document.

Conventional aesthetic assumptions inhibit such a concretist definition of notation, and it is easier to adopt the view that composition is equivalent to notation from the standpoint of a specific aesthetic point of view, advanced in the middle of the twentieth century. While the practice of composition is still conventionally described as a process whereby composers mediate intentions or emotion through a representational technology toward a receiving participant (Davies 1994), mid-century American composers proposed an alternative approach to the same creative technologies, in which the act of composition exists entirely as the creation of a graphic artifact, despite the assumption that others will pursue sonic responses to the created artifact; as the American composer John Cage asked, "Composing's one thing, performing's another, listening's a third. What can they have to do with

one another?” (Cage 2011, p. 14). This question invites a multiplicity of possible relationships between composer, performer, and listener, and invites the formation of an artistic practice that reconsiders the interrelationship of these three musical roles from first principles. Likewise, Cornelius Cardew’s piece, *Treatise* (1963—67), invites the performer to invent correspondences between symbol and action, rather than specifying them via assumed performance practice or explanatory notes. It is in this spirit that the present study circumscribes composition as the creation of a graphic provocation to enacted response, rather than a manifestation of the conduit metaphor of human communication (Reddy 1979) or a transmitter/receiver model of information (Shannon 1949).

1.2.2 Generative Task as an Analytic Framework

Software production exists as an organizationally designed feedback loop between production values and implementation (Derniame, Kaba, and Wastell 1999), and it is possible to understand a system by understanding the purpose for which it was initially designed, the system’s *generative task(s)*. In the analysis of systems created for use by artists, this priority yields a dilemma instantly, as analyses that explain a system’s affordances with reference to intended purpose must contend with the creative use of technology by artists: a system’s intended uses might have little or nothing in common with the way in which the artist finally uses the technology. For this reason, the notion of generative task is best understood as an explanation for a system’s affordances, with the caveat that a user can nonetheless work against those affordances to use the system in novel ways. Generative tasks — informed by the cultural milieu of software development, economic constraints of software production, and the aesthetic proclivities of artists participating in development processes — constrain software features to enable a limited subset of possible representations and user interactions.

While composers working traditionally may allow intuition to substitute for formally defined principles, a computer demands the composer to think formally about music (Xenakis 1992). Keeping in mind generative task as an analytical framework, it is broadly useful to bifurcate an automated notation system’s development

into the modeling of music and composition, on the one hand, and the modeling of musical notation, on the other. All systems model both, to greater or lesser degrees, often engaging in the ambiguous or implicit modeling of music and composition while focusing more ostensibly on a model of notation, or focusing on the abstract modeling of music without a considered link to a model of notation. Due to the intimate link between notation and musical ideas, it is impossible for a system that models notation to avoid at least implicitly modeling musical and compositional ideas, and a computational model of music and composition is an inevitable component of every automated notation system, even when it exists as an unspoken set of technological constraints. Generative task explains a given system's balance between computational models of music / composition and notation by assuming a link between intended use and system development.

Automation, as the computational execution of a previously human-executed task, complicates a system's evaluation via generative task, because the tasks might be assumed to be executed by either human or computer. A compositional practice that positions notation as a central element of the creative process may claim that drawn notation by the human hand can never be eliminated from the process of composition (Hiller and Baker 1965, p. 131), just as well as it may claim that a computer must model in as detailed a manner as possible the typographical palette and choices of compositional thought through notation; that is, after considering generative task, human-computer interaction must be further considered to arrive at a concrete distribution of tasks between human and machine.

1.2.3 Computational Models of Music/Composition

Computational models of music might entail the representation of higher-level musical entities apparent in the acts of listening and analysis but absent in the symbols of notation themselves, as determined to be creatively exigent. Programming researchers and musical artists have modeled many such extrasymbolic musical entities, such as large-scale form and transition (Polansky, McKinney, and Studio 1991, Uno and Huebscher 1994, Dobrian 1995, Abrams et al. 1999, Yoo and Lee 2006), texture (Horenstein 2004), contrapuntal relationships (Boenn et al. 2009,

Acevedo 2005, Anders and Eduardo R. Miranda 2011, Balser and Streisberg 1990, Jones 2000, Uno and Huebscher 1994, Bell 1995, Farbood and Schoner 2001, D. Cope 2002, Laurson and Kuuskankare 2005, Polansky, Barnett, and Michael Winter 2011, Ebcioglu 1980), harmonic tension and resolution (Melo and Geraint Wiggins 2003, G. Wiggins 1999, Foster 1995), melody (Hornel 1993, M. Smith and Holland 1992), meter (Hamanaka, Hirata, and Tojo 2005), rhythm (Nauert 2007, Degazio 1996, Nick Collins 2003), timbre (Xenakis 1991, Creasey, Howard, and Tyrrell 1996, Osaka 2004), temperament (Seymour 2007, Gräf 2006), and ornamentation (Ariza 2003, Chico-Töpfer 1998). This work overlaps fruitfully with analysis tasks, and models of listening and cognition can enable novel methods of high-level musical structures and transformations, like dramatic direction, tension, and transition between sections (Nick Collins 2009, p. 108); the overlap of artistic application with analysis and simulation of cognitive models also causes a muddle of various motivations and methodologies, resulting in a field of research without clear evaluative criteria (Pearce, Meredith, and Geraint Wiggins 2002).

It is possible to model music computationally without recourse to the canonized abstractions above. Mid-century artists pioneered the conflation of signal processing and music composition, architecting systems that regarded compositionally both the spectral and symbolic characteristics of sound; in accordance with modernism's interest in coherence between multiple structural scales, temporal scales, and dimensions (Stockhausen and Barkin 1962), both Hebert Brün and Iannis Xenakis produced composition systems in which larger structural features emerged from mathematical constraints that generate sound files sample by sample (Luque 2009, Brün 1969). These systems demonstrate that the persistence of older theories of music is optional in computational models of music: in the works of James Tenney and Larry Polansky, for example, "mean event time" and probabilistically determined pitch selection algorithms replace traditional musical abstractions, such as tempo (Polansky 2010).

One subset of these extra-symbolic musical entities are those musical entities that overlap with concepts in notation. For example, a "chord" might be a vertically ordered collection of pitch classes in a harmonic conceit, or it might refer to the

specific arrangement of pitched noteheads, stemmed together into a composite notation symbol that instructs a performer to perform a sound that consists of several component pitches. Due to substantial overlap in vocabulary between musical and notational concepts, it can be difficult to separate a system's model of music/composition from its model of notation.

A system that affords a detailed model of music/composition without linking it to a sufficiently detailed model of musical notation does not afford automated notation — sufficiency, however, depends heavily on generative task. For example, if a composer requires an automated notation system to render complex rhythmic ideas that depend typographically on nested tuplets, a system that produces a notation only via a combination of MIDI and quantization must reduce rhythms to a non-hierarchical stream of event times, eliminating the temporally divisive approach of tuplet notation. For many rhythmic applications, though, MIDI suffices.

1.2.4 Computational Models of Notation

Many automated notation systems exist to model musical notation and the act of typographical layout without explicitly affording the computational modeling of music or composition (L. Smith 1972, Nienhuys and Nieuwenhuizen 2003, Hoos et al. 1998, Hamel 1997); many of these systems strongly imply a model of music, such as Grégoire for Gregorian chant, Django for guitar tablature, and GOODFEEL for Braille notation (Kuuskankare and Laurson 2006). In this light, feature-rich systems oriented toward classical composers, such as Finale, Sibelius, SCORE, Igor, Berlioz, and Nightingale fit into the mold of systems that model notation with genre as a primary determinant of generative task. Such a system might go so far as to enable a text-based object-oriented model of notation that automates some aspect of an otherwise point-and-click interface, as in the case of Sibelius's ManuScript scripting language (*Plugins for Sibelius*).

Many models of musical notation were created for purposes of corpus-based computational musicology. Formats such as DARM, SMDL, HumDrum, and MuseData model notation with the generative task of searching through a large amount of data (Selfridge-Field 1997). Commercial notation software developers attempted

to establish a data interchange standard for optical score recognition (NIFF) (Consortium and et al. 1995); since its release in 2004, MusicXML has become a valid interchange format for over 160 applications and maintains a relatively application-agnostic status, as it was designed with the generative task of acting as an interchange format between variously tasked systems (Good 2001).

Notation representations that underly many of these GUI-based systems often go undescribed as computer representations of notation, in favor of discussions about human-computer interaction. For example, Barker and Cantor developed an early model of music notation that underlies a four-oscilloscope GUI and describe their work entirely in terms of user interaction (Cantor 1971); likewise, discussions of modern commercial notation systems are primarily front-end oriented, without much awareness or criticism of the underlying computational models of notation.

1.2.5 Object-oriented Systems

The Crucial Development of Hierarchical Models

Many early models of musical notation were not hierarchical, and Lejaran Hiller, in reflecting on decades of automated notation work, has identified the lack of hierarchical organization as a limitation of early work — although Nick Collins points out that even Hiller's program PHRASE addresses the hierarchical organization of a score up to the level of a phrase, without moving beyond this mid-level of musical structure to concerns of large-scale form (Nick Collins 2009, p. 108). There were several object-oriented music environments by 1990 (Polansky 1990, p. 139), most created in or inspired by the newly popular Smalltalk-80 programming language; while they facilitated the hierarchical modeling of musical abstractions, they omitted or radically simplified the hierarchical nature of common notation. For example, Glen Krasner (Xerox Systems Science Laboratory) created Machine Tongues VIII, a music system that created an object-oriented model of the score/orchestra distinction inherited from Max Mathews' Music N languages, with a simple linear model of "partOn" and "partOff" command sequences (Krasner 1991), omitting hierarchical organization entirely when the system produces notational output; although

subsequent Machine Tongues systems introduced some hierarchical organization via "note" objects that inhabited "event lists," systems did not attempt to model the hierarchical detail of all a traditional score's elements. Like Hiller's PHRASE program, Andreas Mahling's CompAss system organized events hierarchically up to the mid-level "phrase" level of musical structure (Mahling 1991). These systems are perhaps best conceptualized as Smalltalk-based interfaces to the MIDI standard: as basic extensions of Smalltalk, they enabled the user to arbitrarily extend the system with new objects, creating a detailed and robust model of music, which was ultimately flattened into a list of noteOn and noteOff commands to be notated or played back via MIDI interface.

While a hierarchical model of notation and of musical events in time can exist in an entirely object-oriented paradigm, it is possible to observe even in these early systems the need for hybrid procedural/object-oriented approaches for the modeling of musical ideas: somewhat counterintuitively, some of the most important objects in these systems are varieties of transformation, to be enacted upon other objects — the most important nouns are verbs. HMSL — a system influenced heavily by James Tenney's work on temporal gestalt perception in music (Tenney and Polansky 1980), implemented throughout the 1980s in the Forth language, atop a custom object-oriented extension called ODE (Object Development Environment) — organizes objects hierarchically according to membership in "morphs," objects that represent morphological changes to be applied to raw data, such as parameterized event data (Polansky 1990, p. 139). Likewise, Stephen Travis Pope's MODE (Musical Object Development Environment) included "line segment" functions to be applied to event lists to transform the parameters of member objects (Pope 1991). The central role of transformational objects in these first object-oriented systems presages a later preference for hybrid procedural/object-oriented systems, in which built-in primitive data types — floating point numbers, strings for representing text, integers — allowed a variously procedural transformation of data or stateful representation of musical objects. (This tendency might be viewed as the computational persistence of signal generators from modular synthesizers, which allow signals to flexibly modulate other signals.) While procedural programming allows a transformational proce-

dure to be executed, object-oriented programming enables a transformation to exist as an parameterized object, with its own set of attributes.

By 1989, Glendon Diener's Nutation system (written in Objective C for the NeXT computer) had modeled both musical and notational structure hierarchically through the use of directed graphs (Diener 1991a, Diener 1991b, Diener 1989). While Diener mentions that users should be theoretically able to extend the system's hierarchical modeling to encompass alternative notation approaches and increasingly detailed models of common notation, the system does not include such a model of notation.

1.2.6 Graphical Object-oriented Programming Systems

Although realtime languages were available for music synthesis and control as early as 1981 (Mathews and Pasquale 1981, Mathews 1983), it took until the middle of the 1990s for realtime, graphical programming environments to become widely used (Puckette 1991, Puckette et al. 1996). While these systems specialize in either signal processing for synthesis applications or symbolic processing for automated notation applications, there are both extensions of Max/MSP and PD that enable musical notation (Didkovsky and Hajdu 2008, Kelly 2011) and extensions of OpenMusic and PWGL that enable synthesis and control of synthesized sound; notably, the specification of scores in tandem with control parameters for sound synthesis was a generative task in the creation of Pure Data.

IRCAM developed the Crime, CARLA, and Patchwork environments for composition in the second half of the 1980s, and PatchWork was the first object-oriented automated notation environment to catch the attention of established composers, including Brian Ferneyhough, Gérard Grisey, Magnus Lindberg, Tristan Murail, and Kaija Saariaho (Gérard Assayag et al. 1999); IRCAM developed PatchWork further into the OpenMusic environment, which gained, over the course of a decade of development, an interface to the control of synthesis parameters (Agon, Stroppa, and Gerard Assayag 2000), an interface to physical modeling (Polfreman 2002), analysis applications (Buteau and Vipperman 2009), an interface to feature data (Bresson and Agon 2010), and a collection of third-party libraries that extend

the basic "boxes" included in the environment distributed by IRCAM.

The naturalistic aesthetic agenda of spectralism played an important role as a generative task for these extensions. Composers' needs demanded the integration of signal processing and symbolic manipulation, and the SDIF standard, a sound file analysis interchange data representation standard developed jointly by CNMAT and IRCAM in the second half of the 1990s (M. Wright et al. 1999) had been incorporated into OpenMusic with SDIF-specific classes and methods by 1999 (D. Schwarz and M. Wright 2000).

As OpenMusic developed, Mikael Laurson, the creator of PatchWork, was independently developing PatchWork into PWGL (PatchWork Graphical Language) (Laurson and Kuuskankare 2003), a system quite similar to OpenMusic in its graphical approach. PWGL adopts a fundamentally different stance with regard to computationally modeling the details of musical notation. While OpenMusic requires export to a typography program to make choices beyond pitches and rhythms, PWGL provides ENP (Expressive Notation Package) for composers who want to work computationally with common notation symbols (Kuuskankare 2009). (It should be emphasized that this limited model of notation has not prevented composers from successfully realizing their ideas using OpenMusic, as documented in IRCAM's two-volume review of projects created using their environment (Agon, Gérard Assayag, Bresson, and Puckette 2006, Agon, Gérard Assayag, and Bresson 2008).) PWGL also developed an interface to synthesis parameters (Laurson, Norilo, and Kuuskankare 2005) and analysis data (Kuuskankare 2012b, as well as an interface for graphic notation (Kuuskankare and Laurson 2010) and constraint programming (Laurson and Kuuskankare 2006). The Meta-score graphical editor combines procedural programming, common notation, and timeline-based event specification into a single GUI (Kuuskankare 2012a).

Live and Interactive Notation

Some of the most innovative object-oriented musical notation models have been created for applications in which a notation is generated live in realtime with computer assistance, or a pre-composed notation is presented during a performance

by means of computer animation. Harris Wulfson's LiveScore system models notation in the Java programming language via NoteStream objects, which each contain a succession of notes, accompanied by text instructions and dynamic markings; in his composition, *LiveScore*, audience participants tune the knobs of a mixer interface to alter the ranges of musical parameters constraining the output of an algorithmic composition engine (Wulfson, Barrett, and M. Winter 2007, Barrett and M. Winter 2010). Luciano Azzigotti created a similar system in the Processing environment, a simplified dialect of the Java programming language intended to teach artists and designers basic programming skills (Reas and Fry 2007, Azziggoti 2012). Throughout the field of music, increased computation power and programming environments tailored to realtime computation have made it easier for composers to creatively refashion notation to satisfy new goals of collaboration and realtime interactivity (Balachandran and Wyse 2012). As these new trends are equally likely to engage abstract animation and data representation traditions of information display as they are traditional musical notation, they tend to result in computer models of notation that offer either a simplified set of common practice notational constructs or a novel approach to notation suited to a particular performance application; these applications are a good example of the way in which the set of generative tasks that interest current practitioners may reduce or discard the full range of accepted common practice notational constructs (Nicolas Collins 2011).

These new systems cast a distinctly contemporary light on automated notation systems oriented toward a document preparation model of notation production. Whereas computer notation systems could previously agree implicitly to participate in common practice tradition without argument, the proliferation of new approaches to notation in the realm of interactive media marks document preparation systems for common practice notation as definitively conservative technologies. As such, they conceptualize new technology as an assistive technology that aids, enhances, or re-approaches an established notational technology, as opposed to a force for the creation of a radically new paradigm of musical collaboration through graphic media.

Constraint Solvers

The most recent trend in the algorithmic development of automated notation programs has been the integration of constraint solvers, which allow the user to describe the result of a process, without describing the means by which the results must be achieved, a paradigm known in computer science as *declarative programming*. A program consists of a descriptive logic, which specifies what to do, and control, which specifies how to do it (Kowalski 1979); the former is called *declarative programming* and the latter *imperative programming*. Constraint programming is a form of declarative programming, in which the programmer specifies logical constraints that describe the conditions that must be satisfied, without stating exactly how they will be (Van Roy and Haridi 2004, p. 749); when coupled with an existing imperative language, constraint solvers enable a kind of meta-programming (Lloyd 1994). While constraint-solvers depend traditionally on boolean expressions, recent work has devised constraint-specification syntaxes specific to the needs of musical applications, which include arbitrarily chaining the score elements to which constraints apply, as well as specifying constraints that consider relationships between score elements (Anders and Eduardo R Miranda 2008b). Constraint solvers have been used to model specific musical structures, such as melodies (Zhong and Zheng 2005) and polyphony (a conflation of harmony/counterpoint) (Courtot 1990), as well as composition more broadly (Desainte-Katherine and Strandh 1991).

An increase in computational power has facilitated the use of constraint solving for complicated musical decisions, and constraint solvers have been integrated into widely used automated notation environments during the last decade. These systems are now fast enough to use in realtime applications (Anders and Eduardo R Miranda 2008a), as well as in document-oriented notation applications. OpenMusic and PWGL both contain harmonically oriented constraint solvers, and Strasheela is a powerful text-based constraint solver for musical applications (Sandred 2010); the developers of Abjad are currently integrating a constraint solver that can be arbitrarily applied to any of the components in a graph tree representation of a musical score (Bača 2013). For the programming composer, the basics of procedural and object-oriented programming might soon be displaced by the careful description of con-

straints; this development could potentially lower the bar for composer entry into automated notation, because constraint functions as an interface to the automated arrangement of hidden primitive objects and functions, the low-level manipulation of which need not be mastered by the user in order to produce results that meet the specified constraints. Such a system would present a new incarnation of the trade-off between "user friendliness" and technological transparency, potentially minimizing the intricacies of procedural programming for composers.

1.3 Design Values for Automated Notation Systems, Illustrated with the Abjad API for Formalized Score Control

1.3.1 The Abjad API for Formalized Score Control

Abjad is a mature, fully-featured system for algorithmic composition comprising, at the time of writing, more than 178,000 lines of code divided into 50 public packages, 305 public classes, 1003 public functions and a documented API totaling more than 800 pages. The system is not built to implement any one idea of what composition is. Abjad is instead architected in such a way as to encourage composers, music theorists and musicologists to model and implement their own, perhaps highly idiomatic, understandings of what musical score is and how music is to be written, analyzed and understood. After a survey of existing automated notation systems, the author has come to regard it as an example of several desirable design values: it allows the user to navigate complex score hierarchies with a readable syntax and access to both high-level and low-level symbolic manipulations, it contains a sufficiently detailed object model of common practice music notation, in which the user may automate the placement of any of the modeled notational symbols, its second-order relationship to generated notations affords tweakability, and its basis in the Python programming language affords extensibility. This section introduces the Abjad API and elaborates on these design criteria.

Abjad Wraps Lilypond

Abjad is a Python API that creates formatted Lilypond syntax for the generation of notation by the Lilypond automated music typsetting engine. LilyPond is an automated music typesetting program, created in the C++ and Scheme programming languages (Nienhuys and Nieuwenhuizen 2003). Inspired by the deficiencies of computer typesetting work from the last years of the 1980s, LilyPond represents over a decade of research into a text-based interface for the notational constructs of common practice notation, as well as the typographical details and layout of the score as a document (Schankler 2013). By providing an interface to a sufficiently detailed low-level model of notation, Abjad provides automated access to all the specifics of the score as a document, including typographical details such as a text indications and articulations, as well as format and layout details such as page size and font details.

As a minimal example, the code below creates an Abjad measure and then both encodes and displays the measure via LilyPond, by using the `show` function:

```
>>> measure = Measure((5, 8), ”c’8 d’8 e’8 f’8 g’8”)
>>> f(measure)
{
    \time 5/8
    c’8
    d’8
    e’8
    f’8
    g’8
}
>>> show(measure)
```

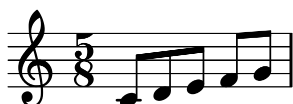


Figure 1.1: Abjad creates notation by scripting the LilyPond typesetting program.

Inception

The code that was to become Abjad began in 1997 and 1998 as the independent work of composers Trevor Bača and Victor Adán. At that time the composers

were working with compositional applications of the electroacoustic techniques of granular synthesis and spectral convolution as well as with matrices and other structures from linear algebra, the use of one- and two-dimensional recursive series to model rhythm, and the use of imaging data from graphic input tablets to model geometric transforms of independent musical parameters. The composers found these and many other ideas from group theory, graph theory and computer science to be imminently compositionally useful. But again and again the barrier to the musical exploration of these ideas was found to be the transcription of these objects into the standard notation of musical score (Bača 2010).

Bača and Adán have been joined by composer Josiah Oberholtzer as principle architects of the system, and composers, such as the author, have shaped the design of the open-source system by communicating their needs while designing their own compositional applications. Two examples of this feedback between composition and system design follow.

Lidércfény

Lidércfény is a 15-minute work for flute, violin and piano. The piece is the work of Trevor Bača and was written in 2007—2008. During the composition of the piece Bača used Abjad to render hundreds of rhythmic structures as a fully notated score. The composer worked iteratively and selected the best results from each round of output for use as input to the next round of work with Abjad. Estimating two and a half handwritten pages of score an hour, this process would have taken four years to complete by hand.

Also important to the construction of the piece was the implementation of the Spanner class. The spanner is a structural component unique to Abjad. Spanners play the role of hierarchy-breaking objects that cross over tree-like parts of the musical score. Bača took inspiration for the Abjad spanner from legal publications that posit the idea of a neomedieval overlapping of legal systems in the emergent transnational institutions of the European Union. The Spanner class is now included in the Abjad public library.

The rhythmic construction of Lidércfény shows how the iterative and tran-

scriptional work that Abjad does well can be leveraged in such a way as to reserve the work of creative elaboration for the composer. And the object-oriented flexibility of Abjad made it possible to combine ideas from computer science and jurisprudence in the writing of a piece of music.

Aurora

Aurora is a work for 22-voice string orchestra by Josiah Wolf Oberholtzer. It was commissioned in 2011 by Berlin's Ensemble Kaleidoskop for a festival commemorating the 10th anniversary of Iannis Xenakis's death. The composer had two main interests when architecting the piece. First, it should be composed of massed clouds of overlapping material, clouds which could permeate, mask or otherwise be superpositioned relative one another. Second, the atoms comprising those massed clouds would be conceived not principally as streams of pitches and rhythms, but as small series of microgestures built from the conglomeration of classes representing idiomatic string techniques. Each instrumental line in Aurora results from multiplexing the traces from each cloud containing that instrument into a single stream, allowing a performer to participate in different composition processes from moment to moment.

Oberholtzer developed the Abjad timeintervaltrees API to accomplish this large-scale formalization. The interval tree is an ordered collection of absolutely-positioned blocks of time to which arbitrary data can be attached. Interval trees can be scaled, split, shifted and exploded without regard for instrumentation or meter because interval trees model the metascore positioning of musical material. Working with Abjad interval trees allows composers to work with large amounts of material that can be rendered as publication-quality notation later in the compositional process.

1.3.2 Design Recommendations

A Sufficiently Detailed Model of Notation

In 1971, Cantor writes, “A full display editor for music would take years to develop, with unforeseen difficulties along the way. To begin, one should construct and use an editor for the notation of some small repertoire” (Cantor 1971, p. 107). The perilous recommendation that notation should be modeled gradually, moving on to more advanced constructs later, rather than creating at the outset a representation that enables both simple and complicated notational constructs, has left even 21st-century notation editors with overly simple models of notation. Even the most sophisticated commercial editors, for example, advance a model of music in which the measure acts as a system atom, despite the presence of non-mensural notational constructs in every period of notated musical history.

Readable Navigation of a Hierarchically Organized Score

Because a score is necessarily a hierarchical arrangement of symbols, the user needs fluid access to a robust object system, the various levels of the hierarchy in which the symbols have been organized, and the ability to filter collections of objects based on the comparison of their properties. Anything else is an impoverished interface to the basically symbolic nature of musical notation — anything else, at least, from a conventional aesthetic viewpoint that prioritizes controlled expression over unpredictability and chance (Gurevich and Treviño 2007). In line with the cognitive priorities of Djikstra and Kay, these goals should be accomplished in as conceptually elegant a way possible, with a programming syntax that closely aligns with the patterns of human thought in this area of application. Stated from the perspective of the user, rather than the designer, this means that the user must be able to forget, reapproach, and newly understand the function of code. “Readability” is paramount for the execution of large projects and to some extent at every moment of the feedback loop between coding and thought. Different programming languages afford readability differentially: Python’s syntax, for example, conflates scope with indentation, enforcing through its syntax a convention of good programming style. The

simple presence of structured "white space," such as indentation and blank lines, has been shown to increase code readability more effectively than even comments, the first resort of programmers concerned with documenting their code effectively for others (Buse and Weimer 2010). The criss-crossing patch cords of graphical programming languages encourage write-only code, while enabling rapid prototyping, and even the syntax of the most elegant object-oriented languages can decrease the readability of code due to the cumbersome task of navigating a hierarchy of symbols.

Consider the difficulties of the following common notational task — that of adding phrasing slurs to groups of notes and chords surrounded on either side by a succession of rests (as in Figure 1.2).

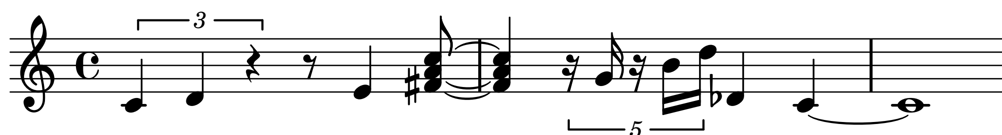


Figure 1.2: Rest-delimited notes and chords.

Described as a cognitive process, the task might be described in two simple steps: 1) segregate the symbols on the staff into groups of rest and non-rest symbols; 2) add a slur to each non-rest group of symbols. While this first step takes a matter of seconds for the gestalt grouping abilities of the human perceptual system (Quinlan and Wilton 1998), it can be a laborious process for an automated notation system.

```
(let (group) (dolist (voice (collect-enp-objects score :voice))
(dolist (chord (collect-enp-objects voice :chord)) (if (rest-p chord)
(progn (when (cdr group)
(insert-expression (reverse group) expression)) (setq group NIL))

(push chord group)))))
```

Figure 1.3: Code to slur groups of rest-delimited notes and chords in PWGL/ENP.

In PWGL's ENP, the code in Figure 1.3 adds slurs to rest-delimited, mixed groups of notes and chords. Although the code refers clearly to the elements of common notation — names like "chord" and "score" indicate that the user has access to a representation of the score — its many nested, parenthesized arguments do not elegantly map to the plain-language description of procedures above. This is largely because the code's syntax introduces a collection of metaphors foreign to the two verbs

of the simple description — the succession "group" and "slur" from the original formulation has become "collect," "collect," "insert," "reverse," "set," and "push" — with the effect of complicating the operation beyond its necessary complexity.

Data abstraction can help reduce syntactic clutter and align the language of code with the language of concept. Abjad's built-in iteration functions leverage Python's ability to iterate through lists of symbols, resulting in the following elegant syntax for the above task:

```
>>> staff = Staff( r"\times 2/3 { c'4 d' r } r8 e'4 <fs' a' c''>8 ~ q4 \times 4/5
    { r16 g' r b' d'' } df'4 c' ~ c'1" )
>>> for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(staff.
    leaves, (Note, Chord)):
...     spannertools.SlurSpanner(group[:])
...
SlurSpanner(c'4, d'4)
SlurSpanner(e'4, <fs' a' c''>8, <fs' a' c''>4)
SlurSpanner(g'16)
SlurSpanner(b'16, d''16, df'4, c'4, c'1)
>>> show(staff)
```

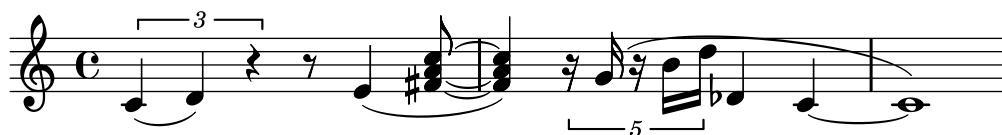


Figure 1.4: Slurred groups of rest-delimited notes and chords.

Using a "for" loop, Python groups the leaves (rests, skips, notes, and chords) of a staff container (the function's first argument) by segregating leaves into groups that consist exclusively of notes and chords and groups that do not (the second argument), and then iterates through each note/chord group, slurring each. The language of the code aligns well with the plain language task description, and there are essentially two operations: "group" (or more accurately, "for group in groups,") and "slur."

Transparency Affords Tweakability and Extensibility

Low-level control over typographical detail and high-level procedural manipulation are seldom found in the same system: many commercial notation programs offer exquisite low-level interfaces without any high-level procedural abili-

ties, and many of the most widely used systems offer an impoverished set of or interface to the symbols of common practice notation. Integrated low- and high-level control over notational symbols encourages two important benefits of technological transparency in the input and output of the system: *extensibility* and *tweakability*. Extensibility assumes that understanding of the low-level construction of a system might enable extensions to the system, to afford new applications or more flexible alignment between thought and individual programming style, while tweakability — related especially to the system's output — allows the programmer to engage in low-level manipulations of materials generated by higher-level specifications.

Tweakability is a persistent issue in automated notation systems, and many notation systems create workflows that invite the user to address either high- or low-level symbolic manipulations. Because Abjad wraps LilyPond and extends an interpreted language that can be used live in a terminal, the system affords a wide variety of uses. Adopting the paradigms of creativity offered by McLean and Wiggins (McLean and Geraint Wiggins 2010), a "planner" user might elect to write a program that generates the entire composition, as is the case with Josiah Oberholtzer's string ensemble composition, *Aurora* (Oberholtzer 2010), while a "bricoleur" user might generate LilyPond syntax live in the interpreter, copying and pasting into a LilyPond document to be meticulously tweaked at a note-by-note level. The system affords note-by-note composition, totalized algorithmic composition, and many hybridized approaches between these extremes.

Extensibility is an important design value, both as it applies to the user's ability to extend a system and the ability of a system to integrate diverse, extant modules of code. The relevance of extensibility to the user's experience depends heavily on the difference between the system interface offered to its programmers and to its users: if users engage the system with the same knowledge model as programmers, extensibility in this sense is highly relevant; for systems with a large knowledge asymmetry between programmer and user, extensibility has been romanticized to the point of assuring that amateur programmers will be able to achieve expert results (Standish 1975). Many automated notation environments assume that their users are programming composers rather than composing programmers, and a large

information asymmetry often exists between programmer and user. Even in modern systems touted as object-oriented, users cannot take advantage of the ability to create new classes in the system, because the system's documentation is oriented exclusively toward the use of existing system classes, rather than their extension or modification.

A second understanding of extensibility is perhaps more relevant to programming for artistic applications. In an age with a surfeit of extant code, the concept of extensibility can be rehabilitated as an assessment of a system's ability to integrate modules of code written in diverse languages and for diverse applications. Such an evaluation is especially relevant to artistic creativity, a realm of activity fraught with interdisciplinary bricolage; for example, recent work in computer-aided algorithmic composition proves that the practice of borrowing concepts and mathematical equations from scientific fields for novel musical applications remains alive and well (C. Magnus 2010; Washka 2007; Zad, Araabi, and Lucas 2005; Acevedo 2005; Gartland-Jones and Copley 2003; Phon-Amnuaisuk, Tuson, and Geraint Wiggins 1999; G.A. Wiggins et al. 1998; E. Miranda 2007; Burraston et al. 2004; Kröger et al. 2008; Laine 1997; Hörnel 1997; Melo and Geraint Wiggins 2003; Spicer 2004; Luque 2009; Peters 2010; Essl 2006).

With a priority of freely integrating ideas and code from disparate realms of inquiry, languages can be meaningfully evaluated as relatively disciplined. For example, the LISP programming language remains popular in the fields of Artificial Intelligence, Linguistics, and Music, while science and design disciplines have embraced modern object-oriented languages. Given the manifold needs of programming artists, successful integration is requisite for a suitably flexible environment, and a system's interoperability and breadth of use play an important role in the artistic limitations of a system; the Python language, for example, has demonstrated success as "glue" between various languages and application domains (Sanner and et. al. 1999). As a language gains a reputation for flexible interoperability, programmers create utilities for this language that further increase the language's abilities in this realm (Beazley and et. al. 1996).

The openness of a software environment also constrains extensibility, and

the interfaces of open-source software development encourage software extensibility. Fees and licenses can prevent users from making valuable contributions. For example, although IRCAM's OpenMusic is open source in one sense — the code can be freely downloaded and revised — the language's basis in LispWorks Common Lisp requires that third party developers buy a license to compile OpenMusic for the testing of additional libraries; this has been a notable barrier to the anarkomposer project, an open-source tool for flexible input/out across automatic notation systems (Echevarría 2013, *anarkomposer SCM Repository*). As an entirely open-source project in an online code repository, with (Sub)version (SVN) version control and class docstrings that integrate code testing and documentation via Sphinx/ReST (*reStructured Text Primer – Sphinx 1.1.3 Documentation*), Abjad streamlines the process of user contribution.

Enabling High-/Low-level and Procedural/Object-oriented Automation

By allowing as much technological transparency as possible, with the goal of ensuring extensibility and tweakability, a system should allow a composer to specify both higher-level relationships that lead to notated results and lower-level procedures that place each notational symbol, one at a time, as desired. (It is conceivably the case that a notational practice might consist entirely of the formalization of the number and position of musical symbols, as has been suspected of the composition of Erik Satie's *Vexations* (Orledge 1998). To solve a given problem, a programmer might think of a number of interacting agents, a mill-like succession of inputs and outputs, or some combination of the two, all embroiled in the infinite field of metaphor that informs and underlies human thought (Lakoff 1980). This is especially true in musical thought, in which entire theories of music can theorize the same basic elements as dynamic processes or stable entities (Berger 1994). This fluidity makes it essential to enable meta-formalizations that place procedural and object-oriented conceptions of the same material into discourse with one another; for example, a sequence of pitches might be considered in time, as an unfolding pattern, as well as out of time, as a structure with characteristics, and the outputs of these two models might usefully inform one another to create contextually aware processes (Hedelin 2008); multiple

statistical models of the same musical elements might influence the production of the notation (Pearce, Conklin, and Geraint Wiggins 2005). For this reason, the musical programmer should be able to accomplish tasks with a flexible mix of procedural and object-oriented programming, as afforded by hybrid programming languages like LISP, Java, and Python. While several systems allow the user to define new procedures, based on built-in objects and classes, it remains unclear in most systems how the user might instantiate new classes.

The performance of contemporary music can be a complicated physical task, and notation often describes physical gesture or position through the use of tablature (Rastall 1983, p. 143); recent work has begun to integrate the constraints of the human hand with the composition of music in automated contexts (Truchet 2004). As an example of low-level typographical control enabled by both procedural and object-oriented thought, consider the creation of an alternative woodwind fingering diagram, as required for the description of multiphonic sounds in contemporary composition (Backus 1978).

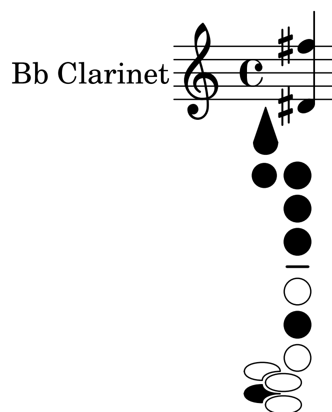


Figure 1.5: A multiphonic notation, including a woodwind diagram.

As important as the visual depiction of physical contact with instruments has become for contemporary notation practice (Alberman 2005, Cassidy 2004, Kanno 2007), no automated notation system has developed an interface for creating woodwind diagrams. For this reason, the user must extend the system by writing new code; however, in most systems, access to document preparation and low-level typographical operations remain too hidden to allow the user to do this. Because Abjad wraps the LilyPond typesetting package, the author was able to create a new Python

interface to Mike Solomon's LilyPond woodwind diagrams (2010). As demonstrated in the code appendix, the above diagram can be implemented variously as a procedure that acts on lists of keys to depress, written with only built-in string manipulation functions from Python's standard library (A.1), as a procedure written more economically with scheme syntax functions from Abjad's schemetools library, (A.2), and lastly with object-oriented programming as a documented WoodwindDiagram class (A.3).

In addition to the above extensibility, the system preserves tweakability: the LilyPond format of the above diagram is easily accessed for copying, pasting, and tweaking, using the `f()` (format) function:

```
>>> f(fingering)
\woodwind-diagram #'clarinet #'((cc . (one two three five)) (lh . (R thumb)) (rh .
    (e)))
```

This notational interface is relatively automated, in that it creates a diagram representing all of the instruments keys, without demanding that the user specify the positions of each constituent filled or unfilled shape; however, the user retains control of low-level visual details, with the use of graphic overrides, and can alter the symbolic or graphical representation of the instrument's keys (the `graphical` markup command), the size of the diagram (the `size` markup command), and the thickness of the lines used to render the diagram (the `thickness` markup command):

```
>>> not_graphical = markuptools.MarkupCommand('override', schemetools.SchemePair('
    graphical', False))
>>> chord = Chord("e' as' gqf''", (1,1))
>>> fingering = instrumenttools.WoodwindFingering('clarinet', center_column=['one
    ', 'two', 'three', 'four'], left_hand=['R','cis'], right_hand=['fis'])
>>> diagram = fingering()
>>> graphical = markuptools.MarkupCommand('override', schemetools.SchemePair('
    graphical', False))
>>> size = markuptools.MarkupCommand('override', schemetools.SchemePair('size',
    .5))
>>> thickness = markuptools.MarkupCommand('override', schemetools.SchemePair('
    thickness', .4))
>>> markup = markuptools.Markup([graphical, size, thickness, diagram], direction=
    Down)
>>> markup.attach(chord)
Markup((MarkupCommand('override', SchemePair(('graphical', False))), MarkupCommand
    ('override', SchemePair(('size', 0.5))), MarkupCommand('override', SchemePair
    (('thickness', 0.4))), MarkupCommand('woodwind-diagram', Scheme('clarinet'),
```

```
    Scheme([SchemePair(('cc', ('one', 'two', 'three', 'four'))), SchemePair(('lh',
     ('R', 'cis'))), SchemePair(('rh', ('fis',)))]))), direction=Down)(<e' as' gqf
    ''>1)
>>> staff = Staff([chord])
>>> contexttools.InstrumentMark('Bb Clarinet', 'clar.')(staff)
InstrumentMark(instrument_name='Bb Clarinet', short_instrument_name='clar.')(Staff
    {1})
>>> score = Score([staff])
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
```

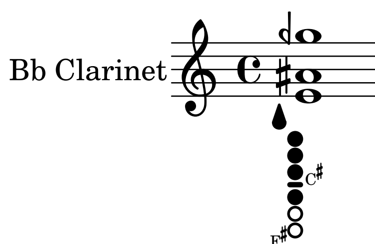


Figure 1.6: Graphic overrides change the appearance of a woodwind diagram.

Document Preparation

When evaluated by a conservative schema — that an OOP system for notation should provide an object-oriented interface to as thorough a model of common practice notation as possible and should enable the composer to control algorithmically the layout and formatting of the score as a printable document, before proceeding to newer models of composer-technology interaction — most systems perform poorly. Musical notation is graphic, and manual control over the visual aspect of a notation is necessary from the outset (Dannenberg 1993). A system should be able to cleanly bridge the gap between composing and document preparation, with algorithmic control over the parameters of the document; however, much of the automated notation work from the last twenty years postpones the most important typographical and formatting choices until after translation into a format appropriate for a musical typesetting program, requiring an export to MIDI, MusicXML, or Lilypond before choices about dynamics, articulations, document formatting, and document layout can be specified. This is not a problem if composition is fundamentally the determination of pitches and rhythms; if anything besides these two musical parameters could potentially occupy a status other than decoration, then an

alternative approach might be necessary.

Conclusion — Plurality and Fluidity of Generative Task Complicate the Evaluation of System Design

Design recommendations for user interaction and feedback remain elusive, despite the above survey, because systems are designed with various applications in mind. For this reason, the range of available user interfaces varies from score modeling systems that produce no notation and rely on auditory feedback for the results of text input, such as Andrew Sorensen's *impromptu* language for live coding (Sorensen 2013), to the conventional point-and-click paper simulations of commercial notation software. Should an automated notation system include an interface to a synthesis engine? Yes, according to BACH and PWGL, and no, according to many other systems. This multitude of implied generative tasks complicates comparative system evaluation of interaction and feedback. In addition to potential applications, aesthetic predilection plays a role, too, and one that extends beyond music and into the process of composition immediately: for example, it is arguably more desirable to compose music with the predictive feedback of imagination alone, despite the ready availability of computer applications that create "mock-ups" of a composer's work (Morris 2002).

A system's generative task can be both plural and fluid. It might be plural if developers hold different concepts of the system's intended use but can agree sufficiently on a certain set of primitives that must be included. In commercial systems, profit becomes a generative task, and the changing demands of a user-base create a constantly shifting agenda for development. It is also the case that systems created for the work of single authors can be suddenly redirected at larger groups of users, causing a radical change in the direction of development but leaving the indelible fingerprint of the system's earlier goals.

Most broadly, these automated notation systems rest upon a common generative task: drawing. But in the same way that order eliminates noise and neutralizes poltical unrest (Attali 1985), automated notation systems insist that musical notation resembles the symbolic arrangements of language more than drawing or

painting, despite a fifty-year-old tradition of deliberately ambiguous relationship between music and abstract graphic art (Evarts 1968, Cardew 1961). The systems described here encroach upon the expressive potentials of drawing's analog creativity: a digital, inherently parametric control has usurped the analog control of the human hand's representative capacities. In this sense, an automated notation system cannot help but be impoverished, relative to the graphic potential of physically enacted representation — but automation excuses itself by hoping to derive benefits orthogonal to those of drawing. Automation eats and metabolizes drawing, to fuel a marathon of symbolic processing. As Glendon Diener writes,

> Striking the delicate balance between *structural organization* on the one hand and *graphical generality* on the other is a major issue in the design of common music notation systems. The problem, described by Donald Byrd as the "fundamental tradeoff" between *semantics* and *graphics* (Byrd, 1986), is readily understood by imagining musical versions of the 'draw' and 'paint' programs available on many small computers. A musical draw program could facilitate high-level editing and performance operations by means of the data structure analogs of notes, staves, parts, and the like, but as a consequence would limit its visual universe to some finite collection of pre-defind symbols. By contrast, a musical paint program by imposing no further organization on its data than that of a two-dimensional array of pixels, would gain graphical generality at the expense of its ability to perform musically meaningful operations on that data (Diener 1989).

This dilemma suggests a final qualitative criterion by which one may evaluate an automated notation system: an automated notation system implicitly models the range of graphic variability required by composers and proposes a point of balance in a requisite compromise between semantics and graphics.

Chapter 2

Computational Modeling as Analysis

2.1 The Conflation of Analysis and Composition Reveals and Posits Construction

Formalized score control conflates the analysis and composition of music: now that the abstract formulation of musical order precedes its instantiation as a notation, it is possible to analyze a composer's code to arrive at new insights about the structure of the music and the cognitive processes at play during composition. The creative process gains increased transparency via two routes: first, because of the text-based nature of code, formalized score control reveals the role of metaphor in the creative process; second, a musical analysis may be tested by implementation, as a valid declarative logic implemented as an imperative program that recreates the score. More radically, a musical analysis might be created first as an imperative program that recreates a score. Two examples of this last approach, in which an analysis proceeds first as an imperative sequence of commands, iteratively revised with the goal of recreating a score, are discussed for the remainder of this chapter, after a brief discussion of the variety of metaphors encountered in composers' programs.

2.1.1 Formalization Reveals Metaphor

A study of Java programmers revealed programs built upon metaphors of explored spatial locations, sentient beings, dancing symbols, buzzing sounds cre-

ated by absent code, graphic-mathematical transformations, mechanical apparatus, and conversations with intelligent agents (Blackwell 2006), and correspondence with programming composers reveals a plethora of idiosyncratic formulations that underly automated notation programming. These include quantifying metaphors, in which a composer invents a quantitative system to specify numerically a previously qualitative dimension of music, such as Clarence Barlow's systematic formalization of acoustic consonance and dissonance via "indigestibility functions" (Barlow 2011) or Pablo Cetta's similar work (Cetta 2011); midwife metaphors, which allow a user community to describe the use of a notation system, whether or not the metaphor appears in the code itself, such as the use of "pouring" notes into containers in the Abjad documentation (Bača 2011); score metaphors, in which the composer improvises an object-based system to track the division of a work into sections, such as José Lopez-Montez's division of his composition, *Autoparaphrasis*, into summarizing procedures — "...granulated sound...explosion...mega-trumpet...granulation in decomposition...hyper-acute... intermediate...disintegration...recapitulation" (author's translation from the Spanish) — which describe the sequence of events in the work (Lopez-Montes 2011); the graphic metaphors of visual programming environments, in which the spatial arrangements of objects organize and communicate data flows; the built-in prescriptive metaphors of programming languages, such as "flattening a list" to remove embedded parentheses; and disciplined metaphors, imported into code via an academic discipline, such as the use of "tree" and "leaf" from graph theory.

2.2 Reverse Engineering as Analysis: Two Case Studies in Formalized Score Control as Analysis

2.2.1 *Cantus in Memory of Benjamin Britten* (1977-1980) by Arvo Pärt

The Composition

Cantus in Memory of Benjamin Britten by Estonian composer Arvo Pärt was composed from 1977 to 1980 and published by Universal Edition in 1980 (Pärt 1980). The composition was originally conceived as a series of simple rules governing scale descents and durational relationships between parts, recorded on a napkin during a train ride (David Cope 2010).

The Approach

The task of creating a program that would generate the published score exactly was approached as a test of the work's origin myth; that is, the working hypothesis from the outset was that the entire composition would be easily expressible using a few functions that act recursively to create a complex effect. As the following code shows, this turned out to be true: a single recursive function creates almost all of the work's pitches and rhythms. This is unsurprising, as an analysis of the score reveals the composition to be a simple prolation canon, in which register correlates to a doubling of duration and a one-octave decrease of register relative to the next higher string voice; all parts descend the a natural minor scale until the work's coda, which cannot be modeled with such straightforward rules. (Subsequent revisions of this code by Josiah Oberholtzer reimplemented much of the procedural code here as object-generating classes and revised many of the procedures to take advantage of some of Python's indigenous idioms, such as dictionaries; this code is part of the Abjad manual and is freely available online at projectabjad.org).

The Code

The following code generates the score for Arvo Pärt's *Cantus in Memory of Benjamin Britten* (1980, following the Universal Edition, Philharmonia Series #555).

The code begins with a typographical wrapper function, which enables the embedding of custom fonts within Lilypond documents:

```
>>> def fonted(aString):
...     fontString = ”\\override #’(font-name . \”Futura\”)”
...     outString = fontString + ” {”+ aString +”}”
...     return outString
...
```

Figure 2.1: A font function enables custom typefaces.

Next, the program models the score, beginning with the staffs and their names:

```
>>> bell = Staff([])
>>> contexttools.InstrumentMark( fonted("Campana in La"), fonted("Camp.")  )(bell)
InstrumentMark(instrument_name='\\override #\'(font-name . "Futura") {Campana in
    La}', short_instrument_name='\\override #\'(font-name . "Futura") {Camp.}')(
    Staff{})
>>>
>>> violin1 = Staff([])
>>> contexttools.InstrumentMark( fonted("Violin I"), fonted("Vl. I") )(violin1)
InstrumentMark(instrument_name='\\override #\'(font-name . "Futura") {Violin I}',
    short_instrument_name='\\override #\'(font-name . "Futura") {Vl. I}')(Staff{})
>>>
>>> violin2 = Staff([])
>>> contexttools.InstrumentMark( fonted("Violin II"), fonted("Vl. II") )(violin2)
InstrumentMark(instrument_name='\\override #\'(font-name . "Futura") {Violin II}',
     short_instrument_name='\\override #\'(font-name . "Futura") {Vl. II}')(Staff
    {})
>>>
>>> viola = Staff([])
>>> contexttools.InstrumentMark( fonted("Viola"), fonted("Va.") )(viola)
InstrumentMark(instrument_name='\\override #\'(font-name . "Futura") {Viola}',
    short_instrument_name='\\override #\'(font-name . "Futura") {Va.}')(Staff{})
>>> contexttools.ClefMark('alto')(viola)
ClefMark('alto')(Staff{})
>>>
>>> cello = Staff([])
>>> contexttools.InstrumentMark( fonted("Cello"), fonted("Vc.") )(cello)
InstrumentMark(instrument_name='\\override #\'(font-name . "Futura") {Cello}',
    short_instrument_name='\\override #\'(font-name . "Futura") {Vc.}')(Staff{})
>>> contexttools.ClefMark('bass')(cello)
ClefMark('bass')(Staff{})
>>>
>>> bass = Staff([])
>>> contexttools.InstrumentMark( fonted("Contrabass"), fonted("Cb.") )(bass)
InstrumentMark(instrument_name='\\override #\'(font-name . "Futura") {Contrabass
    }', short_instrument_name='\\override #\'(font-name . "Futura") {Cb.}')(Staff
    {})
>>> contexttools.ClefMark('bass')(bass)
ClefMark('bass')(Staff{})
```

Figure 2.2: Modeling the bell and string staffs and their names.

Note that this also includes the appropriate clefs for the staffs, as well as both short and long names for each staff. Next, the program groups the string staffs with a bracket, and adds both a time signature and a tempo; because low-level typographical detail can be adjusted, this step also specifies the space between the instrument

name and the left edge of the staff:

```
>>> strings = scoretools.StaffGroup([violin1, violin2, viola, cello, bass])
>>> bell.override.instrument_name.padding = 3
>>> for staff in strings:
...     staff.override.instrument_name.padding = 3
...
>>> score = Score([])
>>> score.append(bell)
>>> score.append(strings)
>>> contexttools.TimeSignatureMark((6,4))(bell)
TimeSignatureMark((6, 4))(Staff{})
>>> tempo = marktools.LilyPondCommandMark('tempo 4 = 112~120 ')(bell)
```

Figure 2.3: Adding string staffs to a score.

Next, the program models the bell part. This is straightforward using a series of data abstractions that call one another, and the entire part results from the concomitant use of several functions. As the levels of musical structure addressed by the functions grow, the code leaves the realm of notation modeling – the modeling of a "bar" – and fluidly enters the realm of music modeling – the modeling of a "phrase":

```
>>> def bellBar():
...     bar = Measure((6,4),"r2. a'2.")
...     marktools.LilyPondCommandMark("laissezVibrer",'after')(bar[1])
...     return bar
...
>>> def restBar():
...     return Measure((6,4), "r1.")
...
>>> def couplet():
...     return Container([bellBar(), restBar()])
...
>>> def bellPhrase():
...     container = Container([])
...     container.extend([couplet(), couplet(), couplet(), restBar(), restBar()])
...     return container
...
>>> def bellPart():
...     container = Container()
...     container.extend(bellPhrase()*11)
...     container.extend(restBar()*19)
...     lastBellBar = Measure( (6,4), "a'1.")
...     marktools.LilyPondCommandMark("laissezVibrer",'after')(lastBellBar[0])
...     container.append(lastBellBar)
...     return container
...
>>> bell.append(bellPart())
>>>
>>> def sixBarsRest():
...     restBars = Container([])
...     restBars.extend(Measure( (6,4), "r1.")*6)
...     return restBars
...
>>> for staff in strings:
...         staff.append(sixBarsRest())
...
```

Figure 2.4: Modeling the bell part.

Next, the program models most of string material, beginning with the pitch material: by creating a set of functions to generate register-dependent descents down the a minor scale; the resulting function is applied to generate a "descent reservoir," a list of pitches, for each instrument, and then a "contoured descent" for each instrument, in which the pitches descend from the top of the scale by one more pitch each time before returning to the top of the reservoir:

```
>>> def descentReservoir(numOctaves,transposition,lastNote):
...     theKey = contexttools.KeySignatureMark('a', 'minor')
...     notes = tonalitytools.make_first_n_notes_in_ascending_diatonic_scale(7*
    numOctaves+1, key_signature=theKey)
...     notes.reverse()
...     for note in notes:
...         note.written_pitch = note.written_pitch + 12*transposition
...     outContainer = []
...     for note in notes:
...         if note.written_pitch >= lastNote:
...             outContainer.append(note)
...     sequencetools.flatten_sequence(outContainer)
...     return Container(outContainer)
...
>>> violin1Res = descentReservoir(3,-1,0)
>>> violin2Res = descentReservoir(2,-1,-3)
>>> vlaRes = descentReservoir(2,-2,-8)
>>> celloRes = descentReservoir(1,-2,-15)
>>> bassRes = descentReservoir(1,-2,-12)
>>>
>>> def contouredDescent(reservoir):
...     cd = []
...     for x in range(len(reservoir)):
...         cd.append(list(reservoir[:x+1][:]))
...     cd = sequencetools.flatten_sequence(cd)
...     return cd
...
>>> vln1cd = contouredDescent(violin1Res)
>>> vln2cd = contouredDescent(violin2Res)
>>> vlacd = contouredDescent(vlaRes)
>>> cellocd = contouredDescent(celloRes)
>>> basscd = contouredDescent(bassRes)
```

Figure 2.5: Modeling pitch as a series of scalar descents.

The next step in modeling the pitches addresses the relationship between the lower notes of the score's diads and the descending upper notes. Declaratively, this relationship may be described succinctly: "The lower note of a diad is the a minor arpeggio note equal to or less than the upper note of the diad." An imperative version has been created through a cascade of conditional statements (a dictionary structure would be both more efficient and more characteristic of the Python programming language; however, the following has been preserved in the name of readability):

```
>>> def addNearestArpNote(note):
...     pitch = note.written_pitch
...     pitchClass = pitch.named_diatonic_pitch_class
...     if pitchClass == pitchtools.NamedDiatonicPitchClass('a'):
...         shadowPitch = note.written_pitch - 5
...     elif pitchClass == pitchtools.NamedDiatonicPitchClass('g'):
...         shadowPitch = note.written_pitch - 3
...     elif pitchClass == pitchtools.NamedDiatonicPitchClass('f'):
...         shadowPitch = note.written_pitch - 1
...     elif pitchClass == pitchtools.NamedDiatonicPitchClass('e'):
...         shadowPitch = note.written_pitch - 4
...     elif pitchClass == pitchtools.NamedDiatonicPitchClass('d'):
...         shadowPitch = note.written_pitch - 2
...     elif pitchClass == pitchtools.NamedDiatonicPitchClass('c'):
...         shadowPitch = note.written_pitch - 3
...     elif pitchClass == pitchtools.NamedDiatonicPitchClass('b'):
...         shadowPitch = note.written_pitch - 2
...     return Chord( [note.written_pitch,shadowPitch], Duration(1,8) )
...
```

Figure 2.6: Modeling the pitches: a switch system for choosing arpeggio notes.

The arpeggio selection function is then applied to the contoured descents, resulting in a sequence of descending diads:

```
>>> def addShadow(cd):
...     shadowed = []
...     notLast = cd[:-1]
...     for note in notLast:
...         chord = addNearestArpNote(note)
...         shadowed.append(chord)
...     shadowed = sequencetools.flatten_sequence(shadowed)
...     last = Chord( [cd[-1].written_pitch], cd[-1].duration)
...     shadowed.append(last)
...     return shadowed
...
>>> vln1shadowed = addShadow(vln1cd)
>>> vln2shadowed = addShadow(vln2cd)
>>> vlaChorded = []
>>> for note in vlacd:
...     chord = Chord([note.written_pitch],note.duration)
...     vlaChorded.append(chord)
...
>>> celloShadowed = addShadow(cellocd)
>>> bassShadowed = addShadow(basscd)
```

Figure 2.7: Applying the arpeggio notes to the scalar descents.

Next, the program models the rhythmic behavior of the score. Each part alternates between two durations, and each lower string part doubles the durations of the previous part; this is modeled via recursion, a programming technique in which a function calls itself until a terminal condition is reached:

```
>>> def durateDescent(longDuration, shadowedDescent):
...     outList = []
...     for x in range(len(shadowedDescent)):
...         if x % 2 == 0:
...             chord = Chord(shadowedDescent[x].written_pitches,longDuration)
...             outList.append(chord)
...         else:
...             chord = Chord(shadowedDescent[x].written_pitches,longDuration/2)
...             outList.append(chord)
...     return outList
...
>>> def prolateRecursively(firstDur, multiplier, descentList, outContainer=[],
    listIndex = 0):
...     if listIndex == len(descentList):
...         return outContainer
...     else:
...         durated = durateDescent(firstDur * pow(multiplier,listIndex),
    descentList[listIndex])
...         firstRest = Rest(firstDur * pow(multiplier, listIndex) *1.5)
...         duratedContainer = Container([])
...         duratedContainer.append(firstRest)
...         for event in durated:
...             duratedContainer.append(event)
...         outContainer.append(duratedContainer)
...         return prolateRecursively(firstDur, multiplier, descentList,
    outContainer,listIndex = listIndex + 1)
...
>>> shadowedDescents = [vln1shadowed, vln2shadowed, vlaChorded, celloShadowed,
    bassShadowed]
>>> duratedDescents = prolateRecursively(Duration(1,2), 2, shadowedDescents)
```

Figure 2.8: Recursively generating most of the string parts' rhythms.

Finally, all the durations are split cyclically at intervals of six quarter notes, tying across the newly created splits, in order to yield a series of durations that comports with the score's time signature:

```
>>> for x in range(2):
...     shards = componenttools.split_components_at_offsets(duratedDescents[x].
    leaves, [Duration(6,4)], cyclic=True)
...
```

Figure 2.9: Splitting durations cyclically by the duration of one bar.

While the above functions model most of the pitches and rhythms of the

piece, each part contains an irreducible surplus that must be nonetheless composed out. The next code manually adds the remaining music to each string part:

```
>>> copies = componenttools.copy_components_and_covered_spanners( duratedDescents
    [0][-20:])
>>> duratedDescents[0].extend(copies)
>>> strings[0].append(duratedDescents[0])
>>> vln1finalSustain = Container([])
>>> for x in range(43):
...     vln1finalSustain.append(Note(0, Duration(6,4)))
...
>>> vln1finalSustain.append(Note(0, Duration(2,4)))
>>> tietools.TieSpanner(vln1finalSustain[:])
TieSpanner(c'1., c'1., ... [40] ..., c'1., c'2)
>>> vln1finalSustain.extend([Rest((1,4)), Rest((3,4))])
>>> strings[0].append(vln1finalSustain)
>>>
>>> copies = componenttools.copy_components_and_covered_spanners( duratedDescents
    [1][-15:])
>>> copies = list(copies[:])
>>> copies[-1].written_duration = Duration(1,1)
>>> copies.append(Note(-3,Duration(1,2)))
>>> for note in copies:
...     accent = marktools.articulation = marktools.Articulation('accent')(note)
...     tenuto = marktools.articulation = marktools.Articulation('tenuto')(note)
...
>>> duratedDescents[1].extend(copies)
>>> strings[1].append(duratedDescents[1])
>>> vln2finalSustain = Container([])
>>> for x in range(32):
...     vln2finalSustain.append(Note(-3,Duration(6,4)))
...
>>> vln2finalSustain.append(Note(-3,Duration(1,2)))
>>> tietools.TieSpanner(vln2finalSustain[:])
TieSpanner(a1., a1., ... [29] ..., a1., a2)
>>> tenuto = marktools.articulation = marktools.Articulation('tenuto')(
    vln2finalSustain[0])
>>> accent = marktools.articulation = marktools.Articulation('accent')(
    vln2finalSustain[0])
>>> strings[1].append(vln2finalSustain)
>>> vln2finalSustain.extend([Rest((1,4)), Rest((3,4))])
>>>
>>> for note in duratedDescents[2][-11:]:
...     tenuto = marktools.Articulation('tenuto')(note)
...     accent = marktools.Articulation('accent')(note)
...
```

Figure 2.10: Manual composition of pitches and rhythms after generation.

Because the durations of the viola part cross bars, it is necessary to split the

viola part's durations cyclically at an interval of six quarter notes, to conform to the score's time signature.

```
>>> copies = componenttools.copy_components_and_covered_spanners( duratedDescents
    [2][-11:])
>>> for note in copies:
...         if note.written_duration == Duration(4,4):
...                 note.written_duration = Duration(8,4)
...         else:
...                 note.written_duration = Duration(4,4)
...
>>> duratedDescents[2].extend(copies)
>>> shards = componenttools.split_components_at_offsets(duratedDescents[2].leaves,
     [Duration(6,4)], cyclic=True)
>>> bridge = Note(-8,Duration(1,1))
>>> tenuto = marktools.Articulation('tenuto')(bridge)
>>> accent = marktools.Articulation('accent')(bridge)
>>> duratedDescents[2].append(bridge)
>>> strings[2].append(duratedDescents[2])
>>> violaFinalSustain = Container([])
>>> for x in range(21):
...     violaFinalSustain.append(Note(-8,Duration(6,4)))
...
>>> violaFinalSustain.append(Note(-8,Duration(1,2)))
>>> tietools.TieSpanner(violaFinalSustain[:])
TieSpanner(e1., e1., ... [18] ..., e1., e2)
>>> tenuto = marktools.Articulation('tenuto')(violaFinalSustain[0])
>>> accent = marktools.Articulation('accent')(violaFinalSustain[0])
>>> strings[2].append(violaFinalSustain)
>>> violaFinalSustain.extend([Rest((1,4)), Rest((3,4))])
```

Figure 2.11: Splitting and finishing the viola part.

Likewise, the cello and contrabass parts must be split cyclically by the duration of the meter:

```
>>> duratedDescents[3][-1].written_pitches = [-15,-20]
>>> copies = componenttools.copy_components_and_covered_spanners( duratedDescents
    [3][-8:] )
>>> for chord in copies[1:]:
...     chord.written_pitches = [chord.written_pitches[1]]
...
>>> for chord in copies:
...     tenuto = marktools.Articulation('tenuto')(chord)
...     accent = marktools.Articulation('accent')(chord)
...
>>> duratedDescents[3].extend(copies)
>>> shards = componenttools.split_components_at_offsets(duratedDescents[3].leaves,
     [Duration(6,4)], cyclic=True)
>>> strings[3].extend(duratedDescents[3])
>>> celloCodetta = Container("a,1. ~ a,2 b,1 ~ b,1. ~ b,1. a,1. ~ a,1. ~ a,1. ~ a
    ,1. ~ a,1. ~ a,2 r4 r2.")
>>> strings[3].append(celloCodetta)
>>>
>>> duratedDescents[4].pop(-1)
Chord('<c>\\maxima')
>>> duratedDescents[4].pop(-1)
Chord('<c d>\\longa')
>>> duratedDescents[4].pop(-1)
Chord('<c e>\\maxima')
>>> cbFinalChords = [Chord([-8,-20], Duration(32,4)), Chord([-10,-22], Duration
    (16,4)), Chord([-12,-24], Duration(32,4)), Chord([-13], Duration(16,4)), Chord
    ([-15], Duration(32,4))]
>>> duratedDescents[4].extend(cbFinalChords)
>>> shards = componenttools.split_components_at_offsets(duratedDescents[4].leaves,
     [Duration(6,4)], cyclic=True)
>>> duratedDescents[4].extend([Rest(Duration(1,4)), Rest(Duration(3,4))])
>>> strings[4].append(duratedDescents[4])
```

Figure 2.12: The cello and contrabass pitches and rhythms composed to completion.

The program now models the entire score's pitches and rhythms; the rest of the code models dynamic and technical indications, as well as document formatting. Because it's useful at this point to be able to refer to the score via measure indexes, the program adds measure objects to the previously generated music (note that the previous code generates the entire composition's pitches and rhythms nonmensurally):

```
>>> bellBars = []
>>> shards = componenttools.split_components_at_offsets(score[0].leaves, [Duration
    (6,4)], cyclic=True)
>>> for shard in shards:
...     bellBars.append(Measure((6,4),shard))
...
>>> stringPartBars = []
>>> for staff in strings:
...     theBars = []
...     for split in componenttools.split_components_at_offsets(staff.leaves, [
    Duration(6,4)], cyclic=True):
...         theBars.append( Measure((6,4),split))
...     stringPartBars.append(theBars)
...
```

Figure 2.13: Placing previously generated pitches and rhythms into measures.

Now that the music can be accessed via measure numbers, the program adds dynamic markings to all the parts, as indicated in the score (without apparent pattern):

```
>>> mark = contexttools.DynamicMark('ppp')(bellBars[0][1])
>>> mark = contexttools.DynamicMark('pp')(bellBars[8][1])
>>> mark = contexttools.DynamicMark('p')(bellBars[18][1])
>>> mark = contexttools.DynamicMark('mp')(bellBars[26][1])
>>> mark = contexttools.DynamicMark('mf')(bellBars[34][1])
>>> mark = contexttools.DynamicMark('f')(bellBars[42][1])
>>> mark = contexttools.DynamicMark('ff')(bellBars[52][1])
>>> mark = contexttools.DynamicMark('fff')(bellBars[60][1])
>>> mark = contexttools.DynamicMark('ff')(bellBars[68][1])
>>> mark = contexttools.DynamicMark('f')(bellBars[76][1])
>>> mark = contexttools.DynamicMark('mf')(bellBars[84][1])
>>> mark = contexttools.DynamicMark('pp')(bellBars[-1][0])
>>>
>>> mark = contexttools.DynamicMark('ppp')(stringPartBars[0][7][1])
>>> mark = contexttools.DynamicMark('pp')(stringPartBars[0][15][0])
>>> mark = contexttools.DynamicMark('p')(stringPartBars[0][22][3])
>>> mark = contexttools.DynamicMark('mp')(stringPartBars[0][32][0])
>>> mark = contexttools.DynamicMark('mf')(stringPartBars[0][39][3])
>>> mark = contexttools.DynamicMark('f')(stringPartBars[0][47][0])
>>> mark = contexttools.DynamicMark('ff')(stringPartBars[0][56][0])
>>> mark = contexttools.DynamicMark('fff')(stringPartBars[0][62][2])
>>>
>>> mark = contexttools.DynamicMark('pp')(stringPartBars[1][7][0])
>>> mark = contexttools.DynamicMark('p')(stringPartBars[1][12][0])
>>> mark = contexttools.DynamicMark('p')(stringPartBars[1][13][0])
>>> mark = contexttools.DynamicMark('mp')(stringPartBars[1][25][0])
>>> mark = contexttools.DynamicMark('mf')(stringPartBars[1][34][1])
>>> mark = contexttools.DynamicMark('f')(stringPartBars[1][44][1])
>>> mark = contexttools.DynamicMark('ff')(stringPartBars[1][54][1])
>>> mark = contexttools.DynamicMark('fff')(stringPartBars[1][62][1])
>>>
>>> mark = contexttools.DynamicMark('p')(stringPartBars[2][8][0])
>>> mark = contexttools.DynamicMark('mp')(stringPartBars[2][19][1])
>>> mark = contexttools.DynamicMark('mf')(stringPartBars[2][30][0])
>>> mark = contexttools.DynamicMark('f')(stringPartBars[2][36][0])
>>> mark = contexttools.DynamicMark('ff')(stringPartBars[2][52][0])
>>> mark = contexttools.DynamicMark('fff')(stringPartBars[2][62][0])
>>>
>>> mark = contexttools.DynamicMark('p')(stringPartBars[3][10][0])
>>> mark = contexttools.DynamicMark('mp')(stringPartBars[3][21][0])
>>> mark = contexttools.DynamicMark('mf')(stringPartBars[3][31][0])
>>> mark = contexttools.DynamicMark('f')(stringPartBars[3][43][0])
>>> mark = contexttools.DynamicMark('ff')(stringPartBars[3][52][0])
>>> mark = contexttools.DynamicMark('fff')(stringPartBars[3][62][0])
```

Figure 2.14: Adding dynamic markings to parts via measure indexes.

Next, the program adds technical and expressive markings to the music, again via measure indexes:

```
>>> markup = markuptools.Markup(”\\left-column {div. \\line {con sord.} }”, Up)(
    stringPartBars[0][6][1])
>>> markup = markuptools.Markup('sim.', Up)(stringPartBars[0][8][0])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[0][58][3])
>>> markup = markuptools.Markup('div.', Up)(stringPartBars[0][59][0])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[0][63][3])
>>>
>>> markup = markuptools.Markup('div.', Up)(stringPartBars[1][7][0])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[1][66][1])
>>> markup = markuptools.Markup('div.', Up)(stringPartBars[1][67][0])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[1][74][0])
>>>
>>> markup = markuptools.Markup('sole', Up)(stringPartBars[2][8][0])
>>>
>>> markup = markuptools.Markup('div.', Up)(stringPartBars[3][10][0])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[3][74][0])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[3][84][1])
>>> markup = markuptools.Markup('\\italic {espr.}', Down)(stringPartBars
    [3][86][0])
>>> markup = markuptools.Markup('\\italic {molto espr.}', Down)(stringPartBars
    [3][88][1])
>>>
>>> markup = markuptools.Markup('div.', Up)(stringPartBars[4][14][0])
>>> markup = markuptools.Markup('\\italic {espr.}', Down)(stringPartBars
    [4][86][0])
>>> shards = componenttools.split_components_at_offsets(stringPartBars[4][88][:],
    [Duration(1,1), Duration(1,2)])
>>> markup = markuptools.Markup('\\italic {molto espr.}', Down)(stringPartBars
    [4][88][1])
>>> markup = markuptools.Markup('uniti', Up)(stringPartBars[4][99][1])
```

Figure 2.15: Adding technical and expressive markings to parts via measure indexes.

Scores might contain markings beyond those in Lilypond's model of common notation; many of them can be created by combining existing markings in LilyPond's lexicon. In this score, a successive upbow-downbow marking must be created and applied, by combining the upbow and downbow markings into a single command:

```
>>> def rebow(component):
...     markup = markuptools.Markup('\\concat { \\musicglyph #\"scripts.downbow\"
    \\hspace #1 \\musicglyph #\"scripts.upbow\" }', Up)(component)
...
>>> rebow(stringPartBars[0][64][0])
>>> rebow(stringPartBars[1][75][0])
>>> rebow(stringPartBars[2][86][0])
```

Figure 2.16: Defining and using a custom technical marking.

Next, a list of measure numbers and a loop add rehearsal markings to the score:

```
>>> rehearsalBars = [7,13,19,25,31,37,43,49,55,61,67,73,79,85,91,97,103]
>>> #use a loop to place the marks. attaching them to the top staff is fine,
... for bar in rehearsalBars:
...         marktools.LilyPondCommandMark("mark \\default")(bellBars[bar-1][0])
...
LilyPondCommandMark('mark \\default')(r1.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r1.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r1.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r1.)
LilyPondCommandMark('mark \\default')(r2.)
LilyPondCommandMark('mark \\default')(r1.)
LilyPondCommandMark('mark \\default')(r1.)
LilyPondCommandMark('mark \\default')(r1.)
```

Figure 2.17: Adding rehearsal marks.

The remainder of the program handles page layout and the formatting of the entire score, adjusting the space = time relationship that governs horizontal placement of typographical elements, the spacing and thickness of score elements, the size of the page, the composer, and the title; document layout is also object-oriented, and the code becomes a series of lines setting the attribute values of a LilyPondFile and

a Score object:

```
>>> score.set.proportional_notation_duration = schemetools.SchemeMoment(1, 8)
>>> score.override.system_start_bar.thickness = 15
>>> score.override.system_start_square.padding = 3
>>> score.override.system_start_square.thickness = 5
>>> score.override.system_start_bracket.padding = 2.5
>>> score.override.rehearsal_mark.padding = 1.3
>>> score.override.rehearsal_mark.font_name = ”Futura”
>>> score.override.script.padding = 0.9
>>>
>>> lily = lilypondfiletools.make_basic_lilypond_file(score)
>>> lily.global_staff_size = 16
>>> lily.layout_block.ragged_right = False
>>> lily.paper_block.markup_system_spacing__basic_distance = 0
>>> lily.paper_block.markup_system_spacing__basic_distance = 0
>>> lily.paper_block.bottom_margin = 1 * 25.4
>>> lily.paper_block.top_margin = 1 * 25.4
>>> lily.paper_block.left_margin = 1.25 * 25.4
>>> lily.paper_block.right_margin = 1.5 * 25.4
>>> lily.paper_block.paper_width = 11 * 25.4
>>> lily.paper_block.paper_height = 17 * 25.4
>>> lily.header_block.composer = markuptools.Markup( fonted(’Arvo Pärt’) )
>>> lily.header_block.title = markuptools.Markup( fonted(’Cantus in Memory of
    Benjamin Britten (1980)’) )
```

Figure 2.18: Document layout and formatting.

Finally, the program renders a .pdf file (B.1).

2.2.2 *Windungen* (1976) by Iannis Xenakis

The Composition

Iannis Xenakis's *Windungen* for twelve cellos was commissioned by the cello section of the Berlin Philharmonic in 1976. The composition of the work has not been carefully documented; however, Xenakis biographer James Harley notes that Xenakis composed the work while exploring basic principles of the branch of mathematics known as group theory (Harley 2004, p. 90), the simplest applications of which pertain to basic combinatoric groups of permutations and combinations (W. Magnus, Karrass, and Solitar 2004). Xenakis's electroacoustic experiments with random walks influenced his acoustic music during the 1970s, and pieces from 1973—

1984 made use of Brownian motion and random walks to generate pitch material (Solomos 2001). In the author's process of modeling, it became clear that the metaphor of a wave reflecting off a surface — a central metaphor of Xenakis's stochastic synthesis algorithms (Luque 2009) — governed the spatial patterns with which the sounds rotate through the ensemble.

The Approach

Unlike the Pärt, no large-scale formal hypothesis for the composition existed prior to the creation of a rendering program. For this reason, the process of reconstruction became an iterative process of successive approximation, section by section through the score, in which code proposed a model of composition and resulting notation assessed the possibility that the original mode of composition could have resembled the generating model. When the resulting notation deviated from the published score, the code was revised to generate results more like those in the published score — acknowledging that techniques of constrained randomness may potentially generate substantially different results from the same generative principles. Given the documented historic context of the work, this process was undertaken in the presence of possible applications of simple set-based selection and random walks, which proved fruitful: along the way, it was discovered that events seem to happen in exclusive arrangements, i.e., each of the twelve players enters at a different time in the measure, although they all enter in the same measure, or each of the twelve player plays a different pitch in a certain set of pitches. Despite these general guidelines, the relationship between the published score and its encoded reconstruction was substantially noisier and less determined than that in the case of the Pärt, due to the presupposed use of stochastic techniques.

The Code

The composition begins with diatonic material — the first three scale degrees of an A major scale — rotated throughout the ensemble; on the page, this seems to be a graphic rotation through the staffs of the score. First, the program models the rotation of materials through the score's twelve staffs:

```
>>> def make_base_list_of_compressed_rotation_tuples(staffIndexBoundsTuple,
    rotationBandwidth):
...     lowerBound = staffIndexBoundsTuple[0]
...     upperBound = staffIndexBoundsTuple[1]
...     bitList = range(lowerBound - 1, upperBound + 1)
...     rotations = [ bitList[x:x+rotationBandwidth] for x in range(0, len(bitList
    ) - rotationBandwidth + 1) ]
...     del(rotations[0][0])
...     del(rotations[-1][-1])
...     return rotations
...
>>> def make_base_list_of_uncompressed_rotation_tuples(staffIndexBoundsTuple,
    rotationBandwidth):
...     lowerBound = staffIndexBoundsTuple[0]
...     upperBound = staffIndexBoundsTuple[1]
...     bitList = range(lowerBound, upperBound)
...     rotations = [ bitList[x:x+rotationBandwidth] for x in range( len(bitList)
    - rotationBandwidth + 1) ]
...     return rotations
...
>>> def make_base_list_of_rotation_tuples(staffIndexBoundsTuple, rotationBandwidth
    , compressedReflections):
...     if compressedReflections == True:
...         rotations = make_base_list_of_compressed_rotation_tuples(
    staffIndexBoundsTuple, rotationBandwidth)
...     else:
...         make_base_list_of_uncompressed_rotation_tuples(staffIndexBoundsTuple,
    rotationBandwidth)
...     return rotations
...
>>> def mirror_base_list_of_rotation_tuples(rotations):
...     copied = rotations[1:-1]
...     copied.reverse()
...     back = copied
...     rotations.extend( back )
...     return rotations
...
>>> def make_mirorred_base_list_of_rotation_tuples(staffIndexBoundsTuple,
    rotationBandwidth, compressedReflections):
...     rotations = make_base_list_of_rotation_tuples(staffIndexBoundsTuple,
    rotationBandwidth, compressedReflections)
...     rotations = mirror_base_list_of_rotation_tuples(rotations)
...     return rotations
...
```

Figure 2.19: Modeling the rotation of material through the score.

With these functions, it is possible to create a single function that makes a cyclic matrix, specifying which staffs the rotating music should be written on at any given moment in given rotation:

```
>>> def make_cyclic_matrix_for_rotation_by_bandwidth(staffIndexBoundsTuple,
    rotationBandwidth, compressedReflections = True):
...     rotations = make_mirorred_base_list_of_rotation_tuples(
    staffIndexBoundsTuple, rotationBandwidth, compressedReflections)
...     matrix = sequencetools.CyclicMatrix(rotations)
...     return matrix
...
```

Figure 2.20: A function that returns a matrix of cyclic tuples to specify which staffs the rotating music should be written on.

One quirky behavior of the way that Xenakis rotates material throughout the ensemble is that the endpoints of the rotation are often solos, as though the rotating material compresses and expands as it rotates, compressing into the part of a single instrument at the endpoints of rotation:

```
>>> def goingUp(staffIndexes):
...     if staffIndexes[0][0] == staffIndexes[1][0]:
...         return False
...     else:
...         return True
...
>>> def add_bookends_to_staff_indexes(staffIndexes):
...     if goingUp(staffIndexes) == True:
...         frontBookend = [ staffIndexes[0][1] ]
...         rearBookend = [ staffIndexes[-1][0] ]
...     else:
...         frontBookend = [ staffIndexes[0][0] ]
...         rearBookend = [ staffIndexes[-1][1] ]
...     staffIndexes.insert(0, frontBookend)
...     staffIndexes.append( rearBookend )
...     return staffIndexes
...
```

Figure 2.21: Modeling Xenakis's bookended rotations.

With a model of rotation in place, the music can be modeled as a single staff, to be rotated throughout the ensemble; the single staff can be divided, and portions of

it copied to the staffs specified by the two-dimensional staff index matrix described above:

```
>>> def pair_tuples_with_splits_using_pitches_from_expr(splits, staffIndexes):
...     splitTuplePairs = [ ]
...     for x in range( len(splits) ):
...         pair = staffIndexes[x], splits[x]
...         splitTuplePairs.append(pair)
...     return splitTuplePairs
...
>>> def pair_pitches_with_splits(matrix, splits, phaseOffset, soloBookends = False
    ):
...     if soloBookends == True:
...         staffIndexes = [matrix[x + phaseOffset] for x in range( len(splits) -
    2 ) ]
...         staffIndexes = add_bookends_to_staff_indexes(staffIndexes)
...     else:
...         staffIndexes = [matrix[x + phaseOffset] for x in range( len(splits) )
    ]
...     splitTuplePairs = pair_tuples_with_splits_using_pitches_from_expr(splits,
    staffIndexes)
...     return splitTuplePairs
...
>>> def repitch_copy(copied, pitch):
...     for note in iterationtools.iterate_notes_in_expr(copied):
...         note.written_pitch = pitch
...     return copied
...
>>> def add_split_to_score_by_tuple(split, score, staffTuple):
...     allStaffs = set(range( len(score) ) )
...     soundStaffs = set(staffTuple)
...     silenceStaffs = allStaffs - soundStaffs
...     for x in soundStaffs:
...         copied = componenttools.copy_components_and_covered_spanners(split)
...         score[x].extend( copied )
...     for x in silenceStaffs:
...         duration = sum( [y.written_duration for y in split] )
...         duration = Duration( duration )
...         leaves = leaftools.make_tied_leaf(Rest, duration)
...         score[x].extend( leaves )
...
```

Figure 2.22: From single staff to rotating staffs.

Sometimes the pitch material varies within the rotation; for example, each of the three playing cellos might play a different pitch, rather than all three playing the

same pitch. It becomes necessary to select different pitches for the different cellos performing at any given moment. Because the section of the score has already been modeled as a distributed single staff, another stage is added to the process of rotation, in which the single staff may be dynamically repitched when it is copied to multiple staves:

```
>>> def add_repitched_split_to_score_by_tuple(split, score, staffIndexTuple,
    pitchGroup):
...     shuffle(pitchGroup)
...     cyclicPitchTuple = sequencetools.CyclicTuple(pitchGroup)
...         #pair the pitch with each staff.
...     staffPitchPairs = zip(staffIndexTuple,cyclicPitchTuple)
...         #(3, 1), (4,-3), (5, 1)
...     #use set difference to get two lists: which staffs have music, and which
    silence.
...     allStaffs = set(range( len(score) ) )
...     soundStaffs = set(staffIndexTuple)
...     silenceStaffs = allStaffs - soundStaffs
...     for pair in staffPitchPairs:
...         staffIndex = pair[0]
...         pitch = pair[1]
...         copied = componenttools.copy_components_and_covered_spanners(split)
...         copied = repitch_copy(copied, pitch)
...         score[ staffIndex ].extend( copied )
...     for x in silenceStaffs:
...         duration = sum( [y.written_duration for y in split] )
...         duration = Duration( duration )
...         leaves = leaftools.make_tied_leaf(Rest, duration)
...         score[x].extend( leaves )
...
>>> def add_splits_to_score_by_tuples(score, splitTuplePairs, pitchGroup):
...     for pair in splitTuplePairs:
...         staffTuple = pair[0]
...         split = pair[1]
...         if pitchGroup == [ ]:
...             add_split_to_score_by_tuple(split, score, staffTuple)
...         else:
...             add_repitched_split_to_score_by_tuple(split, score, staffTuple,
    pitchGroup)
...
```

Figure 2.23: Repitching the staff as it rotates.

Because the rotation occurred at a resolution of a sixteenth note, rests must be fused in order to create durationally appropriate rests (rather than successions of

many sixteenth-note rests.) Beam spanners are applied to beats containing notes:

```
>>> def fuse_rests_in_beat(beat):
...     for group in componenttools.yield_topmost_components_grouped_by_type(beat)
    :
...         if isinstance(group[0], Rest):
...             leaftools.fuse_leaves( group[:] )
...
>>> def fuse_rests_in_staff_by_beats(beats):
...     for beat in beats:
...         fuse_rests_in_beat(beat)
...
>>> def apply_beam_spanner_to_non_rest_beat(beat, brokenBeam = False):
...     if not all( [isinstance(x,Rest) for x in beat] ):
...         beamtools.BeamSpanner(beat[:],Up)
...         if len(beat) == 4 and brokenBeam == True:
...             right = marktools.LilyPondCommandMark("set stemRightBeamCount =
    #1")(beat[1])
...             left = marktools.LilyPondCommandMark("set stemLeftBeamCount = #1")
    (beat[2])
...
>>> def apply_beam_spanner_to_non_rest_beats(beats, brokenBeam = False):
...     for beat in beats:
...         apply_beam_spanner_to_non_rest_beat(beat, brokenBeam = brokenBeam)
...
```

Figure 2.24: Modeling the low-level typographical habits in the rotation section.

All of this code so far has modeled high-level behaviors of material without considering the selection of pitch. The ability to choose pitch from a reservoir of possible pitches, with and without repetition, is a central element of the music; here a function makes a random choice from a set and the iterative use of this function in a second function creates a staff of randomly chosen pitches:

```
>>> def choose_pitch_without_repetition(pitch, choices):
...     chosen = pitch
...     while chosen == pitch:
...         candidate = choice(choices)
...         if candidate != pitch:
...             chosen = candidate
...     return chosen
...
>>> def make_staff_with_random_pitches(choices, numPitches, brokenBeam = False):
...     notes = [ ]
...     pitchList = choices
...     chosen = choices[-1]
...     for x in range(numPitches):
...         pitch = choose_pitch_without_repetition(chosen, choices)
...         chosen = pitch
...         note = Note(pitch, Duration(1,16) )
...         notes.append( note )
...     staff = Staff(notes)
...     #if brokenBeam == True:
...     #   beats = componenttools.split_components_at_offsets(staff.leaves, [
    Duration(1,4)], cyclic=True, tie_split_notes=False)
...     #   apply_beam_spanner_to_non_rest_beats(beats, brokenBeam = False)
...     return staff
...
```

Figure 2.25: Function for creating a staff of random pitches.

Finally, another set of typographical functions fuses rests and makes uniform the position of short rests on each staff:

```
>>> def beam_and_fuse_beats_in_score_by_durations(score, durations, cyclic=False,
    brokenBeam = False):
...     for staff in score:
...         beats = componenttools.split_components_at_offsets(staff.leaves,
    durations, cyclic=cyclic)
...         fuse_rests_in_staff_by_beats(beats)
...         beats = componenttools.split_components_at_offsets(staff.leaves,
    durations, cyclic=cyclic)
...         apply_beam_spanner_to_non_rest_beats(beats, brokenBeam = brokenBeam)
...
>>> def fuse_consecutive_rests_of_duration_by_duration_threshold(run, duration,
    durationThreshold):
...     toFuse = [x for x in run if x.written_duration == duration]
...     runDuration = componenttools.sum_duration_of_components(run[:])
...     if durationThreshold <= runDuration:
...         leaftools.fuse_leaves(toFuse)
...
>>> def fuse_rests_of_duration_in_bar_by_duration_threshold(bar, duration,
    durationThreshold):
...     for run in componenttools.yield_topmost_components_grouped_by_type(bar):
...         fuse_consecutive_rests_of_duration_by_duration_threshold(run, duration
    , durationThreshold )
...
>>> def fuse_rests_of_duration_in_bars_by_duration_threshold(bars, duration,
    durationThreshold):
...     for bar in bars:
...         fuse_rests_of_duration_in_bar_by_duration_threshold(bar, duration,
    durationThreshold )
...
>>> def fuse_rests_of_duration_in_score_by_duration_threshold(score, duration,
    durationThreshold):
...     for staff in score:
...         bars = componenttools.partition_components_by_durations_exactly(staff.
    leaves, [Duration(4,4)], cyclic=True)
...         fuse_rests_of_duration_in_bars_by_duration_threshold(bars, duration,
    durationThreshold)
...
>>> def set_vertical_positioning_pitch_on_rests_in_staff(staff, pitch):
...     for rest in iterationtools.iterate_rests_in_expr(staff):
...         resttools.set_vertical_positioning_pitch_on_rest(rest, pitch)
...
```

Figure 2.26: The final stage of typographical adjustment for the rotation section.

All of the previously defined functions work together in a function that rotates an arbitrary staff of music around a score of x staffs, placing the music on y staffs

at any given moment, with keyword arguments that allow bookending, repitching according to a given pitch set, and an arbitrary phase offset to start midway through rotations:

```
>>> def rotate_expression_through_adjacent_staffs_at_bandwidth_by_durations(
    expression, score, staffIndexBoundsTuple, rotationBandwidth, durations,
    compressedReflections=True, cyclic=False, phaseOffset= 0, soloBookends = False
    , brokenBeam = False, pitchGroup = [ ]):
...     matrix = make_cyclic_matrix_for_rotation_by_bandwidth(
    staffIndexBoundsTuple, rotationBandwidth, compressedReflections )
...     splits = componenttools.split_components_at_offsets(expression.leaves,
    durations, cyclic=cyclic, tie_split_notes=False)
...     splitTuplePairs = pair_pitches_with_splits(matrix, splits, phaseOffset,
    soloBookends)
...     add_splits_to_score_by_tuples(score, splitTuplePairs, pitchGroup)
...     beam_and_fuse_beats_in_score_by_durations(score, [Duration(1,4)],cyclic=
    True, brokenBeam = brokenBeam)
...
```

Figure 2.27: The final rotation function.

Utility functions enable this rotation function to put material onto the score:

```
>>> def make_empty_cello_score(numStaffs):
...     score = Score([])
...     for x in range(numStaffs):
...         score.append( Staff([]) )
...         contexttools.ClefMark('bass')(score[x])
...         score[x].override.beam.damping = "+inf.0"
...     return score
...
>>> def add_expression_to_staffs_in_score_by_index_tuple(expr, score, indexTuple,
    brokenBeam = False):
...     for x in range(indexTuple[0], indexTuple[1]):
...         copies = componenttools.copy_components_and_covered_spanners(expr[:])
...         beats = componenttools.split_components_at_offsets(copies, [Duration
    (1,4)], cyclic = True)
...         apply_beam_spanner_to_non_rest_beats(beats, brokenBeam = brokenBeam)
...         score[x].extend( componenttools.copy_components_and_covered_spanners(
    expr[:]) )
...
```

Figure 2.28: Utility functions enable rotation.

At this point, it becomes straightforward to model the succession of events

in the score, using the previous functions. First, the program models the first section of music, in which a unison figure rotates throughout the ensemble:

```
>>> def make_rotating_unison_section(score):
...     unison = make_staff_with_random_pitches([-3, -1, 1], 24)
...     add_expression_to_staffs_in_score_by_index_tuple(unison[:-8], score,
    (0,12) )
...     add_expression_to_staffs_in_score_by_index_tuple(unison[-8:], score, (0,1)
     )
...     for staff in score[1:]:
...         staff.append( Rest("r2") )
...     firstRotationPitches = make_staff_with_random_pitches([-3, -1, 1], 64)
...     rotate_expression_through_adjacent_staffs_at_bandwidth_by_durations(
    firstRotationPitches, score, (0,6), 3, [Duration(1,16)], compressedReflections
    =True, cyclic=True, brokenBeam = True)
...     for staff in score:
...         staff.extend("b2 ~ b1 ~ b ~ b2.")
...     #m9b4
...     secondRotationLowerPitches = make_staff_with_random_pitches([-3, -1, 1],
    (13*4) + 6  )
...     rotate_expression_through_adjacent_staffs_at_bandwidth_by_durations(
    secondRotationLowerPitches, score, (6,12), 3, [Duration(1,16)],
    compressedReflections=True, cyclic=True, phaseOffset = 5, soloBookends = True,
     brokenBeam = True)
...     #m13b3
...     for x in range(6):
...         del(score[x][-3:])
...         score[x].append( "r4" )
...     score[5].pop(-1)
...     score[5].append("r8. b16")
...     secondRotationHigherPitches = make_staff_with_random_pitches([-3, -1, 1],
    56  )
...     rotate_expression_through_adjacent_staffs_at_bandwidth_by_durations(
    secondRotationHigherPitches, score, (0,6), 3, [Duration(1,16)],
    compressedReflections=True, cyclic=True, phaseOffset = 5, brokenBeam = True)
...     for x in range(6,12):
...         del( score[x][-2:] )
...         score[x].append( "r2" )
...         score[x].append( "r16" * 14 )
...     #m16b3
...     add_expression_to_staffs_in_score_by_index_tuple(unison, score, (0,6),
    brokenBeam = True )
...
```

Figure 2.29: Modeling the score's first rotation with the rotation function.

In the next section, the rotation continues with each moment sounding all three pitches; the rotation function still applies, this time with an optional pitch set argument:

```
>>> def make_rotating_pitch_group_section(score):
...     thirdRotationLowerPitches = make_staff_with_random_pitches([-3, -1, 1], 38
      )
...     rotate_expression_through_adjacent_staffs_at_bandwidth_by_durations(
    thirdRotationLowerPitches, score, (6,12), 3, [Duration(1,16)],
    compressedReflections=True, cyclic=True, phaseOffset = 0, soloBookends = True,
     brokenBeam = False, pitchGroup = [ -3, -1, 1 ] )
...     for x in range(0,6):
...         del(score[x][-2:])
...         score[x].append( ”r16” )
...     thirdRotationUpperPitches = make_staff_with_random_pitches([-3, -1, 1], 55
      )
...     rotate_expression_through_adjacent_staffs_at_bandwidth_by_durations(
    thirdRotationUpperPitches, score, (0,6), 3, [Duration(1,16)],
    compressedReflections=True, cyclic=True, phaseOffset = 5, soloBookends = True,
     brokenBeam = False, pitchGroup = [ -3, -1, 1 ] )
...     for x in range(6,12):
...         del(score[x][-1])
...
```

Figure 2.30: Modeling a rotation with multiple simultaneous pitches.

Next, the entire ensemble plays a tutti unison; because no rotation occurs, the random selection functions alone model this section of the score:

```
>>> def generate_random_sixteenth_note():
...     pitches = [ -3, -1, 1]
...     chosen = choice(pitches)
...     return Note( chosen, (1,16) )
...
>>> def generate_n_random_sixteenth_notes(n):
...     notes = [ ]
...     for x in range(n):
...         note = generate_random_sixteenth_note()
...         notes.append( note )
...     return notes
...
>>> def add_n_random_sixteenth_notes_to_staff(n, staff):
...     notes = generate_n_random_sixteenth_notes(n)
...     staff.extend( notes )
...
>>> def add_n_sixteenth_rests_to_staff(n, staff):
...     for x in range(n):
...         staff.extend( ”r16” )
...
>>> def add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(n, score,
    staffTuple):
...     staffIndexes = range(staffTuple[0], staffTuple[1])
...     staffSet = set(staffIndexes)
...     allStaffs = set( range(len(score)) )
...     silentSet = allStaffs - staffSet
...     for staffIndex in staffIndexes:
...         add_n_random_sixteenth_notes_to_staff(n, score[ staffIndex ] )
...     for staffIndex in silentSet:
...         add_n_sixteenth_rests_to_staff(n, score[ staffIndex ])
...
```

Figure 2.31: Modeling the tutti section with randomly selected sixteenth notes.

Using these functions, the tutti section can be encapsulated into a single function:

The first “diatonic” section of the score, comprised of the first twenty-nine measures of music, may now be encapsulated in a single function:

```
>>> def make_tutti_pitch_group_section(score):
...     rightSide = (0,6)
...     leftSide = (6,12)
...     tutti = (0, 12)
...     add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(40, score,
    tutti )
...     add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(6, score,
    rightSide )
...     add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(5, score,
    leftSide )
...     add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(4, score,
    rightSide )
...     add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(6, score,
    leftSide )
...     add_n_random_sixteenth_notes_to_staffs_in_score_by_index_tuple(31, score,
    tutti )
...     beam_and_fuse_beats_in_score_by_durations(score, [Duration(1,4)],cyclic=
    True)
...
```

Figure 2.32: The tutti section as single function.

```
>>> def make_diatonic_section(score):
...     make_rotating_unison_section(score)
...     make_rotating_pitch_group_section(score)
...     make_tutti_pitch_group_section(score)
...
```

Figure 2.33: The first section of the score as a single encapsulation.

The next section of the music appears to be based on a random walk. It has been modeled using a weighted choice from a list of possible intervals; the probability weights were adjusted in a process of trial and error to approximate the published result:

```
>>> def make_weighted_choice_from_list( theList ): #makes a weighted choice (by
    Kevin Parks at snippets.dzone.com)
...     n = uniform(0, 1)
...     for item, weight in theList:
...         if n < weight:
...             break
...         n = n - weight
...     return item
...
>>> def choose_interval_change_from_weights(pitchWeights):
...     interval = make_weighted_choice_from_list(pitchWeights)
...     octave = make_weighted_choice_from_list( [(0, .95), (1, .05)] )
...     if octave:
...         interval += 8
...     down = make_weighted_choice_from_list( [(0, .5), (1, .5)] )
...     if down:
...         interval = interval * -1
...     return interval
...
>>> def choose_pitch_based_on_previous_pitch(previous, changeWeights, pitchWeights
    ):
...     change = make_weighted_choice_from_list( changeWeights )
...     if change:
...         chosen = choose_interval_change_from_weights(pitchWeights)
...         return chosen
...     else:
...         return previous
...
```

Figure 2.34: Weighted probability choice functions for the first random walk section.

Because the random pitch walk might cause a given part to wander lower or higher than the perceptual boundaries of its initial clef's ledger lines, the program reviews the pitches resulting from the selected intervals to automate clef switching:

```
>>> def make_from_to_interval_tuples_from_expr(expr):
...     tuples = [ ]
...     for x in range( len(expr) -1):
...         fromPitch = expr[x].written_pitch
...         toPitch = expr[x+1].written_pitch
...         interval = pitchtools.calculate_melodic_chromatic_interval(fromPitch,
    toPitch)
...         tuple = (fromPitch.chromatic_pitch_number, toPitch.
    chromatic_pitch_number, interval)
...         tuples.append( tuple )
...     return tuples
...
>>> def check_for_change_to_treble(tuple):
...     fromPitch = tuple[0]
...     toPitch = tuple[1]
...     interval = tuple[2]
...     if interval.number >= 5 and toPitch >= 5:
...         return 'treble'
...     else:
...         return False
...
>>> def check_for_change_to_bass(tuple):
...     fromPitch = tuple[0]
...     toPitch = tuple[1]
...     interval = tuple[2]
...     if interval.number <= -5 and -5 >= toPitch:
...         return 'bass'
...     else:
...         return False
...
>>> def check_to_from_interval_tuple_for_clef_add(x, tuple, staff):
...     effectiveClef = contexttools.get_effective_clef(staff[x])
...     if effectiveClef.clef_name == 'bass':
...         clef = check_for_change_to_treble(tuple)
...     else:
...         clef = check_for_change_to_bass(tuple)
...     return clef
...
```

Figure 2.35: Conditional checks to determine a clef change.

The above functions can then be applied to an expression to add clef changes:

```
>>> def add_clef_if_needed(x, tuple, staff):
...     clef = check_to_from_interval_tuple_for_clef_add(x, tuple, staff)
...     if clef:
...         contexttools.ClefMark(clef)( staff[x+1] )
...
>>> def add_clefs_to_expr(expr):
...     differenceTuples = make_from_to_interval_tuples_from_expr(expr)
...     for x,tuple in enumerate(differenceTuples):
...         add_clef_if_needed(x, tuple, expr)
...
>>> def add_clefs_to_exprs(staffs):
...     for staff in staffs:
...         add_clefs_to_expr(staff)
...
```

Figure 2.36: Applying automatic clef changes to an expression.

Then the program adds random walk notes to the staffs in the score:

```
>>> def choose_pitch_values_from_weights(seedPitch, numNotes, changeWeights,
    pitchWeights):
...     pitches = [ ]
...     previous = seedPitch
...     for x in range(numNotes):
...         chosen = choose_pitch_based_on_previous_pitch(previous, changeWeights,
     pitchWeights)
...         pitches.append( chosen )
...         previous = chosen
...     return pitches
...
>>> def make_n_notes_from_random_pitch_walk(seedPitch, numNotes, changeWeights,
    pitchWeights):
...     pitches = choose_pitch_values_from_weights(seedPitch, numNotes,
    changeWeights, pitchWeights)
...     notes = [ Note( x,(1,16) ) for x in pitches]
...     return notes
...
>>> def add_components_from_staffs_to_score(staffs, score):
...     for x,staff in enumerate(staffs):
...         copied = componenttools.copy_components_and_covered_spanners(staff.
    leaves)
...         score[x].extend(copied)
...     staff = Staff([])
...     contexttools.ClefMark('bass')(staff[0])
...     add_clefs_to_exprs( [staff] )
...     return staff
...
```

Figure 2.37: Adding random walk notes to the score.

The random walk sections are entered and exited via unison drones which may or may not be inflected with trills. The following functions query whether or not a trill spanner has been attached to a given score component and fuse trill spanners attached to adjacent components:

```
>>> def trill_spanner_attached_to_component(component):
...     spanners = spannertools.get_spanners_attached_to_component( component )
...     for x in spanners:
...         if isinstance(x, spannertools.TrillSpanner):
...             return True
...     return False
...
>>> def fuse_trill_spanners_attached_to_components(left, right):
...     leftTrill = spannertools.get_the_only_spanner_attached_to_component(left,
    klass=spannertools.TrillSpanner)
...     rightTrill = spannertools.get_the_only_spanner_attached_to_component(right
    , klass=spannertools.TrillSpanner)
...     leftTrill.fuse(rightTrill)
...
```

Figure 2.38: Querying and fusing trill spanners.

Next, three functions add three respective materials to a specified staff — an initial drone (preceding the random walk), a random walk, or a terminal drone (following the random walk):

```
>>> def add_in_drone_to_staff(staff, inDroneDuration, trill_in, tie_to_previous):
...     startLength = len(staff)
...     endOfBeginningIndex = startLength - 1
...     beginningOfEndIndex = startLength
...     inDrone = leaftools.make_tied_leaf(Note, inDroneDuration, pitches =
    pitchtools.NamedChromaticPitch(”b”))
...     staff.extend( inDrone )
...     if tie_to_previous == True:
...         tietools.apply_tie_spanner_to_leaf_pair( staff[endOfBeginningIndex],
    staff[beginningOfEndIndex] )
...     if trill_in:
...         inTrillSpanner = spannertools.TrillSpanner(inDrone[:])
...         inTrillSpanner.written_pitch = 2
...     if trill_spanner_attached_to_component( staff[endOfBeginningIndex] ):
...         fuse_trill_spanners_attached_to_components( staff[endOfBeginningIndex
    ], staff[beginningOfEndIndex] )
...     if trill_in and not trill_spanner_attached_to_component( staff[
    endOfBeginningIndex] ):
...         marktools.LilyPondCommandMark(”tieDown”)(inDrone[0])
...         marktools.LilyPondCommandMark(”tieUp”,”after”)(inDrone[0])
...
>>> def add_walk_to_staff(staff, walkDuration, changeWeights, pitchWeights):
...     numSixteenths = walkDuration.numerator * (16 / walkDuration.denominator)
...     walkNotes = make_n_notes_from_random_pitch_walk(-1, numSixteenths,
    changeWeights, pitchWeights)
...     if walkNotes[0].written_pitch < pitchtools.NamedChromaticPitch(”c’”):
...         contexttools.ClefMark(’bass’)(walkNotes[0])
...     else:
...         contexttools.ClefMark(’treble’)(walkNotes[0])
...     intermediateStaff = Staff(walkNotes)
...     add_clefs_to_expr(intermediateStaff)
...     copied = componenttools.copy_components_and_covered_spanners(
    intermediateStaff[:])
...     staff.extend(copied)
...
>>>
```

Figure 2.39: Functions for adding drones and random walks.

Just as a single rotation function added a rotation to the score to generate the first section of the score, the previous functions culminate in two functions that add a “drone and back” random walk gesture to a staff and score respectively. The “startEncroachment” and “endEncroachment” arguments specify the number of sixteenth notes of the random walk through which the drone will continue; that is, if

a random walk section of three measures has been chosen, the startEncroachment will continue the preceding drone into the first measure by n sixteenth notes and the endEncroachment will begin the subsequent drone n sixteenth notes before the end of the third measure. Other arguments specify the durations of the initial drone, the random walk section, and the following drone:

```
>>> def add_drone_and_back_tutti_to_staff(staff, inDroneBaseDuration,
    startEncroachment, walkBaseDuration, outDroneBaseDuration, endEncroachment,
    changeWeights, pitchWeights, trill_in=False, trill_out = False,
    tie_to_previous = False):
...     startLength = len(staff)
...     startEncroachmentAsSixteenths = Duration(startEncroachment, 16)
...     endEncroachmentAsSixteenths = Duration(endEncroachment, 16)
...     inDroneDuration = inDroneBaseDuration - startEncroachmentAsSixteenths
...     if inDroneDuration != 0:
...         add_in_drone_to_staff(staff, inDroneDuration, trill_in,
    tie_to_previous)
...     walkDuration = walkBaseDuration + startEncroachmentAsSixteenths +
    endEncroachmentAsSixteenths
...     add_walk_to_staff(staff, walkDuration, changeWeights, pitchWeights)
...     outDroneDuration = outDroneBaseDuration - endEncroachmentAsSixteenths
...     if outDroneBaseDuration != 0:
...         spanner = add_out_drone_to_staff(staff, outDroneDuration, trill_out,
    tie_to_previous, startLength)
...     if outDroneBaseDuration != 0:
...         return spanner
...
>>> def add_drone_and_back_tutti_to_score(score, inDroneBaseDuration,
    startEncroachments, walkBaseDuration, OutDroneBaseDuration, endEncroachments,
    changeWeights, pitchWeights, trill_in = False, trill_out = False,
    tie_to_previous = False):
...     startEncroachments = sequencetools.CyclicTuple( startEncroachments )
...     endEncroachments = sequencetools.CyclicTuple(endEncroachments)
...     for x, staff in enumerate(score):
...         spanner = add_drone_and_back_tutti_to_staff(staff, inDroneBaseDuration
    , startEncroachments[x], walkBaseDuration, OutDroneBaseDuration,
    endEncroachments[x], changeWeights, pitchWeights, trill_in, trill_out,
    tie_to_previous)
...
```

Figure 2.40: Functions for adding the random walk gesture to score, framed by drones as specified.

The use of this previous function allows a single function to model the random walk section of the score:

```
>>> def make_random_walk_section(score):
...     changeWeights = [ (0, .25), (1, .75) ]
...     pitchWeights = [ (.5, .14), (1, .35), (2, .35), (1.5, .14), (3, .01), (4,
    .01) ]
...     encroachments = range(1,17)
...     endEncroachments = sample(encroachments, 12)
...     add_drone_and_back_tutti_to_score(score, Duration(11,4), [1], Duration
    (4,1), Duration(2,1), endEncroachments, changeWeights, pitchWeights)
...     encroachments = range(1,33)
...     startEncroachments = sample(encroachments, 12)
...     encroachments = range(1,19)
...     endEncroachments = sample(encroachments, 12)
...     spanner = add_drone_and_back_tutti_to_score(score, Duration(2,1),
    startEncroachments, Duration(1,1), Duration(2,1), endEncroachments,
    changeWeights, pitchWeights, trill_out=True, tie_to_previous = True)
...     encroachments = range(0,33)
...     startEncroachments = sample(encroachments, 12)
...     add_drone_and_back_tutti_to_score(score, Duration(2,1), startEncroachments
    , Duration(1,1), 0, [0], changeWeights, pitchWeights, trill_in=True,
    tie_to_previous = True)
...
```

Figure 2.41: The random walk section as a single function.

Because the previous operations have been executed by adding many small durations to a staff, a metric hierarchy must imposed upon the durations in order for the music to comport with common practice conventions regarding rhythmic division, mainly by fusing chains of many shorter, tied durations into longer durations. (Recent versions of the API have eliminated this step of the process with an object oriented model of metrical hierarchies.) Note that this is the opposite of the previous example, in which longer durations needed to be divided in order to comport with the duration of the piece's meter; the system enables either approach, and the formulation of a procedural strategy must address the relationship between the note as an abstraction and the note as a read symbol that conforms to the conventions of common notation:

```
>>> def get_quarter_runs_in_group(group):
...     runs = [ ]
...     run = [ ]
...     for leaf in group:
...         if leaf.written_duration == Duration(1,4):
...             run.append(leaf)
...         else:
...             if run != []:
...                 runs.append(run)
...                 run = [ ]
...     if run != []:
...         runs.append(run)
...     return runs
...
>>> def get_quarter_runs_in_expr(expr):
...     runs = [ ]
...     for group in componenttools.yield_topmost_components_grouped_by_type(expr)
    :
...         if isinstance(group[0], Note):
...             groupRuns = get_quarter_runs_in_group(group)
...             runs.extend(groupRuns)
...     return runs
...
>>> def fuse_quarter_runs_in_bar(bar_leaves_in_chain):
...     runs = get_quarter_runs_in_expr(bar_leaves_in_chain)
...     for run in runs:
...         if len(run) > 1:
...             leaftools.fuse_leaves(run[:])
...
>>> def get_bar_leaves_in_chain(bar, chain):
...     leaves = []
...     for leaf in bar:
...         if bar[0].timespan.start_offset <= leaf.timespan.start_offset and leaf
    .timespan.stop_offset <= bar[-1].timespan.stop_offset:
...             leaves.append(leaf)
...     return leaves
...
>>> def fuse_leaves_if_fully_tied(shard):
...     if tietools.are_components_in_same_tie_spanner(shard[:]):
...         leaftools.fuse_leaves(shard)
...
>>>
```

Figure 2.42: Imposing metric hierarchy by fusing chains of small durations.

These functions for fusing culminate in two functions for imposing metric

hierarchy on a staff and the entire score, respectively:

```
>>> def fuse_tied_through_quarters_by_bars(chain, bars):
...     for bar in bars:
...         bar_leaves_in_chain = get_bar_leaves_in_chain(bar, chain)
...         if bar_leaves_in_chain:
...             fuse_tied_through_quarters_in_bar(bar_leaves_in_chain)
...
>>> def fuse_quarter_runs_by_bars(chain, bars):
...     for bar in bars:
...         bar_leaves_in_chain = get_bar_leaves_in_chain(bar, chain)
...         fuse_quarter_runs_in_bar(bar_leaves_in_chain)
...
>>> def clean_up_durations_in_staff(staff):
...     beats = componenttools.split_components_at_offsets(staff.leaves, [Duration
    (1,4)], cyclic=True)
...     bars = componenttools.split_components_at_offsets(staff.leaves, [Duration
    (4,4)], cyclic=True)
...     for chain in tietools.iterate_nontrivial_tie_chains_in_expr(staff):
...         fuse_tied_through_quarters_by_bars(chain, bars)
...     bars = componenttools.split_components_at_offsets(staff.leaves, [Duration
    (4,4)], cyclic=True)
...     for chain in tietools.iterate_nontrivial_tie_chains_in_expr(staff):
...         fuse_quarter_runs_by_bars(chain, bars)
...
>>> def clean_up_durations_in_score(score):
...     for staff in score:
...         clean_up_durations_in_staff(staff)
...
```

Figure 2.43: Imposing metric hierarchy on the entire score.

The remaining sections of the score could all be modeled similarly: some of the unorthodox notational constructs proved impossible to model using LilyPond without intimate knowledge of the Scheme programming language, which underlies the most basic functions of the typesetting engine. Assuming a valid model of each score section, each section would finally become a constituent line of a function that generates a score object; i. e., with the two previously coded sections:

```
>>> def make_windungen_score():
...     score = make_empty_cello_score(12)
...     make_diatonic_section(score)
...     make_random_walk_section(score)
...     clean_up_durations_in_score(score)
...     clean_up_rests_in_score(score)
...     return score
...
```

Figure 2.44: Creating the score object.

Finally, the score object can be used to initialize a LilyPondFile object, as seen at the end of the Pärt example, the attributes of which can be specified to change the layout and formatting properties of the completed document (B.2).

2.3 Revealed Strengths and Weaknesses of Formalized Score Control

Although laboriously detailed, this step-by-step description of two modeling tasks makes clear the advantages and disadvantages of this method of notation generation. The Pärt example shows that the system models low-complexity structures relatively easily, as demonstrated by the small set of generative pitch and rhythm functions that model most of the score; the addition of dynamics and technical markings does not follow a pattern and must be specified in lists, although even these tasks may be automated somewhat with the use of loops, as in the case of rehearsal marks and some technical indications. This suggests that formalized score control most effectively models works with maximum coherence, works that derive their components in an integral way from a minimally small set of generative principles. As the diversity of organizational logics within a work increases, the complexity of a modeling program must scale proportionally.

Accordingly, the Xenakis example offers a less unanimous view on this method's efficacy. To the modeling process, each shift in texture results in the equivalent of a completely new model of music, and the process of managing the contrasts of one piece becomes the process of modeling many different kinds of musical activ-

ity, each of which could result in its own composition. That the system can indeed model each of these varying musical organizations and its accompanying notational comportments is a testament to the flexibility of the system, but one envies the agility with which a more traditional approach to composing facilitates the sudden invention of contrasting material. (It is arguably the case that traditional composition can occur within the system by simply specifying the pitches and durations to be added to score container objects; however, the lack of an underlying model of music/composition guiding the generation of components would locate this in the realm of composition, not analysis/modeling.) In an analysis task, the demand for this kind of rigorous specification leads to a more concrete understanding of which principles of construction might lead to the composition; at the same time, this reveals that traditional musical analysis can be understood as the task of identifying not every single operative constraint, as must be done here, but the most relevant structures and constraints at each moment in a work of music.

It is also the case, from the perspective of musical analysis, that many of the functions here that fuse smaller durations or divide larger durations model the metric conventions of common practice notation rather than symbolic manipulations indigenous to a specific composition. One might argue that these operations remain outside of the realm of analysis, as they ally more closely to a notion of performance practice or notational technology than to the qualities of a work composed within a performance tradition, using a specific notational technology. (They model notation, not music/composition.) One may also argue, in response, that compositional style extends into simple notational choices: the additive rhythms of Olivier Messiaen, for example, extend additive formulations of meter and rhythm into the realm of rhythmic convention by flouting the regular metric division of a bar; likewise, notations that elect to show clearly the way that a beat has been divided communicate through a primarily divisive, rather than additive, view of the relationship between rhythm and meter. Although there might be a disconnect between musical idea and rhythmic convention — it would be equally possible to convey Messiaen's rhythms with a divisive metric notation — there is nonetheless a relationship between the two; notational habit countenances compositional thought.

These analytic tasks elide with compositional applications of the system, and the most significant potential of this kind of comprehensive modeling may lie in future compositional applications of musical models derived from analysis: if coding is done with good style, in easily testable and reconfigurable modules, it becomes easy to reuse modeling functions, substituting new values for the function's arguments, in order to create new music, which may or may not resemble the original composition. In this sense, the interplay between composition and analysis, between historical understanding and contemporary creation, has also been formalized, and a unit of code created to understand history can easily be repurposed for the creation of new work.

Chapter 3

Automated Notation for the Analysis of Recorded Music

3.1 Background

The literature shows that data visualizations can illustrate complex interrelationships in musical structures that are otherwise difficult to notice in large data sets extracted from recorded music (Sapp 2005, Kunze and Taube 1996, Mitroo, Herman, and Badler 1979, Sapp 2011, Cook 2007) and that toolkits for visualizing and annotating musical content are now widely available (Cannam et al. 2006, Herrera et al. 2005, Marsden et al. 2007); at the same time, an increasing number of classical musicians and historical musicologists have become comfortable engaging recorded history as a meaningful input into the practice of performance and scholarship (Butt 2002, Cook 2010, *Reactions to the Record* conferences at Stanford University, 2007, 2009, and 2011 (no proceedings available)). While trained musicians read musical notation fluently, they are often unfamiliar with other data visualization formats. If data about recordings is to meaningfully inform performance practice, there are two possible compromises: institutions can change, teaching music students to read scientific visualizations, or researchers can communicate multidimensional analyses through musical notation. This chapter addresses the latter possibility. How can data visualization reconsider and remap musical notation to share recorded performance

data with musicians? As one answer to this question, a Python-based system repurposes the Abjad API for Formalized Score Control to notate a comparative analysis of data corresponding to 61 recordings of the first movement of Anton Webern's Piano Variations (op. 27), previously analyzed to extract onset timing and amplitude information by Craig Sapp.

3.2 Methodology for Representing Amplitude and Onset Time as Notation

The link between phrasing and amplitude in the performance of music is important but difficult to study, but a computational, quantitative analysis of recordings may elucidate this relationship. Given a data set that expresses the loudness of each event in a piano performance, as well as events' relative temporal occurrence, it becomes possible to study comparatively the ways in which performers shape musical events into larger gestalts by visualizing this information and comparing different performances from the same score. To this end, the Abjad API for Formalized Score Control is used to process data from many recordings and visualize the data using an augmented version of conventional musical notation.

Augmented Notation

Two key modifications enable common practice notation to convey additional information about performance data. First, proportional notation, in which space and time are linked with a consistent ratio between measured time and horizontal space on the page, ensures that the strict temporal relationships of events have been given as accurate a graphic correspondence as possible; in order to give this horizontal, spatial correspondence rhythmic primacy, the conventional elements of rhythmic notation — beams, tuplet brackets, flags, and dots — have been hidden. (They are artifacts of the quantization process's arbitrary division of time.) Second, rather than the traditional, low-resolution system of dynamic indications, which offers around a maximum of ten possible amplitude indications, the greyscale color value of the notehead indicates the loudness of the event; the whiter the notehead, the softer the

event, allowing around one hundred possible amplitudes. Silences have not been represented.

The Program

The following Python code notates performance data for multiple recordings as a score with one interpretation per staff, according to the system described above. Much of the program requires basic data processing from files, especially the first section of the code, in which the performance data must be input into the program:

```
>>> def getFiles(dirString):
...     names = os.listdir(dirString)[1:]
...     files = []
...     for name in names:
...         filename = "/Users/jeffreytrevino/Documents/UCSD/dissertation/chapters
    /webern/performances/mvmt1/" + name
...         with open(filename) as f:
...             lines = f.readlines()
...             files.append( lines )
...             f.close()
...     return files
...
>>> def splitFile(lines):
...     splits = []
...     for line in lines[1:]:
...         chopped = line.split()
...         splits.append(chopped)
...     return splits
...
>>> def listData(splits):
...     typedFile = []
...     for line in splits[8:]:
...         typed = [ float(line[0]), float(line[1]), int(line[2])]
...         typedFile.append( typed )
...     return typedFile
...
>>> def linesToLists(lines):
...     splits = splitFile(lines)
...     lists = listData(splits)
...     return lists
...
>>> def getNameMarkupFromLists(files):
...     nameStrings = []
...     for lineList in files:
...         name = lineList[3][14:-1]
...         year = lineList[4][9:-1]
...         record = lineList[5][10:-1]
...         nameString = name + ", (" + year + ")"
...         nameStrings.append( nameString )
...     return nameStrings
...
```

Figure 3.1: Reading recording data from file in Python.

A representation of the score is necessary, as well, because each recorded performance must be pitched according to the score's pitch information. This may either

be input from the file or hard-coded into the program; because file input allows the program to work more generally, this approach has been adopted here:

```
>>> def readScore(fileDir):
...     score = open(fileDir)
...     scoreLines = score.readlines()
...     score.close()
...     return scoreLines
...
>>> def splitScore(score):
...     splits = []
...     for line in score[1:]:
...         chopped = line.split()
...         splits.append(chopped)
...     return splits
...
>>> def typeScore(splits):
...     typedScore = []
...     for line in splits:
...         typed = [ int(line[0]), float(line[1]), int(line[2]), float(line[3]) ]
...         replaced = []
...         replaced.extend( typed[:] )
...         replaced.extend( line[4:] )
...         typedScore.append( replaced )
...     return typedScore
...
>>> def getScore(fileDir):
...     lines = readScore(fileDir)
...     splits = splitScore(lines)
...     typed = typeScore(splits)
...     return typed
...
>>> def truncate_score_events_to_pitch_lists(score):
...     pitch_lists = [ ]
...     for event in score:
...         pitch_list = event[4:]
...         pitch_lists.append( pitch_list )
...     return pitch_lists
...
```

Figure 3.2: Processing recording data in Python.

Next, the program must translate between the file format's pitch representation and Abjad's pitch representation; this varies greatly according to the text file's data representation but will always be straightforward using Python's string pro-

cessing functions:

```
>>> def last_character_in_string_is_accidental(string):
...     if string[-1] == '-' or string[-1] == '#':
...         return True
...     else:
...         return False
...
>>> def get_octave_tick_string_from_craigslist_string(craigslist_string):
...     if last_character_in_string_is_accidental(craigslist_string):
...         octave_displacement = len(craigslist_string) - 1
...
...     else:
...         octave_displacement = len(craigslist_string)
...
...     if craigslist_string[0].isupper():
...         octave_number = 4 - octave_displacement
...     else:
...         octave_number = 3 + octave_displacement
...     tick_string = pitchtools.octave_number_to_octave_tick_string(octave_number
    )
...     return tick_string
...
>>> def get_pitch_name_string_from_craigslist_string(craigslist_string):
...     pitch_string = craigslist_string[0].lower()
...     return pitch_string
...
>>> def convert_accidental(accidental):
...     accidental_dictionary = {'-': 'f', '#': 's'}
...     return accidental_dictionary[ accidental ]
...
>>> def get_pitch_string_from_craigslist_string(craigslist_string):
...     pitch_string = ""
...     pitch_name_string = get_pitch_name_string_from_craigslist_string(
    craigslist_string)
...     pitch_string += pitch_name_string
...     if last_character_in_string_is_accidental(craigslist_string):
...         accidental_string = convert_accidental( craigslist_string[-1] )
...         pitch_string += accidental_string
...     octave_tick_string = get_octave_tick_string_from_craigslist_string(
    craigslist_string)
...     pitch_string += octave_tick_string
...     return pitch_string
...
```

Figure 3.3: Reading pitch data from the score file.

After these conversion functions query the pitches in the score representation, the pitches can be converted to Abjad pitch representations:

```
>>> def convert_craigslist_pitch_string_to_named_chromatic_pitch(craigslist_string
    ):
...     pitch_string = get_pitch_string_from_craigslist_string(craigslist_string)
...     pitch = pitchtools.NamedChromaticPitch(pitch_string)
...     return pitch
...
>>> def convert_craigslist_event_to_abjad_pitches(craigslist_event):
...     out_list = [ ]
...     for pitch in craigslist_event:
...         abjad_pitch = convert_craigslist_pitch_string_to_named_chromatic_pitch
    (pitch)
...         out_list.append( abjad_pitch )
...     return out_list
...
>>> def convert_craigslist_to_abjad_pitches(craigslist):
...     output = [ ]
...     event = [ ]
...     for event in craigslist:
...         converted_event = convert_craigslist_event_to_abjad_pitches(event)
...         output.append( converted_event )
...     return output
...
```

Figure 3.4: Converting the score to Abjad's pitch representation.

The use of data abstraction allows the previous functions to culminate in a single function that parses multiple files:

```
>>> def parseFiles(files):
...     eventLists = []
...     for eachFile in files:
...         eventList = linesToLists( eachFile )
...         eventLists.append( eventList )
...     return eventLists
...
```

Figure 3.5: The final file parsing function.

To determine notehead color, each performance's maximum and minimum amplitude values must be determined; after this, the color of each notehead may

be determined by measuring amplitudes relative to the minimum and maximum amplitude in a given performance:

```
>>> def calculateAmplitudeMinAndMax(eventList):
...     amplitudes = [x[1] for x in eventList]
...     return min(amplitudes), max(amplitudes)
...
>>> def amplitudeToGrayscale(amplitude, bounds):
...     shifted = amplitude - bounds[0]
...     range = bounds[1] - bounds[0]
...     scaled = 100 - int(shifted/range * 100 )
...     return scaled
...
>>> def colorChain(pair, bounds):
...     chain = pair[0]
...     event = pair[1]
...     amplitude = event[1]
...     for note in chain:
...         grayscale = amplitudeToGrayscale(amplitude, bounds)
...         note.override.note_head.color = schemetools.Scheme("x11-color", "'grey
    "+str(grayscale))
...         note.override.accidental.color = schemetools.Scheme("x11-color", "'
    grey"+str(grayscale))
...
```

Figure 3.6: Coloring noteheads according to amplitude.

Event data must be pre-processed before quantization, to ensure that all performances begin from a time of 0 and to convert a list of onset times into a list of durations (assuming 100% legato, because the data set does not include silences); after this, Abjad's Q-grids quantizer converts the list of durations for each performance into a voice containing Abjad Note objects, durated according to an error-minimizing search, derived from the work of Paul Nauert and implemented by Josiah Oberholtzer (Nauert 1994):

```
>>> def shiftOnsets(onsets):
...     shift = onsets[0]
...     shifted = [x - shift for x in onsets]
...     return shifted
...
>>> def quantizeOnsets(quantizer, q_schema, durations):
...     msDurations = [1000 * x for x in durations]
...     msDurations = [int(x) for x in msDurations]
...     q_events = quantizationtools.QEventSequence.from_millisecond_durations(
    msDurations)
...     voice = quantizer(q_events, q_schema = q_schema, attach_tempo_marks =
    False)
...     return voice
...
>>> def labelEvents(expr):
...     for i,chain in enumerate(tietools.iterate_tie_chains_in_expr(expr.leaves
    [:-1])):
...         markup = markuptools.Markup(r'\rounded-box "' + str(i+1) + r'"', Up, "
    event_number")(chain[0])
...
>>> def onsetsToDurations(onsets):
...     durations = []
...     for x in range(len(onsets) -1 ):
...         duration = onsets[x+1] - onsets[x]
...         durations.append(duration)
...     durations.append(1)
...     return durations
...
>>> def makeDuratedVoice(quantizer, q_schema, eventList):
...     onsets = [x[0] for x in eventList]
...     onsets = shiftOnsets(onsets)
...     durations = onsetsToDurations(onsets)
...     voice = quantizeOnsets(quantizer, q_schema, durations)
...     return voice
...
```

Figure 3.7: Quantizing performance events with Abjad.

After the performance file's events have been converted into a Voice object containing notes, the score's pitch information can be used to pitch the performance with the notes in the score:

```
>>> def make_chord_string_from_pitch_list(pitch_list):
...     string = pitch_list[0].chromatic_pitch_name
...     for pitch in pitch_list[1:]:
...         string += (” ” + pitch.chromatic_pitch_name)
...     return string
...
>>> def make_chord_from_pitch_list(chord_duration, pitch_list):
...     chord_duration_string = chord_duration.lilypond_duration_string
...     chord_pitches_string = make_chord_string_from_pitch_list(pitch_list)
...     chord_string = ”<” + chord_pitches_string + ”>” + chord_duration_string
...     chord = Chord(chord_string)
...     return chord
...
>>> def replace_leaf_with_pitched_leaf(leaf, pitch_list):
...     index = leaf.parent.index( leaf )
...     duration = leaf.written_duration
...     if 1 < len(pitch_list):
...         chord = make_chord_from_pitch_list(duration, pitch_list)
...         leaf.parent[ index ] = chord
...     else:
...         note = Note(pitch_list[0], duration)
...         leaf.parent[ index ] = note
...
>>> def pitch_chain_with_pitch_list(chain, pitch_list):
...     for leaf in chain:
...         replace_leaf_with_pitched_leaf(leaf, pitch_list)
...
>>> def pitch_tie_chains_in_voice_with_pitch_lists(chains, pitch_lists):
...     for pair in zip(chains, pitch_lists):
...         chain = pair[0]
...         pitch_list = pair[1]
...         pitch_chain_with_pitch_list(chain, pitch_list)
...
```

Figure 3.8: Pitching each performance according to the score data.

Next, each performance can be split at middle C and redistributed onto the treble and bass clefs of a piano staff, so that the notation may correspond more clearly to that of the published score:

```
>>> def remove_number_label_from_chord(chord):
...     markups = markuptools.get_markup_attached_to_component(chord)
...     for markup in markups:
...         if markup.markup_name == "event_number":
...             markup.detach()
...
>>> def replace_note_below_split_with_rest(note, split_pitch):
...     if split_pitch.chromatic_pitch_number > note.written_pitch.
    chromatic_pitch_number:
...             duration = note.written_duration
...             rest = leaftools.make_tied_leaf( Rest, duration )
...             index = note.parent.index( note )
...             note.parent[ index: index+1 ] = rest
...
>>> def remove_chord_pitches_below_split(chord, split_pitch):
...     index = chord.parent.index( chord )
...     for note in reversed(chord):
...         if split_pitch.chromatic_pitch_number > note.written_pitch.
    chromatic_pitch_number:
...             note_index = chord.written_pitches.index( note )
...             chord.pop(note_index)
...     if 0 == len(chord.written_pitches):
...         rest = leaftools.make_tied_leaf( Rest, chord.written_duration )
...         chord.parent[ index:index+1 ] = rest
...
>>> def remove_chord_pitches_above_split(chord, split_pitch):
...     index = chord.parent.index( chord )
...     popped = 0
...     for note in reversed(chord):
...         if split_pitch.chromatic_pitch_number <= note.written_pitch.
    chromatic_pitch_number:
...             note_index = chord.written_pitches.index( note )
...             chord.pop(note_index)
...             popped = 1
...     if 0 == len(chord.written_pitches):
...         rest = leaftools.make_tied_leaf( Rest, chord.written_duration )
...         chord.parent[ index:index+1 ] = rest
...     elif popped:
...         remove_number_label_from_chord(chord)
...
```

Figure 3.9: Typographical manipulations to split one staff to a piano staff.

After functions to replace components appropriately with rests have been initialized, it becomes possible to split the staff by creating two copies of the staff and then deleting certain components either above or below the specified split point:

```
>>> def split_components_to_piano_staff_at_pitch(voice, split_pitch = pitchtools.
    NamedChromaticPitch(”c’”)):
...     labelEvents(voice)
...     piano_staff = scoretools.PianoStaff()
...     treble_staff = Staff()
...     treble_staff.name = ”treble”
...     bass_staff = Staff()
...     bass_staff.name = ”bass”
...     copies = componenttools.copy_components_and_covered_spanners( [voice] )
...     treble_voice = Voice(copies)
...     copies = componenttools.copy_components_and_covered_spanners( [voice] )
...     bass_voice = Voice(copies)
...     remove_pitches_below_split_in_components(treble_voice, split_pitch)
...     remove_pitches_above_split_in_components(bass_voice, split_pitch)
...     bass_staff.extend(bass_voice[:])
...     treble_staff.extend(treble_voice[:])
...     contexttools.ClefMark(’bass’)(bass_staff)
...     piano_staff.extend([treble_staff,bass_staff])
...     piano_staff.override.beam.transparent = True
...     piano_staff.override.tuplet_bracket.stencil = False
...     piano_staff.override.tuplet_number.stencil = False
...     piano_staff.override.dots.transparent = True
...     piano_staff.override.rest.transparent = True
...     piano_staff.override.tie.transparent = True
...     piano_staff.override.stem.transparent = True
...     piano_staff.override.flag.stencil = False
...     for staff in piano_staff:
...         for chain in tietools.iterate_tie_chains_in_expr(staff):
...             for note in chain[1:]:
...                 note.override.note_head.transparent = True
...                 note.override.note_head.no_ledgers = True
...                 note.override.accidental.stencil = False
...     contexttools.TimeSignatureMark((1, 4))(piano_staff[0])
...     return piano_staff
...
```

Figure 3.10: Splitting a performance to a piano staff.

Finally, the previous functions can be encapsulated into a function that converts a single performance into a notated staff and a second function that calls the first to convert an arbitrary number of performance files into a score containing one performance per staff:

```
>>> def make_pitched_and_colored_piano_staff(voice, score_pitch_lists,
    performance_event_list):
...     bounds = calculateAmplitudeMinAndMax(performance_event_list)
...     chains = tietools.iterate_tie_chains_in_expr(voice)
...     pitch_tie_chains_in_voice_with_pitch_lists(chains, score_pitch_lists)
...     chains = tietools.iterate_tie_chains_in_expr(voice)
...     chainsAndEvents = zip(chains,performance_event_list)
...     for pair in chainsAndEvents:
...         colorChain(pair, bounds)
...     piano_staff = split_components_to_piano_staff_at_pitch(voice)
...     return piano_staff
...
>>> def makeAmplitudePhrasingScore(score_pitch_lists, performance_event_lists):
...     staffs = []
...     q_schema =  quantizationtools.BeatwiseQSchema(tempo = contexttools.
    TempoMark((1,4), 60))
...     quantizer = quantizationtools.Quantizer()
...     for x,performance_event_list in enumerate(performance_event_lists):
...         voice = makeDuratedVoice(quantizer, q_schema, performance_event_list)
...         print voice
...         piano_staff = make_pitched_and_colored_piano_staff(voice,
    score_pitch_lists, performance_event_list)
...         staffs.append( piano_staff )
...         print ”MAKE SCORE -- Added staff ” + str(x+1) + ”/” + str( len(
    performance_event_lists) )
...     score = Score([])
...     for staff in staffs:
...         score.append( staff )
...     score.set.proportional_notation_duration = schemetools.SchemeMoment(1, 56)
...     score.set.tuplet_full_length = True
...     score.override.spacing_spanner.uniform_stretching = True
...     score.override.spacing_spanner.strict_note_spacing = True
...     score.set.tuplet_full_length = True
...     score.override.tuplet_bracket.padding = 2
...     score.override.tuplet_bracket.staff_padding = 4
...     score.override.tuplet_number.text = schemetools.Scheme(’tuplet-number::
    calc-fraction-text’)
...     score.override.time_signature.stencil = False
...     score.override.span_bar.stencil = False
...     marktools.LilyPondCommandMark(”set Timing.defaultBarType = \”dashed\””)(
    score)
...     return score
...
```

Figure 3.11: Final encapsulations, including format and layout of the Score object.

Lastly, the score initializes a LilyPondFile object, the attribute values of which

determine the layout and format of the generated document; after this, the entire process can be encapsulated in a single function:

```
>>> def formatAmplitudePhrasingScore(files, score):
...     nameStrings = getNameMarkupFromLists(files)
...     namesAndStaffs = zip(nameStrings,score)
...     for name,staff in namesAndStaffs:
...         contexttools.InstrumentMark(name,name, target_context = scoretools.
    PianoStaff)(staff)
...     lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
...     lilypond_file.paper_block.paper_width = 36 * 25.4
...     lilypond_file.paper_block.paper_height = 48 * 25.4
...     lilypond_file.paper_block.left_margin = 1.5 * 25.4
...     lilypond_file.paper_block.right_margin = 1.5 * 25.4
...     lilypond_file.paper_block.top_margin = .5 * 25.4
...     lilypond_file.paper_block.ragged_bottom = False
...     lilypond_file.global_staff_size = 8
...     lilypond_file.layout_block.indent = 0
...     lilypond_file.header_block.title = markuptools.Markup("Loudness and
    Duration in 61 Recordings of Webern's Piano Variations")
...     lilypond_file.layout_block.ragged_right = False
...     return lilypond_file
...
>>> def quantizeInterpretations(scoreDirectory, performanceDirectory):
...     performance_files = getFiles(performanceDirectory)
...     webern_score = getScore(scoreDirectory)
...     score_events = truncate_score_events_to_pitch_lists(webern_score)
...     score_pitch_lists = convert_craigslist_to_abjad_pitches(score_events)
...     performance_event_lists = parseFiles(performance_files)
...     analysis_score = makeAmplitudePhrasingScore(score_pitch_lists,
    performance_event_lists)
...     lily = formatAmplitudePhrasingScore(performance_files, analysis_score)
...     return lily
...
```

Figure 3.12: The final automatic notation function.

Calling this last function on a score and performance directory will generate a comparative score from the performance files in the performance directory; because the document comparing the full 61 performances was too large for the current document, the results of the program when run on performance data only for Glenn Gould's interpretations of the Webern movement have been included in the score appendix (B.3).

3.3 Conclusion and Future Directions for Research

Musical notation functions as a data visualization, not a musical analysis, in this application. This comparative notation does not make obvious the relationships between timing and amplitude that together constitute phrasing in an interpretation. Rather, the score offers, as an interface for analysis, an exact notation of two constituent elements that may be closely examined to reveal the nature of phrasing. A review of the diversity of interpretations represented in the comparative notation here reveals the myriad possibilities inherent in performances from just one score, in the choice of tempo, the stretching of time within a tempo, and the corresponding archings of amplitude that group musical events. Each interpretation proposes its own language of speeding and slowing, of loudening and softening, in order to convey larger musical shapes, and each interpretation can be heard as an equally valid parsing of the musical score.

This application also demonstrates that digital tools for analysis and composition are converging: Michael Cuthbert, leader of the Music21 project, a Python library for corpus-based musicology, has indicated that upcoming versions of Music21 will include new functions that make it easier for composers to create new notations using the library (Cuthbert 2013); likewise, this analytical use of an ostensibly composer-oriented Python library shows that it is possible to use the same notational utilities for musicological purposes.

Many possible extensions of this interface could offer an improved visualization suite for recording data. A visualization that links more clearly each interpretation to the score from which the interpretations have been is an important goal: while the current system of enumerated events allows reference to an annotated musical score, it does not offer a transparent link between interpretation and score. A legible relationship could be easily introduced by reintroducing beams into the notation to group noteheads as the score beams them, while retaining the proportional spacing that results from the interpretation.

A GUI specific to this application would also be helpful. The linking of audio playback and score would allow the scholar-performer to scrub or loop audio, in order to focus attention on a specific moment in the music, and an interface as

simple as a series of radio buttons would allow the user to select specific interpretations to compare. A graphical link between notation and audio, in the form of a scrolling timeline and the ability to highlight notation and hear the corresponding audio, would facilitate the use of the proposed data representation.

Chapter 4

Compositional Applications

This chapter discusses the author's algorithmic composition practice, which recently has begun to include applications of the Abjad API for Formalized Score Control. First, notations made between 2004 and 2008 — created without the aid of automated notation systems but nonetheless with adherence to *a priori* constraints — provide a context for recent, computer-assisted notations. Subsequent discussion addresses the aesthetic goals and compositional methods of recent, computer-assisted notations as both continuations of and departures from the techniques and agendas of these earlier works, as is expected from artistic careers that divide into periods before and after the adoption of the computer as a compositional aid (Rosen et al. 2011).

The chapter does not seek to explain the recent work primarily as the application of a system; rather, the goal is to demonstrate a continuity of artistic concern between earlier work and later work that positions an object-oriented notation system as a novel but reasonable strategy for achieving extant goals, while remaining critically aware of the way that the new technology might alter or reframe these goals. This alignment between aesthetic interest and technology serves to emphasize — again in the spirit of generative task as an analytic framework — that technologies are more or less appropriate for different artistic practices.

4.1 Algorithmic Tendencies, 2004—2008

Although notations created from 2004 to 2008 rely upon algorithmic construction and *a priori* constraint for both the large-scale formal disposition of materials and the local profile of musical gestures, they were composed without the aid of automated notation systems: although the work depended heavily on modeled musical notation, in the form of commercial typesetting programs, there was no computational modeling of musical abstractions or compositional processes.

The design of unorthodox notational constructs, rather than computer programs, models novel musical concepts in some these early works. Alternative graphic communication strategies express a composition's approaches to form: so-called "mobile" notational constructs express the navigable boundaries of indeterminate formal structures. Although pragmatically engaged in order to leave unspecified the particular succession of events in a musical experience, the graphic communication of available trajectories through a system, rather than the specification of the succession of events itself, reorients the graphic artifact away from the execution of sequenced events, toward the documentation and communication of broader compositional thought.

Subtle additions to common notation principles inject a formalized indeterminacy into the procession of events, as can be seen in the colored pitch notation system for *Perfection Factory* and the invitation to perform indeterminate "solo" gestures in *Binary Experiment for James Tenney*.

The physical constraints of performance play a substantial role in formalizing indeterminate constraints for these early works, as well. Empirical constraints, such as the instruction to perform the highest note possible in *Forty-two Statcoulombs*, the instruction to listen for the emergence of multiphonic sounds in *Unit for Convenience and Better Living 003* before proceeding, and the instruction to listen until a sound has completely died away in *Perfection Factory* before proceeding locate musical choices conventionally determined by abstract measurement — musical pitch and the temporal placement of events via divided metric time — in the acts of listening and performing.

These early works point to an artistic agenda of unpredicted discovery ap-

proached mainly through a core set of strategies: 1) enabling the unpredictably rich by circumscribing via constraint the liminal and contingent; 2) enabling the unpredictably coherent by allowing musical syntax to emerge from generative principles or systems; and 3) enabling the unpredictably metaphoric by providing concrete frames of reference for listening experiences and composition.

4.1.1 *Substitute Judgment* (2004) for Solo Percussionist

Inspired by readings of philosophical inquiries into the ethics of Alzheimer's Disease patients' legal status as decision makers, *Substitute Judgment* for solo percussion presents four simple materials as one composition based primarily on interruption. In the same way that contextualized assessments based on memory give way to intuitive assessments of kindness or enmity as Alzheimer's runs its course, the piece focuses on the profound changes that come about by an apparently simple, even trivial change in priority: each of the four materials consists of a single process which runs its course independently of the others and radically alters its material.

Although the score is notated with common notation, the nature of the four processes was first determined according to an arbitrary durational constraint expressed as a drawing. The materials' total durations are by successive divisions in half, so that material A would be four minutes long, material B two minutes long, material C one minute long, and material D thirty seconds long:

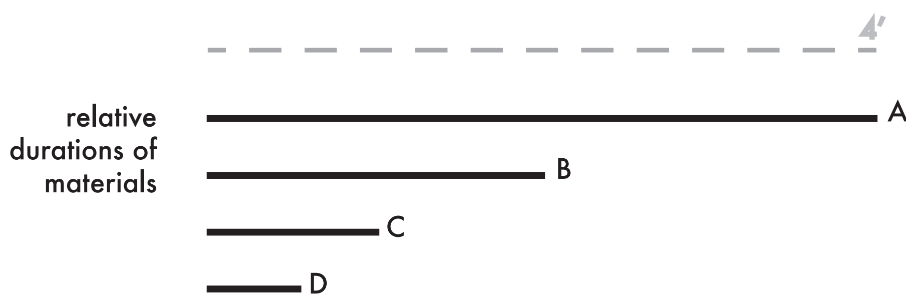


Figure 4.1: Relative Durations of Materials in *Substitute Judgment* (2004).

The form of the piece was determined from this initial drawing, by chopping materials B, C, and D into two, three, or four parts and distributing their materials

in time; as a simplified example, if all three materials were to be cut into two equal parts and redistributed, the form would appear as the following:

Figure 4.2: Division and Formal Disposition of Materials in *Substitute Judgment* (2004).

To arrive at the final composition, this diagram is treated as the plan (overhead view) of a structure. If a viewer positioned in front of the structure (in the plan view, graphically "below" the structure) looks at the structure, the materials as seen from left to right (the x-axis still expresses temporal succession) constitute the materials to be heard; that is, the order of heard materials, as read from the diagram, would be A, B, C, D, C, B, A, etc. This is an unnecessarily elaborate generative mechanism to derive a series of simple palindromes, a structure used without this cumbersome, generative apparatus in the trio composition, *Zoetropes*.

The duration of each material suggested various processes. Material A consists of the gradual displacement of one groove figure by another over the course of four minutes. Material B consists primarily of silence and of randomly selected soft sounds, which gradual converge on the glass bottle sound over the course of two minutes. Material C consists of jeté gestures between randomly selected instruments, the selection of which gradually converges on the bongos over the course of one minute. Material D consists entirely of woodblock eighth notes, increasingly ornamented by glass bottle eighth notes over the course of thirty seconds.

Both the form and the individual trajectory of musical materials in the composition have been highly constrained; in fact, the identity of each of the materials is primarily the communication of these governing constraints. Rather than allow each of these trajectories to speak clearly, however, formal constraint fractures and rearranges these autonomous participants into an unpredictably coherent play of sudden shifts between materials.

4.1.2 *Binary Experiment for James Tenney* (2005) for Four Contrabasses

Binary Experiment for James Tenney is a mobile notation for four contrabasses. For each of the two sections, the players move clockwise or counter-clockwise around "mobiles" of three pitches or actions, starting on an arbitrarily chosen pitch, moving one around the circle in an arbitrary direction, and performing each pitch for the duration of a single bowstroke at the dynamic through which the player passes to arrive at the performed note. Because the players may navigate around the mobile in an arbitrary direction, the link between a certain pitch and a certain dynamic varies, as does the duration of each pitch, because of the link between the dynamic of the note and its duration; this is due to the link between the physical length of the bow and the time it takes to perform a single bowstroke at a specified dynamic. Stopwatches determine when a performer should move to a subsequent mobile — there are three for each of the two parts — and a small set of timelines at the top of the score illustrates the times at which each of the four players moves to a new mobile; the change times are also specified between the mobile graphics, for performance convenience.

Figure 4.3: Section A of *Binary Experiment for James Tenney* (2005) for four contrabasses.

Part B rewrites Part A with a new set of constraints. Each bassist performs on only one of the four bass strings, bouncing the bow on the string for the specified amount of time (slow, medium, or fast, each accompanied by a duration in seconds). In the second mobile, a "solo" of between five and fifteen seconds may be performed, still on only one string. Rather than passing through dynamic markings as they navigate the mobiles, the performers pass through wait times, during which they perform silence. These wait times increase throughout, with the result of an increasingly sparse texture toward the end of the composition.

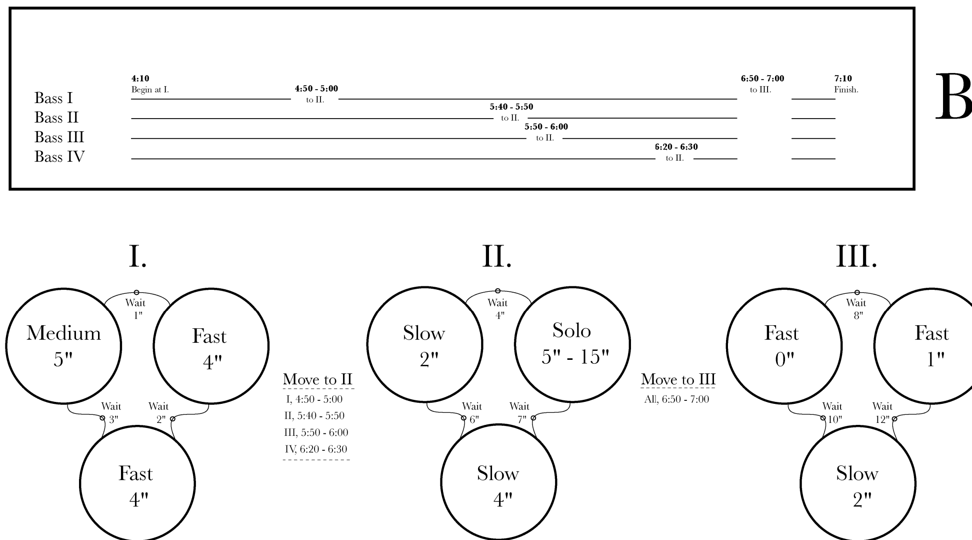


Figure 4.4: Section B of *Binary Experiment for James Tenney* (2005) for four contrabasses.

4.1.3 *Mobile* (2005) for Tenor Saxophone

Mobile for Tenor Saxophone adopts the same binary, mobile-based notational construct with two different sets of constraints. In an A section, a wandering, diatonic melody is successively ornamented by recursively nesting its own intervals upon itself; the performer passes through pairings of dynamic markings and tempos while circulating around the mobile. In contrast to the seamless mobile navigations of the contrabass quartet, this makes audible, via potentially sudden shifts in both tempo and dynamic, the move from one mobile cell to another.

Figure 4.5: Section A of *Mobile* (2005) for tenor saxophone.

While this first section allows flexible navigation of traditionally specified musical material, the second section leaves material radically unconstrained. The performer is instructed to first construct a "scale" of seven multiphonic trills, arranging them from least to most dissonant, as listened (one is the most consonant and seven the most dissonant). The performer then navigates a mobile, passing through couplings of dynamic markings and trill shape, in the form of an illustrated signal; a wave crest indicates a move to the upper sound in the trill, and a wave trough indicates a move to the lower sound of a trill. Although the performer navigates just one mobile, rather than several, global form is nonetheless carefully specified: the score specifies that the A section should last three minutes, while the B section should last four minutes.

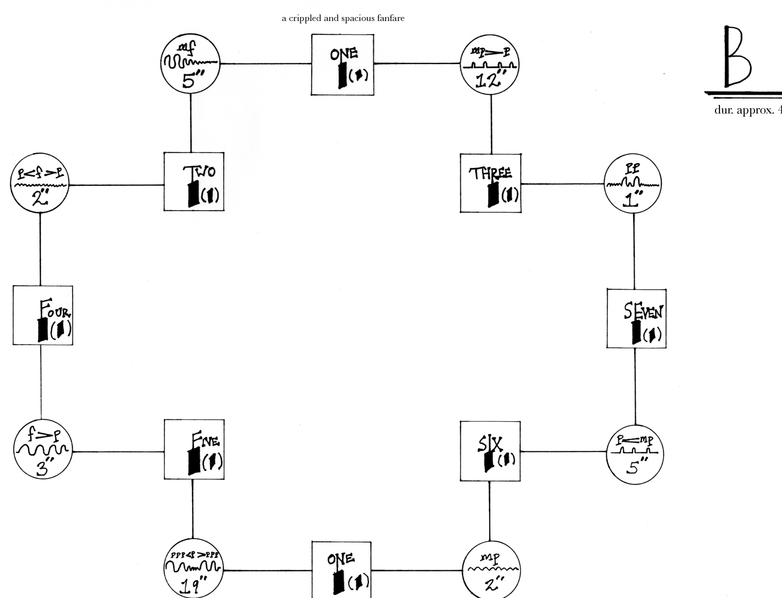


Figure 4.6: Section B of *Mobile* (2005) for tenor saxophone.

In addition to the described traversals, the score also instructs to perform silences of two to forty seconds duration between the performance of each cell in both parts A and B. To weight the probability distribution of silence durations, the score instructs that relatively longer silences should be performed relatively less frequently.

4.1.4 *Zoetropes* (2005—6) for Bass Clarinet, Cello, and Percussion

Inspired by the eponymous, proto-cinematic machine, the structure of *Zoetropes* was planned initially as an entirely palindromic structure, the local phrase structure of which would also be palindromic: just as sufficiently rapid movement of still images causes, beyond some perceptual threshold, the impression of a moving image, the application of palindromic structures to increasingly small timescales would eventually produce an audibly palindromic experience. (This analogy is dubious, but artistically intriguing.) In contrast to the previously described works, in which form and material were governed by differing but complementary logics of constraint, Zoetropes applies the same constraint at two levels of structure, leaving

the temporally shortest level of structure open to a diversity of constraints, in the manner of the constituent materials of *Substitute Judgment*. In composition, it was decided that a non-palindromic coda would break free from the initial design after a prolonged bass clarinet solo; however, the formal plan was executed in tact for the majority of the work.

Figure 4.7: Bass clarinet solo from *Zoetropes* (2005—6).

The bass clarinet solo that marks the point between rigorous observance of structural constraint and its subsequent abandonment serves as a representative example of the nature of constraint governing the specific materials in the work. Here, a single gesture repeats, with several types of inflection and interruptions: hyper-specified “tacet” durations stand in for conventionally notated rests, and equally overdetermined fermatas specify the duration of embouchure multiphonics (unfilled parallelogram noteheads). As if the result of one of the mobile structures described above, sudden, simultaneous changes of tempo and dynamic intervene to shift the flow of time suddenly throughout. Measure 122 represents the local midpoint of the structural palindrome, and the sixteen-second measure indicates this sonically with a simple, pyramid-shaped embouchure multiphonic, which increases attack density

and dynamic into the structural point of reflection, and reduces these parameters exiting the midpoint; after this, the music is an exact reflection of the previous music (measures 123—6 are the reverse of measures 116—120).

As in *Substitute Judgment*, a straightforward process or simple repetition has been inflected via the simultaneous interference of competing modifiers. This suggests a conceptual model akin to the object in object-oriented programming: a musical material seems to have attributes – in the case of the main material, tempo; traversal in one of two directions, depending on which side of the palindromic structure it resides; and dynamic. In the case of the intervening interruptions, materials have attributes like dynamic, tempo, and something akin to "window size": measures 108 and 119 seem to be truncated samples from measure 122. Even the "tacet" circles seem to have a single duration attribute determined randomly within a range of possible values. The repetitive, although fractured, nature of the final audible surface draws attention to this parametric variation of otherwise constant musical material.

4.1.5 *Unit for Convenience and Better Living 003* (2006) for Solo Bass Clarinet

Figure 4.8: Materials in *Unit for Convenience and Better Living 003* (2006).

Unit for Convenience and Better Living 003, like *Substitute Judgment*, depends on the mutually interrupting exposition of four contrasting materials. In this case, each material is given a distinctive profile via dynamics, tempo, register, and gestural comportment: the materials are a slurred altissimo figure; a "backwards" sounding multiphonic, inspired by the sound of reversed magnetic tape playback, the duration

of which is denoted with a "TS" time signature to indicate that the duration of the sound should be equal to the time necessary for the sound "to speak"; a low, punched staccato figure, and a "ploit" sound made with the mouth alone.

These materials repeat in their order — A, B, C, D — throughout the entire composition; however, a formal scheme dictates the relative durations of the materials and the total duration of one cycle through all four materials. As the form progresses, the duration of the total cycle lengthens, until the maximum cyclic duration has been achieved. After this, a fifth material, a multiphonic trill, begins expanding during each cycle and eventually crowds out the four initial materials, claiming the entirety of the cycle's duration for itself.

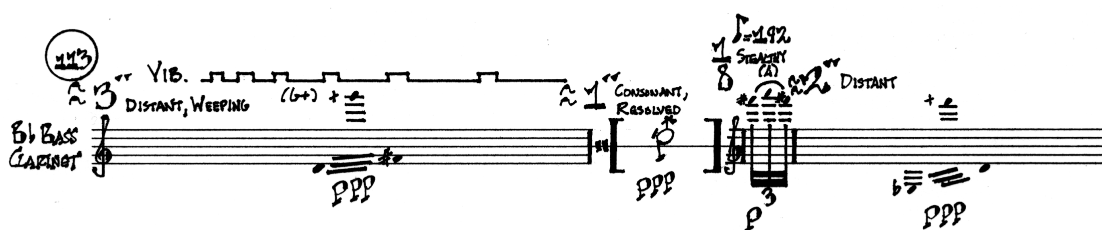


Figure 4.9: Material E has almost entirely crowded out the other materials in *Unit for Convenience and Better Living 003* (2006).

4.1.6 *Mexican Apple Soda (Consumer Affect Simulation I.1)* (2006) for Contrabass and Chamber Ensemble

Mexican Apple Soda (Consumer Affect Simulation I.1) (2006) begins with a windowing of materials similar to that of the interruption materials in the bass clarinet solo found in *Zoetropes*: every material that will make up the entire seventeen-minute composition is heard briefly, for just a few seconds, during the first twenty-five measures of the piece. After this, a three-minute contrabass solo gradually integrates two initially disparate materials. While the formalized interruption of one material by another has appeared often in previous works, this solo communicates the interruption as a gradual process: the two materials are initially heard in tact, without interruption, and begin to gradually interrupt one another more and more, until the solo reaches a condition in which a subsequent measure must be from a different

material, resulting in a terminal state of rapid alternation between the two.

Throughout the composition, unison exclamations performed on six cracklebox, an unpredictable electronic instrument invented by Michael Waisvisz, punctuate the musical order with electronic noises. In keeping with the previous mobile forms, the entire work obeys a binary form, in which the second half of the composition consists of glacial, anti-rhetorical materials, in contrast to the relatively rhetorical modality of the first section. In this sense, a formal constraint has again been applied at two levels, in the manner of the palindromic structures of *Zoetropes*: the contrabass solo integrates two materials that are introduced first as a strictly binary pair, and the large-scale form that includes this contrabass solo creates a similar binary disposition.

4.1.7 *Mexican Apple Soda Paraphrase* (2007) for Contrabass and Live Electronics

Mexican Apple Soda Paraphrase (2007) reduces the chamber concerto to a duo between pre-recorded cracklebox samples and the contrabass material from the concerto's solo. This is the work's first example of a computationally formalized model of music, executed in the graphical programming language, Max/MSP. An animated GUI directs the performer around a mobile score, replacing arbitrary choice with selection via random number generator; each time the performer is redirected, the program plays a randomly selected cracklebox sound sample, during which the performer rests. The resulting performance is a spastic intercutting of frenetic contrabass material and mercurial electronic interjection.

4.1.8 *Perfection Factory* (2008) for Two Percussonists

In *Perfection Factory* (2008), two percussionists paint a bell tree to reduce a set of over twenty pitches to a set of five pitches. This process is a pragmatic solution to the inevitable indeterminacy of a bell tree's pitches: used primarily as an effect instrument, the specific pitches of the bells vary entirely from one instrument to the next, preventing a composer from approaching the instrument with traditional pitch

notation. In response to the uniquely indeterminate quality of the instrument, a system of listening, memory, and painting creates a link between symbol and action as the score is performed, as can be seen in the score's description of the "memory notation":

§

Memory Notation.

Choose any unmarked bell in the indicated bell tree register. Rather than consistently indicating the same bell, this suggests that the player choose anew at each notehead; however, immediate repetition can occur. May or may not be accompanied by an instruction to mark the chosen bell with paint.

Same as above, and mark the chosen bell with the indicated color of paint. Play this marked bell when you see a square notehead in the same register as the one in which the selection took place / the same color as the paint with which the bell is marked.

Same as above; mark a second bell in the indicated bell tree register. Play this marked bell when you see a square notehead accompanied by a 2 in the same register as the one in which the selection took place / the same color as the paint with which the bell is marked.

Figure 4.10: Memory notation navigates between listened selection and notated pitch in *Perfection Factory* (2008) for two percussionists.

When integrated with conventional musical notation, the memory notation instructs the selection of a random bell in one of three registers of the instrument, the painting of a bell to indicate that it will correspond to a colored notehead, and the subsequent performance of the marked bell when the colored notehead reappears:

* The duration of measures with a time signature of "LV" should be judged according to the length of the indicated sound.

Figure 4.11: Colored noteheads indicate selected pitches in *Perfection Factory* (2008) for two percussionists.

This system of pitch choice gradually selects a set of five pitches from an initial set of over twenty pitches, dramatizing the process of selection with episodes

performed with the growing set of pitches. In this way, the challenge of responding to an instrument's inherent indeterminacies with a circumscribing constraint yielded an episodic formal strategy; the form's primary agenda is to expresses the selection procedure.

4.2 Installation and Visual Music, 2009—2010

Between 2008 and 2009, the author curated a series of instruction score performances in formerly abandoned places in Berlin, Germany, in collaboration with the members of the Institute for Intermediate Studies, an ensemble dedicated to the realization of past and present instruction scores; this work was a continuation of experiences working with Fluxus artists Henry Flynt and Allison Knowles in 2007 and 2008. Through this experience, the author began to consider the emergence of an entire work from an elegantly specified instruction. The first work described here was created as a contribution to one of the performances and was later presented as a contribution to the VIDEOKILLS international video festival. The subsequent computer-generated animations were created for a solo exhibition at Golden Parachutes Gallery in Berlin.

4.2.1 *Algorithmically Generated Trees* (2009)

Algorithmically Generated Trees (2009) is a generative computer animation created using Processing, a simplified version of the Java programming language created to teach artists and designers about programming (Reas and Fry 2007). A video projection algorithmically generates a cartoonish, abstract tree each frame, stopping at a specified time interval to label the tree with a number and write the image to disk. On a desk next to the projection, a sign-up sheet invites the observer to note the number of a tree found especially attractive; a rating of the color, beauty, and height of the tree on a scale from 1—5; and an e-mail address. After the exhibition, trees were e-mailed to their corresponding observers.

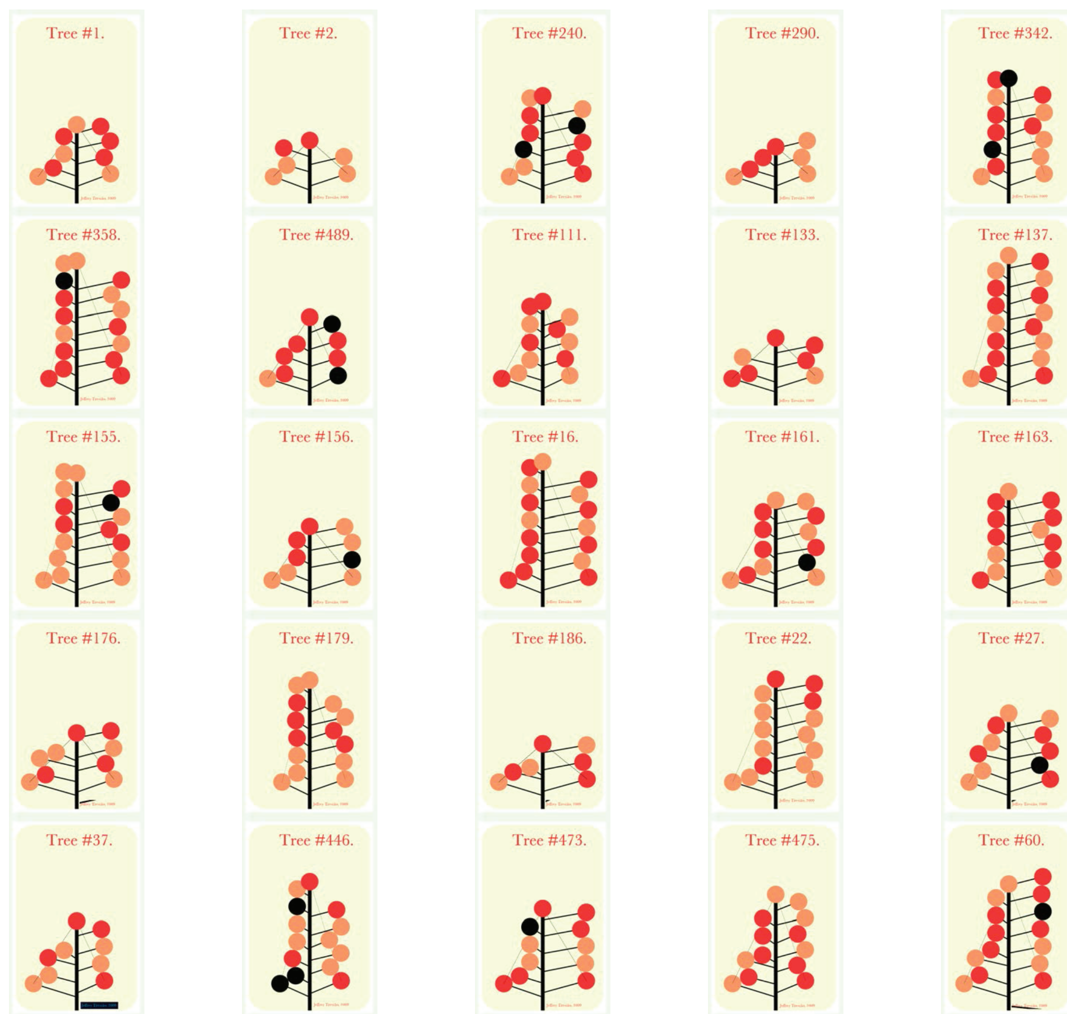


Figure 4.12: Trees generated and e-mailed to the audience in *Algorithmically Geneated Trees* (2009).

The code created for this project was made with primitive coding skills and did not take advantage of data encapsulation (A.4); however, it presages the concerns of subsequent computational works and continues the parametric agenda of the previous, non-computational work. Many variations on a single form are determined with parametrically constrained randomness, and the height of the tree, the number and angle of its branches, and the color of each leaf are determined via random number generation within tuned value boundaries.

4.2.2 *Blooms* (2010)

Blooms are three looping, abstract animations, commissioned for a gallery exhibition on the topic of ecstatic sensual experience. Departing from the religious uses of the mandala as an aid for contemplation and meditation, three programs create and gradually transform simple rotational patterns based on parametrically changing geometric figures. These works also engage a tradition of "visual music," a tradition of abstract animation that adopts the vocabulary and conceptual framework of music to create visual work. The animations were created in the Field programming environment, a hybrid of timeline and text-based coding that allows the programmer to embed breakpoint functions, sliders, and menus directly into the code, and to arrange code boxes on a canvas for time-sensitive execution according to a left-to-right timeline. The first animation is a study in nested circles, the radii of which expand and shrink gradually over time with a period of four minutes:

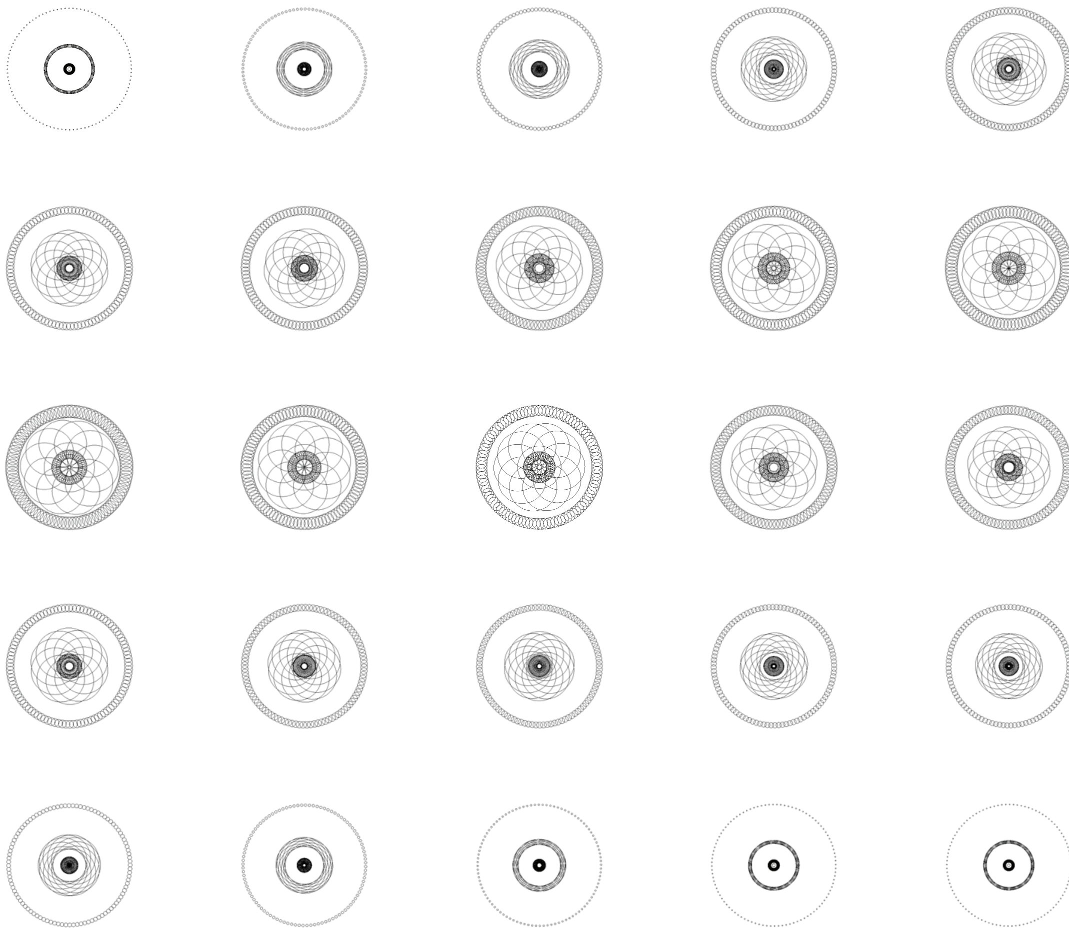


Figure 4.13: Stills captured from the rotating motion of *Bloom I* (2010).

The second animation engages music by proposing a kind of visual noise: by adding a random coordinate deviation to the endpoints of drawn ellipses, a figure distorts while maintaining to some extent its original form. The boundaries of this deviation increase and decrease along a cosine curve, resulting in a figure that gradually loses and regains its original geometric regularity.

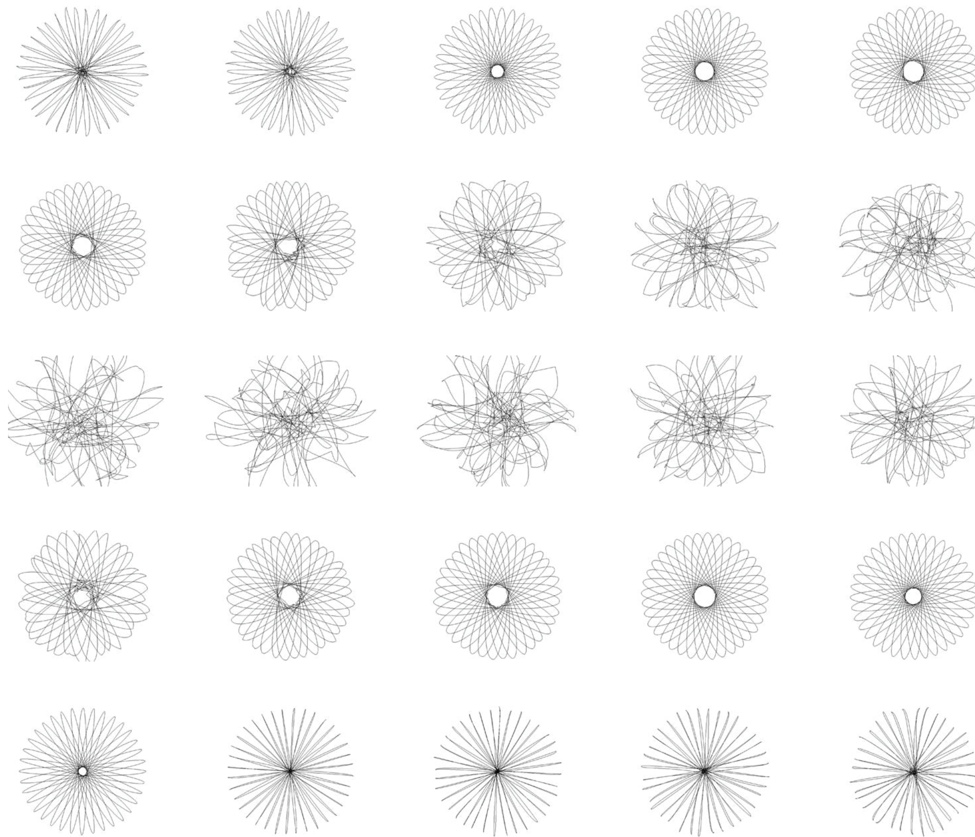


Figure 4.14: Stills captured from the rotating motion of *Bloom II* (2010).

The third animation borrows the concept of "phasing" from contemporary minimal music, and three identical forms rotate at three different rates, creating emergent patterns. The figures scintillate as the result of specifying the size of constituent elements smaller than one pixel, causing the rendering of the computationally described image to compromise at each frame on the precise location of each pixel, often rendering "L"-shaped forms instead of single pixels.

In all three works, simple instructions — nest circles inside other circles, add noise to points, rotate at a certain rate — create rich perceptual experiences, either through constrained randomness, as is the case with the precise location of drawing in the second and third animations, or through gradual changes in a simple, parametric model of an object's behavior, as in all three animations.

4.3 Computer-assisted Works, 2010—2013

4.3.1 *Being Pollen* (2010—2011) for Solo Percussion

Being Pollen, a collaboration between the author and poet Alice Notley, is a sixteen-minute work for one percussionist playing nineteen instruments and one loudspeaker. Its title is taken from Notley's poem "Pollen." The work grew from the artists' discussions about how western art music uses text: conventionally, artists import text into a musical environment and stretch its spokenness across musically quantized rhythms; this implies a process of mediation in which a poem is first assumed text, not talking. In response to history, Notley requested that archival recordings of her poems, housed in UCSD's special collections archive, be treated as extant musical voices, as sonically complete entities that need not first be taken as soundless words to become notated invitations to sound. In response to this impetus, the composer began with a curatorial phase of archival listening to select the recorded recitations for the work.

In a production chain of multiple computer programs, Abjad was used to cultivate sensitivities to the natural rhythms of the recorded poems. First, the composer slowed down the poems to half speed. Next, the composer created a program that allowed him to tap along with a poem on a laptop trackpad, recording the relative temporal relationships of all of its syllables. Having associated each syllable (or intentional breath) with an onset time in milliseconds, Audacity was used to graphically adjust temporal locations to an accuracy of one millisecond. Finally, using Josiah Oberholtzer's implementation of Paul Nauert's Q-Grids quantizer, Abjad rendered as musical notation a list of attack-times and syllables describing the recitation of Notely's poem, "Pollen." This allowed the composer to consider a detailed rhythmic representation of Notley's spoken word in the composition of the work.

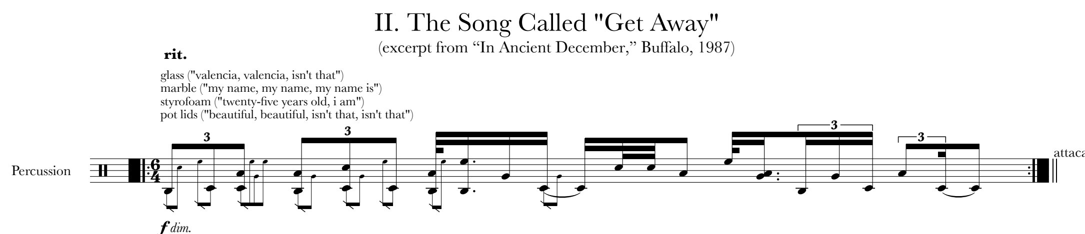
Like the alternating episodes of *Perfection Factory*, the form of the work adheres to a kind of rondo form, in which episodes of solo percussion alternate with duos between percussion and recorded recitation. Each section was realized with its own notational construct, appropriate to the relationship between percussion and recitation. The introductory section and coda relate directly to the gradual, visual

processes in *Blooms*. Percussion instruments were grouped from least to most resonant, and half-cosine interpolations executed gradual transitions from dryer to wetter sounds. At the same time, the half-cosine curves added more or less rhythmic "noise" to a steady eighth-note grid, modulating the music from pulse to unpredictably complex rhythms and back again.

Figure 4.15: Half-cosine interpolations transition from complex rhythms to pulse in *Being Pollen* (2011).

Rather than the Python programming language in Field, as used for *Blooms*, the transitions were executed in the LISP programming language with the aid of Sibelius's quantizer (Oberholtzer developed the Abjad quantizer during the composition of the work, with feedback from the author).

The second recitation's percussion accompaniment relates directly to the instruction score tradition and consists of a single measure of music — a composite of speech rhythms from the first recitation — accompanied by an instruction to repeat the music gradually more and more slowly until tempo has dissolved:



This rhythm is a composite of several rhythms from the poem, "Conversation," as described. Perform it many times, slowing gradually from the initial tempo (that of the previous section's end) to a tempo so glacially slow that no speech rhythms or meter can be perceived. Gradually soften throughout the repetitions. Let all attacks ring.

Begin playback of the eponymous poem at any time. The end of the poem need not align with the end of a repetition. The repetition that finishes immediately after the end of the recording is the end of this section of the piece.

Figure 4.16: The second recitation pairs common notation with an instruction score *Being Pollen* (2011).

The third recitation's accompaniment is entirely algorithmically generated: a timbre matching algorithm aligns percussion timbres with any syllables if there exists a spectral match of 90% or greater between a syllable and a percussion sound; if there is less similarity, the syllable is accompanied by a silence. This results in a sparse percussion texture that acts as a kind of skeleton of the speech rhythm. It is heard first without the recitation, as an arrangement of sounds in itself, and again in synchronization with the recitation.

4.3.2 +/- (2011—2012) for Twenty French Horns

+/- for twenty french horns is programmatic, naturalistic recreation of a sonic experience from the everyday environment of San Diego and an exploration of the perceptual experience of negative and positive space in the auditory domain. Inspired by the sound of driving under a highway overpass in the rain, a sudden silence must be contextualized as an event. Accordingly, the form proceeds from a state of primarily silence, with sounds grouped between large pauses, toward a state of uniformly distributed sound, to be suddenly interrupted by a single silence, and then back from a state of uniformly distributed sound toward a state of primarily silence. This formal trajectory is essentially identical to the palindromic disposition of material in *Zoetropes*. Rather than a nesting of palindromic structures, as in the trio, the focus is on the temporal distortion of the palindromic structure at a single

structural level: the path traversed into the point of reflection is the same as that traced out of the midpoint; however, the duration of the path traced out of the point of reflection has been multiplied by a factor of four to cause the same gradual change to take place over a substantially longer duration.

These naturalistic and formal agendas conspire to reduce the composition's sounds to points, to placeholders for sound rather than vivid sonic entities: the pieces uses only the sound of the palm of the hand slapped onto a french horn mouthpiece (inserted into the instrument) in order to communicate the sound/silence dichotomy elegantly and to approximate the sound of a raindrop landing on a surface. A unity of representational impetus gives way to a plurality of reference in listening, and the resulting experience demonstrates that this sound, when massed and gradually altered, evokes manifold, vivid links to everyday experience — popcorn, fireworks, gunfire, the pouring of rice or gravel — resulting in a play of reference and participation in the large-scale shape.

Like the previous works, the composition was realized using half-cosine interpolations, this time to control event densities. Via Python, Csound was used to create mock-ups of the final listening experience, and variables in the code were tuned to change global durations of a parameterized musical form; the duration of the first "leg" of the transition, into the point of palindromic reflection, and the duration of the remainder of the form were especially important.

Despite working with the Abjad API's object-oriented model of notation throughout the compositional process, it was decided upon examination of the resulting notation that an animated notation interface would be a more appropriate choice for the realization of a multi-tracked studio recording. The notation data was translated into a numeric "score" and written to a text file, which then served as an input to a sketch in the Processing coding environment, which produced a quicktime movie for each of the work's twenty parts. These animations may be used for a live performance of the work, as well, through synchronized digital tablet devices.

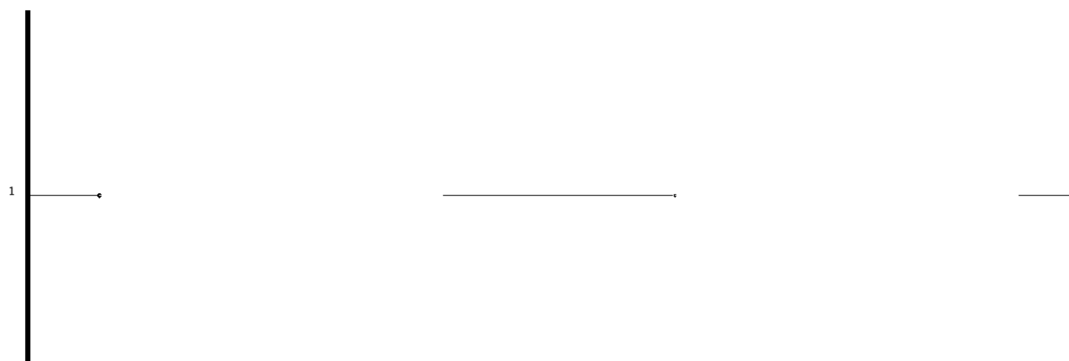


Figure 4.17: Screenshot from the animated notation parts created for +/- (2011—2012). Mouthpiece pops are indicated by points that scroll from right to left along a midline, to be performed when they cross the vertical line at the left boundary. The minimal aesthetic of the interface is inspired by early video games, such as Pong (1972).

This process revealed an unanticipated flexibility of output medium. Instead of considering the working process as the creation of a computer model of the resulting artifact (a musical notation), steered by a model of musical/compositional ideas, the link between code and artifact loosened, revealing the possibility of a model of musical notation redirected as a mapped data source for the creation of artifacts in a variety of possible media.

4.3.3 *The World All Around* (2013) for Harp, Clarinet, and Piano

Concept

The World All Around for prepared piano, Eb clarinet, and harp is a double tribute to Maurice Sendak and John Cage. It is most apparently a late contribution to the Cage centennial celebration: the piano preparations from Cage's *Sonatas and Interludes* (1946—8) have been used in a musical language closer to the later style of Cage than to the earlier style in which they were born. The piece is equally a tribute to the late Maurice Sendak, author of *Where the Wild Things Are* (Lystad 1989), the text of which contains the name of the commissioning San Francisco ensemble, Wild Rumpus.

The specific inspiration from Sendak's work here is far from a wild rumpus: the title, *The World all Around*, refers to the transformation that frames the main character's adventures, the metamorphosis of Max's room into a jungle, and then again back into a room. The return journey from the land of the wild things, in which "the world all around" becomes walls again, lasts, in the text, longer than an entire year. The composition renders this journey as a gradual transition from the "wild" timbres of the prepared piano to the unprepared sound of the concert piano; punctuating fermatas of varied lengths create a formal experience akin to a slowly paced cinematic montage, each shot of which returns to an almost unchanging scene of glacial passage.

Construction

The score was created using only the Abjad API for Formalized Score Control. Composition began by choosing beautiful multiphonic sounds in collaboration with the clarinetist. The process then consisted of the formulation of a set of random operations and constraints that would produce the three parts of the piece. First, the clarinet part was composed by creating a measure for each multiphonic sound and inserting a rest and a single pitch, taken from the bottom of the multiphonic, at random locations in each measure. The fermata over the rest would be variably chosen from four different lengths. Next, these measures were shuffled; the resulting

order is the order of measures in the clarinet part. The rhythms of the piano and harp part derive from the spacing of the rests in the clarinet part: the durations between rests in the clarinet part were shuffled to determine the sequence of durations in the piano and harp parts. All sounds are performed “laissez vibrer,” and the notation assumes 100% legato (no silence) between sounds. Silences result when the sustain of a preceding sound is shorter than the duration between a sound and its successor.

The pitches of the piano and harp parts are chosen according to a division of the entire work’s duration into four equally long formal sections, each of which specifies different constraints for the choice of interval between clarinet and harp part, on the one hand, and the possibly selected notes in the piano part, on the other; within these constraints, a harp note’s sounding octave, timbre (an octave harmonic or not), and doubling (a pitch doubled at the octave or not) are made by random choice, and the choice of the prepared piano sounds were categorized by ear into four groups of sound that move toward the unprepared sounds that form the fourth category. The harp notes are chosen by randomly choosing an interval from the list of active intervals, randomly choosing a note from a measure’s clarinet part, and measuring the interval from the selected clarinet note. All dynamics are determined via random selection without repetition. The sustain and una corda pedal positions were determined independently according to random selection informed by the previous choice.

Measurement as Form

The work is both a tribute to and a commentary on Cage’s practice: with reference to the work of Marcel Duchamp, Cage’s prepared piano instructions are reframed as a practice of measurement. Cage describes the physical location of piano preparations, not the resulting timbres, and sonorous quality has been usurped by measurement. In Duchamp’s *Three Standard Stoppages* (1913-1914), the curve of a dropped thread produces the “canned chance” of three undulating forms, three one-meter-long measures that suggest the form of rulers, of instruments of subsequent measurement. The use of measurement-determined piano timbres in a form governed itself by arbitrary correspondences of measurement and reorganization offers

a congruence of method between the organization of the work and its piano timbres.

Comments on the Code

Several aspect's of the work's code are notable. An object-oriented implementation of the WoodwindDiagram (A.1) class was needed to render the multiphonic sounds that play a central role in the work. After this, the algorithms used to execute the above construction were fairly simple. Basic math functions were of use in the piano part. Using set theoretic operations, the set of prepared piano notes was subtracted from the set of all piano notes to derive the set of all unprepared piano notes. (While laborious using basic comparison and elimination between lists, Python's built-in support for unordered collections (sets) as well as operations on them — union, disjunction, difference — made this a matter of three lines of code.) In deference to the metaphor of travel and return, the piano's preparations have been sorted into categories of proximity to the unprepared piano timbre, with the aim of gradually unpreparing the instrument's timbre throughout the form (by making the performance of an unprepared note more likely, not by physically removing the preparations from the instrument); four categories of proximity correspond to four, two-minute quarters of the eight-minute form. For each note, psuedo-random number generators select a dynamic, based on that of the previous note, and the sustain and una corda pedals are either depressed or lifted. The timing of events results from a shuffling of the durations between rests in the clarinet part; the rhythmic notation results from the placing of tie chains equal to the durations between rests, followed by functions that hide all but the first note of a tie chain. All of this is assembled on a single staff, which is then split into two staffs, forming a bracketed piano staff, using middle C as a split point. (A.7)

The creation of the harp part proceeds identically to that of the piano part, with the exception of pitch selection. While the form has still been divided into four equal parts, the selection of pitches is determined with reference to a set of intervals. For each note to be added to the harp part, a pitch class from the clarinet part is selected, within a span of seven "beats" (although the composition is rendered without meter, the program assumes a time signature of 7/4 as it calculates the work's

parts); then, an interval from the active set of intervals is added to the clarinet pitch to determine the harp pitch. The octave of the pitch class, possible doubling of the selected pitch at the octave, and whether or not a pitch is a harmonic or a traditionally plucked note, are determined by pseudo-random number generators. (A.8)

Finally, the three instrument parts are added to a score, which is contained in a LilyPondFile object. Overrides at the level of score set the space/time ratio of proportional notation, and overrides at the file level determine the layout and formatting of the completed document, such as margins, inter-system spacing, and paper size. (A.9) Finally, the program generates the formatted score as a .pdf file (B.4).

The format of the score engages in a useful trick of formatting sleight-of-hand to preserve the impression of relatively unmeasured music: the notated measure has been conflated with the system, giving the appearance of entirely unmeasured music while conceptually preserving the utility of the measure as a temporal unit in compositional choice.

Revisions

A reading session with the ensemble yielded a list of revisions to be made before the submission deadline for final scores and parts. To demonstrate the efficacy of this method of working with regard to possibilities of efficient revision, the following list enumerates the nature of each revision and the approximate amount of time required to generate a revised score that implements each specific revision:

1. The clarinet can slur only to or from the lowest pitch in a multiphonic. (12 minutes.)
2. Diaphragmatic vibrato should be much less likely in the clarinet part. (4 minutes.)
3. All single noteheads in the clarinet part should be harmonic noteheads. (10 minutes.)
4. The harp should only play two octaves above middle C or lower. (1 minute.)

5. Harp harmonics should be executed only only on strings from F in the octave above middle C and lower, and not the lowest octave of the instrument. (10 minutes.)

6. Harp notes can be converted to octaves as frequently as to harmonics. (26 minutes.)

7. The piano should only play three octaves above middle C or lower. (5 minutes.)

8. Remove the harsh multiphonics. (1 minute.)

9. Event density in the harp and clarinet parts should be doubled. (30 minutes.)

The nature of these revisions illustrates the constraint-based nature of the compositional process. As though adding detail to a sketch, additional boundary conditions accrue to limit specific aspects of the work. The small amount of time required to implement each of these revisions shows that existing formalizations can normally be retuned in order to enact new constraints, i.e. a numeric value determining the lower and upper bounds of an instrument's possible register can be altered. Sometimes a new constraint must be formalized, which takes longer than tuning an existing code variable.

The duration of the whole work, eight minutes, was specified by the ensemble in the commission. The original version of the work lasted twenty minutes when first performed in rehearsal and was reduced by changing a single variable in the code; all of the work's formal relationships scaled accordingly.

4.4 Conclusion

The introduction of computational modeling techniques has both continued and fundamentally changed the aesthetic agenda of the earlier work. Parametric approaches abound in both early and later work and offer the most obvious source of continuity. At the structural level of gesture or phrase, the explicit proposition of musical entities to be repeated and varied, whether as the cells of a mobile form in earlier work or a catalogue of shuffled and slightly differing measures in the later

work, act as a source of balance between proposition and negation. At higher structural levels, gradual change formulated as a transition between two states abounds.

Within these continuities, strategies shift. Reference to the mobile notations offers an especially stark contrast: whereas the presence of musical entities with varying parameters was formerly communicated by graphic means, leaving their navigation up to constraints on performance, formal conceit, or pseudo-random selection, the recent work replaces this graphic strategy with common notation, removing the formalization of parametric modeling from the graphic realm and relocating it in notation-generating code. The editorial capacity so fundamental to composition has been removed from the moment of performance and relocated in an iterative process of revised continuity; the design of a system for improvisation has been replaced with a more conventional idea of composition.

From the perspective of previous notations that employ a traditional staff notation, the contrast looks relatively subtle but nonetheless substantial. An earlier profligacy of graphic specification — of invented notations, meticulously specified in the notations' front-notes — has given way to a much more limited set of notational strategies. A generous reading views this change as a move toward elegance, toward the use of only what is necessary. A more critical view blames an overly restrictive computer model of notation for preventing the imaginative graphic strategies of the earlier work; however, the move to abandon conventional notation in favor of animated parts in +/- shows that this working strategy enables a flexible plethora of divergent output media in a way that drawing cannot.

The compositional ideas and formal conceits, like the notational strategies, have been similarly pruned: the application of palindromic structure in a nested, simultaneous disposition (*Zoetropes*) becomes a single structural palindrome (+/-); the hyperspecified interruption of materials (*Substitute Judgment*) gives way to a uniform conception of material, inflected by operative constraints (*The World all Around*). On the whole, there is much less contrast in the recent work, potentially because contrast does not scale well in this working process. Optimistically, the modular utility of previously written code suggests that contrast will become easier as more varieties of music have been modeled; however, this also suggests a kind of escape ve-

locity, in which the technology will afford homogeneity of material until a sufficient number of reusable code modules and a certain facility with their organization and recombination can overcome the medium's gravity.

To what extend might any of these differences be good or bad? It is difficult to uncouple the link between technology and aesthetics, because, while the technologies at play here are apparent, aesthetic priority remains a moving target, a matrix of the author's experiences, the limitations on a work, and fidelity to a programmatic inspiration. An abiding interest in programmatic agenda throughout all of the work suggests that the most meaningful evaluation for the work might be that of an affective congruence between inspiration and musical experience. This priority renders the pressure toward homogeneity of material innocuous, as long as the programme communicated is one of stasis or gradual transition.

Pragmatic evaluations certainly exist. The ability to revise the duration and proportion of an entire composition with a single variable change reduces to mere seconds a revision process that might normally take months. As importantly, the process of working iteratively, with many successively refined, complete drafts, can be much more enjoyable than the a conventional approach in which a single iteration of the work seems to occupy the entire time allotted for creation; it is at least qualitatively different.

This method of working seems also to bring the editorial capacity of the composer to the fore: when faced with multiple solutions that all represent robust solutions to specified constraints, how does one choose the best version? As architect and digital fabricator Mark Goulthorpe asserts, "Faced with only the most robust solutions to a problem, the architect nonetheless was called upon in this instance to select *that which he found most suitable*" (Goulthorpe 2011, p. 122). In this way, progress on a given work has been circumscribed as the formulation of distinctions based on the examination of multiple versions of a work that may seem initially identical. The composer discovers, rather than formulates, the identity of the work, through iterative refinement and the development of initially unknown distinctions.

Appendix A

Code Examples

A.1 Abjad Interface to Mike Solomon's LilyPond Woodwind Diagrams As a Function, Implemented with Basic String Functions

```
\scriptsize
def key_list_to_string(key_list):
    key_string = ”(”
    for key in key_list:
        key_string += (key + ” ”)
    key_string = (key_string[:-1] + ”)”)
    return key_string

def make_woodwind_diagram_markup(instrument, cc = [], lh = [], rh = []):
    key_group_strings = [ ]
    if cc:
        cc_string = ”(cc . ”
        key_string = key_list_to_string(cc)
        cc_string += (key_string + ”)”)
        key_group_strings.append(cc_string)
    else:
        cc_string = ”(cc . ())”
        key_group_strings.append(cc_string)

    if lh:
        cc_string = ”(lh . ”
        key_string = key_list_to_string(lh)
        lh_string += (key_string + ”)”)
        key_group_strings.append(lh_string)
    else:
        lh_string = ”(lh . ())”
        key_group_strings.append(lh_string)

    if rh:
        rh_string = ”(rh . ”
        key_string = key_list_to_string(rh)
        rh_string += (key_string + ”)”)
        key_group_strings.append(rh_string)
    else:
        rh_string = ”(rh . ())”
        key_group_strings.append(rh_string)

    diagram_string = r”\woodwind-diagram #’” + instrument + ” #’(”
    for key_group_string in key_group_strings:
        diagram_string += (key_group_string + ” ”)
    diagram_string = diagram_string[:-1] + ”)”
```

```
    markup = markuptools.Markup(diagram_string, direction = Up)
    return markup
}
```

A.2 Abjad Interface to Mike Solomon's LilyPond Woodwind Diagrams As a Function, Implemented with Abjad Scheme Functions

```
def make_woodwind_diagram_markup_command(instrument, cc = [], lh = [], rh = []):
    key_groups_as_scheme = [ ]
    schemed_cc_list = schemetools.Scheme(cc[:])
    schemed_lh_list = schemetools.Scheme(lh[:])
    schemed_rh_list = schemetools.Scheme(rh[:])
    if cc:
        cc_scheme_pair = schemetools.SchemePair('cc', schemed_cc_list)
        key_groups_as_scheme.append( cc_scheme_pair)
    else:
        cc_scheme_pair = schemetools.SchemePair('cc', ())
        key_groups_as_scheme.append( cc_scheme_pair)
    if lh:
        lh_scheme_pair = schemetools.SchemePair('lh', schemed_lh_list)
        key_groups_as_scheme.append( lh_scheme_pair)
    else:
        lh_scheme_pair = schemetools.SchemePair('lh', ())
        key_groups_as_scheme.append( lh_scheme_pair)
    if rh:
        rh_scheme_pair = schemetools.SchemePair('rh', schemed_rh_list)
        key_groups_as_scheme.append( rh_scheme_pair)
    else:
        rh_scheme_pair = schemetools.SchemePair('rh', ())
        key_groups_as_scheme.append( rh_scheme_pair)
    print key_groups_as_scheme
    key_groups_as_scheme = schemetools.Scheme(key_groups_as_scheme[:], quoting
        ="'")
    instrument_as_scheme = schemetools.Scheme(instrument, quoting = "'")
    return markuptools.MarkupCommand('woodwind-diagram', instrument_as_scheme,
        key_groups_as_scheme)
```

A.3 Abjad Interface to Mike Solomon's LilyPond Woodwind Diagrams As the WoodwindDiagram Class, Inheriting from Abjad's AbjadObject Abstract Class

```
from abjad.tools import schemetools
from abjad.tools import markuptools
from abjad.tools.abctools import AbjadObject


class WoodwindFingering(AbjadObject):
    r'''Abjad model of a woodwind fingering, inspired by Mike Solomon's LilyPond
        woodwind diagrams.

    Initialize from a valid instrument name and up to three keyword lists or
        tuples:

    ::

        >>> center_column = ('one', 'two', 'three', 'five')
        >>> left_hand = ('R', 'thumb')
        >>> right_hand = ('e',)
        >>> ww = instrumenttools.WoodwindFingering('clarinet', center_column =
            center_column, left_hand = left_hand, right_hand = right_hand)
        >>> ww
        WoodwindFingering('clarinet', center_column=('one', 'two', 'three', 'five
            '), left_hand=('R', 'thumb'), right_hand=('e',))

    ::

    Initialize a WoodwindFingering from another WoodwindFingering:

        >>> ww2 = instrumenttools.WoodwindFingering(ww)
        >>> ww2
        WoodwindFingering('clarinet', center_column=('one', 'two', 'three', 'five
            '), left_hand=('R', 'thumb'), right_hand=('e',))

    ::

    Call a WoodwindFingering to create a woodwind diagram MarkupCommand:

        >>> fingering_command = ww()
        >>> fingering_command
```

```
    MarkupCommand('woodwind-diagram', Scheme('clarinet'), Scheme([SchemePair
        (('cc', ('one', 'two', 'three', 'five'))), SchemePair(('lh', ('R', '
        thumb'))), SchemePair(('rh', ('e',)))]))
```

::

Attach the MarkupCommand to score components, such as a chord representing a multiphonic sound:

```
    >>> markup = markuptools.Markup(fingering_command, direction=Down)
    >>> chord = Chord("<ds' fs''>4")
    >>> markup.attach(chord)
    Markup((MarkupCommand('woodwind-diagram', Scheme('clarinet'), Scheme([
        SchemePair(('cc', ('one', 'two', 'three', 'five'))), SchemePair(('lh',
         ('R', 'thumb'))), SchemePair(('rh', ('e',)))])),), direction=Down)(<
        ds' fs''>4)
```

::

```
    >>> f(chord)
    <ds' fs''>4
        _ \markup {
                \woodwind-diagram
                        #'clarinet
                        #'((cc . (one two three five)) (lh . (R thumb)) (rh .
                            (e)))
                }
```

::

```
    >>> show(chord) # doctest: +SKIP
```

::

Initialize fingerings for eight different woodwind instruments:

```
    >>> instrument_names = ['piccolo', 'flute', 'oboe', 'clarinet', 'bass-
        clarinet', 'saxophone', 'bassoon', 'contrabassoon' ]
    >>> for name in instrument_names:
    ...    instrumenttools.WoodwindFingering(name)
    ...
    WoodwindFingering('piccolo', center_column=(), left_hand=(), right_hand=()
        )
    WoodwindFingering('flute', center_column=(), left_hand=(), right_hand=())
    WoodwindFingering('oboe', center_column=(), left_hand=(), right_hand=())
    WoodwindFingering('clarinet', center_column=(), left_hand=(), right_hand
        =())
    WoodwindFingering('bass-clarinet', center_column=(), left_hand=(),
```

```
    right_hand=())
WoodwindFingering('saxophone', center_column=(), left_hand=(), right_hand
    =())
WoodwindFingering('bassoon', center_column=(), left_hand=(), right_hand=()
    )
WoodwindFingering('contrabassoon', center_column=(), left_hand=(),
    right_hand=())
```

::

An override displays diagrams symbolically instead of graphically:

```
>>> chord = Chord("e' as' gqf''", (1,1))
>>> fingering = instrumenttools.WoodwindFingering('clarinet',
    center_column=['one', 'two', 'three', 'four'], left_hand=['R','cis'],
    right_hand=['fis'])
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand('override', schemetools.
    SchemePair('graphical', False))
>>> markup = markuptools.Markup([not_graphical, diagram], direction=Down)
>>> markup.attach(chord)
Markup((MarkupCommand('override', SchemePair(('graphical', False))),
    MarkupCommand('woodwind-diagram', Scheme('clarinet'), Scheme([
    SchemePair(('cc', ('one', 'two', 'three', 'four'))), SchemePair(('lh',
     ('R', 'cis'))), SchemePair(('rh', ('fis',)))]))), direction=Down)(<e'
     as' gqf''>1)
```

::

```
>>> f(chord)
<e' as' gqf''>1
        _ \markup {
                \override
                        #'(graphical . #f)
                \woodwind-diagram
                        #'clarinet
                        #'((cc . (one two three four)) (lh . (R cis)) (rh
                            . (fis)))
                }
```

::

```
>>> show(chord) # doctest: +SKIP
```

::

The thickness and size of diagrams can also be changed with overrides:

```
    >>> chord = Chord("e' as' gqf''", (1,1))
    >>> fingering = instrumenttools.WoodwindFingering('clarinet',
        center_column=['one', 'two', 'three', 'four'], left_hand=['R','cis'],
        right_hand=['fis'])
    >>> diagram = fingering()
    >>> not_graphical = markuptools.MarkupCommand('override', schemetools.
        SchemePair('graphical', False))
    >>> size = markuptools.MarkupCommand('override', schemetools.SchemePair('
        size', .5))
    >>> thickness = markuptools.MarkupCommand('override', schemetools.
        SchemePair('thickness', .4))
    >>> markup = markuptools.Markup([not_graphical, size, thickness, diagram],
         direction=Down)
    >>> markup.attach(chord)
    Markup((MarkupCommand('override', SchemePair(('graphical', False))),
        MarkupCommand('override', SchemePair(('size', 0.5))), MarkupCommand('
        override', SchemePair(('thickness', 0.4))), MarkupCommand('woodwind-
        diagram', Scheme('clarinet'), Scheme([SchemePair(('cc', ('one', 'two',
         'three', 'four'))), SchemePair(('lh', ('R', 'cis'))), SchemePair(('rh
        ', ('fis',)))]))), direction=Down)(<e' as' gqf''>1)

::

    >>> f(chord)
    <e' as' gqf''>1
            _ \markup {
                    \override
                            #'(graphical . #f)
                    \override
                            #'(size . 0.5)
                    \override
                            #'(thickness . 0.4)
                    \woodwind-diagram
                            #'clarinet
                            #'((cc . (one two three four)) (lh . (R cis)) (rh
                                . (fis)))
                    }

::

    >>> show(chord) # doctest: +SKIP

::

Return woodwind fingering.

'''
```

```
### INITIALIZER ###

def __init__(self, arg, center_column=None, left_hand=None, right_hand=None):
    assert isinstance(center_column, (type(None), list, tuple))
    assert isinstance(left_hand, (type(None), list, tuple))
    assert isinstance(right_hand, (type(None), list, tuple))
    #initialize from a string and up to three lists:
    if isinstance(arg, str):
        assert arg in self._valid_instrument_names
        self._instrument_name = arg
        if center_column is None:
            self._center_column = ()
        else:
            self._center_column = tuple(center_column)
        if left_hand is None:
            self._left_hand = ()
        else:
            self._left_hand = tuple(left_hand)
        if right_hand is None:
            self._right_hand = ()
        else:
            self._right_hand = tuple(right_hand)
    #initialize from a WoodwindDiagram with up to three overriding lists:
    elif isinstance(arg, type(self)):
        self._instrument_name = arg.instrument_name
        self._center_column = arg.center_column
        self._left_hand = arg.left_hand
        self._right_hand = arg.right_hand
        if center_column is not None:
            self._center_column = tuple(center_column)
        if left_hand is not None:
            self._left_hand = tuple(left_hand)
        if right_hand is not None:
            self._right_hand = tuple(right_hand)

### PRIVATE READ-ONLY PROPERTIES ###

@property
def _positional_argument_names(self):
    return ('_instrument_name',)

@property
def _valid_instrument_names(self):
    return ('piccolo', 'flute', 'oboe', 'clarinet', 'bass-clarinet', '
        saxophone', 'bassoon', 'contrabassoon')

 ### READ-ONLY PUBLIC PROPERTIES ###
```

```
@property
def center_column(self):
    r'''Read-only tuple of contents of key strings in center column key group:

::

    >>> ww.center_column
    ('one', 'two', 'three', 'five')

Return tuple.
    '''
    return self._center_column

@property
def instrument_name(self):
    r'''Read-only string of valid woodwind instrument name:

::

    >>> ww.instrument_name
    'clarinet'

Return string.
    '''
    return self._instrument_name

@property
def left_hand(self):
    r'''Read-only tuple of contents of key strings in left hand key group:

::

    >>> ww.left_hand
    ('R', 'thumb')

Return tuple.
    '''
    return self._left_hand

@property
def right_hand(self):
    r'''Read-only tuple of contents of key strings in right hand key group:

::

    >>> ww.right_hand
    ('e',)
```

```
Return tuple.
    '''
    return self._right_hand

### OVERRIDES ###

def __call__(self):
    key_groups_as_scheme = [ ]
    cc_scheme_pair = schemetools.SchemePair('cc', self._center_column)
    key_groups_as_scheme.append( cc_scheme_pair)
    lh_scheme_pair = schemetools.SchemePair('lh', self._left_hand)
    key_groups_as_scheme.append( lh_scheme_pair)
    rh_scheme_pair = schemetools.SchemePair('rh', self._right_hand)
    key_groups_as_scheme.append( rh_scheme_pair)
    key_groups_as_scheme = schemetools.Scheme(key_groups_as_scheme[:], quoting
        ="'")
    instrument_as_scheme = schemetools.Scheme(self._instrument_name, quoting =
         "'")
    return markuptools.MarkupCommand('woodwind-diagram', instrument_as_scheme,
         key_groups_as_scheme)

###PUBLIC METHODS###

def print_guide(self):
    r'''Print read-only string containing instrument's valid key strings,
        instrument diagram, and syntax explanation.

::

Return string.
'''
    if self._instrument_name == 'clarinet':
        lines = [
        'list of valid key strings for clarinet:',
        '',
        'cc',
        'possibilities for one:',
        '(one oneT one1qT oneT1q one1q one1qT1h one1hT1q one1qT3q one3qT1q
            one1qTF oneFT1q one1hT oneT1h one1h one1hT3q one3qT1h one1hTF
            oneFT1h one3qT oneT3q one3q one3qTF oneFT3q oneFT oneF)',
        'possibilities for two:',
        '(two twoT two1qT twoT1q two1q two1qT1h two1hT1q two1qT3q two3qT1q
            two1qTF twoFT1q two1hT twoT1h two1h two1hT3q two3qT1h two1hTF
            twoFT1h two3qT twoT3q two3q two3qTF twoFT3q twoFT twoF)',
        'possibilities for three:',
        '(three threeT three1qT threeT1q three1q three1qT1h three1hT1q
            three1qT3q three3qT1q three1qTF threeFT1q three1hT threeT1h
```

```
    three1h three1hT3q three3qT1h three1hTF threeFT1h three3qT
    threeT3q three3q three3qTF threeFT3q threeFT threeF)',
'possibilities for four:',
'(four fourT four1qT fourT1q four1q four1qT1h four1hT1q four1qT3q
    four3qT1q four1qTF fourFT1q four1hT fourT1h four1h four1hT3q
    four3qT1h four1hTF fourFT1h four3qT fourT3q four3q four3qTF
    fourFT3q fourFT fourF)',
'possibilities for five:',
'(five fiveT five1qT fiveT1q five1q five1qT1h five1hT1q five1qT3q
    five3qT1q five1qTF fiveFT1q five1hT fiveT1h five1h five1hT3q
    five3qT1h five1hTF fiveFT1h five3qT fiveT3q five3q five3qTF
    fiveFT3q fiveFT fiveF)',
'possibilities for six:',
'(six sixT six1qT sixT1q six1q six1qT1h six1hT1q six1qT3q six3qT1q
    six1qTF sixFT1q six1hT sixT1h six1h six1hT3q six3qT1h six1hTF
    sixFT1h six3qT sixT3q six3q six3qTF sixFT3q sixFT sixF)',
'possibilities for h:',
'(h hT h1qT hT1q h1q h1qT1h h1hT1q h1qT3q h3qT1q h1qTF hFT1q h1hT hT1h
     h1h h1hT3q h3qT1h h1hTF hFT1h h3qT hT3q h3q h3qTF hFT3q hFT hF)',
'',
'lh',
'possibilities for thumb:',
'(thumb thumbT)',
'possibilities for R:','(R RT)',
'possibilities for a:',
'(a aT)',
'possibilities for gis:',
'(gis gisT)',
'possibilities for ees:',
'(ees eesT)',
'possibilities for cis:',
'(cis cisT)',
'possibilities for f:',
'(f fT)',
'possibilities for e:',
'(e eT)',
'possibilities for fis:','(fis fisT)',
'',
'rh',
'possibilities for one:',
'(one oneT)',
'possibilities for two:',
'(two twoT)',
'possibilities for three:',
'(three threeT)',
'possibilities for four:',
'(four fourT)',
'possibilities for b:',
```

```
'(b bT)',
'possibilities for fis:',
'(fis fisT)',
'possibilities for gis:',
'(gis gisT)',
'possibilities for e:',
'(e eT)',
'possibilities for f:',
'(f fT)',
'',
'diagram syntax',
'',
'   Lilypond woodwind diagram syntax divides an instrument into
    keyholes and keys.',
'   Keyholes belong to the central column (cc) group.',
'   Keys belong to either left-hand (lh) or right-hand (rh) groups.',
"   In Abjad's diagrams, central column (cc) keyholes appear along a
    central dotted line.",
'   Keys are grouped relative to the presence or absence of a dividing
     horizontal line:',
'   If a horizontal line divides a side of the diagram,',
'   keys above the line are left-hand keys (lh),',
'   and those below are right-hand keys (rh).',
'   If no horizontal line appears, all keys on that side of the
    diagram are left-hand keys (lh).',
'   A key located along the central dotted line will be grouped',
'   according to the playing hand of the nearest keyhole fingers.',
'',
'   To draw half- or quarter-covered keys, and to draw trills,',
'   refer to the comprehensive list of possible key strings that
    precedes this explanation.',
'',
'',
'             a  gis',
'    R        |',
'            one',
'    thumb    h',
'            two',
'             |  ees',
' --------- three',
'             |',
'         one | cis',
'         two |    f',
'       three | e',
'        four |    fis',
'             |',
'            four',
'             |',
```

```
'             five',
'            b |',
'             six',
'       fis    |',
'          gis |',
'       e      |',
'           f  |',
'',
'    clarinet',
'    as modeled in LilyPond by Mike Solomon',
'    diagram explanation and key string index above',
''
]
for line in lines:
    print line
```

A.4 Processing Code for *Algorithmically Generated Trees*

```
//Darling Clementine Haberdashery Series - Personal, 2006 (Gestalten's book
    _Naive_, copyright 2009)
//Code by Jeff Trevino, June 29, 2009.  Version 8.
//Program that generates trees randomly on the left. A clock on the right draws a
    randomly placed red line whenever the the number of minutes is divisible by
    two, and erases the red line when the second hand meets the randomly drawn
    line.

//include libraries.
import processing.video.*;
import maxlink.*;

//random variables.
int seed = millis();

//camera input.
//Capture cam;

//make a movie.
//MovieMaker mm;

//text
PFont theFont;

//color variables.
```

```
color backPaper = color(240,255,238);
color innerBackYellow = color(250,255,220);
color ellipseRed =color(255,51,50);
color ellipseRedLight=color(255,150,100);

//frame variables.
float outerFrameThickX = 10;
float outerFrameThickY = 14;
float innerFrameThickX = 10;
float innerFrameThickY = 10;



//tree variables.
float bottomRand = 50; //range from which the y coord of the low leaf gets picked.
float trunkRand = 400; //range of the length of the trunk (pixels).
float radius = 26.0; //radius of the ellipses.
int trunkStrokeWeight = 11; //how thick the trunk is.
int branchStrokeWeight = 3; //how thick the branches are.
int lowestBranch = 40; //how high up the lowest branch will be drawn.
int lowBranchSmallY = 40; //slope of the lowest branch.
float lowLeafXrange = 140; //the range from which the x coordinate of the lowest
    leaf gets picked.
int lowLeafYrange= 80;//the range fromw which the y coordinate of the lowest leaf
    gets picked.
int climbMin = 10; //the minimum climb between branches.
float angleRange = 45.;
int climbRange = 4; //the range of randomly chosen additional distance between
    branches. calculated as subtraction from the radius to the trunk or leftBound.
int branchMin = 20; //The minimum downward slope in pixels of any branch with
    relation to its leaf.
float branchRange = radius/3.; //the amount of variation in the Y axis for each
    branch.
float fromBranch = 4.; //the number of pixels away from a branch a leaf must be.

//tree selection system variables.
int oldSecond;
int newSecond;
float j = 0;
int redLineDrawn = 0;
int redThetaPicked = 0;
float redLineTheta = 0;
int isStopped = 0;
int numStops = 0;
int redLineWait = 60;

//maxLink variables
```

```
int zeroSent = 0;
int oneSent = 0;
int twoSent = 0;

//draw the tree.

void setup(){
  size(800,600);
  background(backPaper);

  randomSeed(seed);

  //setup camera input.
  //String[] devices = Capture.list();
// println(devices);
  //cam= new Capture(this,640,480,devices[0],30);

  //setup the movie file.
 // mm = new MovieMaker(this, width, height, ”drawing.mov”, 5, MovieMaker.RAW,
     MovieMaker.LOSSLESS);

  //setup fonts.
  theFont = loadFont(”Baskerville-48.vlw”);

//the frame, made of an overlapping white and yellow rectangle.


  }

  /**********************
   DRAW LOOP.
  *********************/

    void draw() {


if (isStopped == 1) {
  save(”tree”+numStops+”.tiff”);
  link.output(0);
  for (int i = 0; i < 16; i++) {
  //mm.addFrame();
  }
  delay(15000);
  isStopped = 0;
}
link.output(1);
background(backPaper);
int seed = millis();
```

```
smooth();
 randomSeed(seed);
//the frame, made of an overlapping white and yellow rectangle.
strokeJoin(ROUND);
rectMode(CENTER);
fill(255);
noStroke();
rect(width/4,height/2,width/2-outerFrameThickX*2,height-outerFrameThickY*2);
fill(innerBackYellow);
stroke(innerBackYellow);
strokeWeight(100);
strokeJoin(ROUND);
float frameWidth = width/2-outerFrameThickX*2-innerFrameThickX*2-100;
float frameHeight = height-outerFrameThickY*2-innerFrameThickY*2-100;
rect(width/4,height/2,frameWidth,frameHeight);
float frameBaseY = height - outerFrameThickY-innerFrameThickY;

//draw the other side of the frame, too.
noStroke();
fill(255);
rect(width - width/4,height/2,width/2-outerFrameThickX*2,height-outerFrameThickY
    *2);
fill(innerBackYellow);
stroke(innerBackYellow);
strokeWeight(100);
strokeJoin(ROUND);
rect(width - width/4,height/2,frameWidth,frameHeight);
//signature.
 textFont(theFont,12);
 fill(ellipseRed);
 text(”Jeffrey Trevi□o, 2009”,width - 178, height/2 + 4);

//clock.
float clockMidX = width-width/4;
float clockMidY = height/2;

//put the video in the clock’s background.
//if (cam.available() == true) {
 // cam.read();
 // imageMode(CENTER);
 // image(cam,clockMidX,clockMidY,320,240);
//}


//draw the middle of the clock.
fill(ellipseRed);
noStroke();
strokeWeight(3);
```

```
ellipse(clockMidX,clockMidY,15,15);

//draw a line that’s the hand.

float clockTheta = 0.0;
float handX = 0.0;
float handY = 0.0;
clockTheta = (360/60)*second();
handX = 150*cos(radians(clockTheta))+clockMidX;
handY = 150 *sin(radians(clockTheta))+clockMidY;
j = millis()/1000;

//if the red line isn’t there yet, and you didn’t pick theta yet, then pick theta.
if (redLineDrawn == 0) {
  if (redThetaPicked == 0) {
    redLineTheta = randInt(0,59)*6;
    redThetaPicked = 1;
  }
//if the number of seconds is divisible by a certain amount, draw a red line
    wherever.
  if (j % redLineWait == 0 ) {
     stroke(ellipseRed);
     strokeWeight(4);
     line(clockMidX,clockMidY,150*cos(radians(redLineTheta))+clockMidX,150*sin(
         radians(redLineTheta))+clockMidY);
     redLineDrawn = 1;
  }
} else {
//when j isn’t divisible by a certain amount, just keep redrawing clock, unless...
     stroke(ellipseRed);
     strokeWeight(4);
     line(clockMidX,clockMidY,150*cos(radians(redLineTheta))+clockMidX,150*sin(
         radians(redLineTheta))+clockMidY);
//the clock hand crosses the red line - in which case, save the tree, ask for a
    sign up, and pause.
     if (clockTheta == redLineTheta) {
       stroke(0);
       strokeWeight(4);
       line(clockMidX,clockMidY,handX,handY);
       textFont(theFont,48);
       fill(ellipseRed);
       numStops++;
       save(”tree”+numStops+”.tiff”);
       text(”Tree #”+numStops+”.”,width/4-90,85);
       textFont(theFont,30);
       text(”If you would like this tree,”,width-width/2+40, 130);
       text(”please sign up on the sheet.”,width-width/2+35, height-130);
       redLineDrawn = 0;
```

```
        redThetaPicked = 0;
        isStopped = 1;
      }
}

//  stroke(0);
//strokeWeight(4);
//line(clockMidX,clockMidY,handX,handY);
//}






stroke(0);
strokeWeight(4);
line(clockMidX,clockMidY,handX,handY);

//tree selection system.






//other width/height - based variables.

float lowestBranchOut = width /4.4; //how far does the lowest branch stick out to
    the left?
float fromTrunk = width / 1.; // how close to trunk can the leaves be?


//the trunk.
stroke(0);
strokeWeight(trunkStrokeWeight);
strokeCap(SQUARE);
float trunkTop = outerFrameThickY+innerFrameThickY;
int trunkShorter =randInt(100.,trunkRand);
trunkTop = trunkTop+trunkShorter;
line(width/4,frameBaseY,width/4,trunkTop);
float trunkLength = frameBaseY-trunkTop;
float lengthLeft = trunkLength;


//-- THE LEAVES AND BRANCHES --
//Draw the lowest branch/leaf on the left.
ellipseMode(RADIUS);
```

```
float lowBranchX = width/4;
float lowBranchY = frameBaseY - lowestBranch;
float lowLeafX = width/4 - randInt(80,lowLeafXrange);
float lowLeafY = lowBranchY - lowBranchSmallY;
strokeWeight(branchStrokeWeight);
line(lowBranchX,lowBranchY,lowLeafX,lowLeafY);
noStroke();
fill(chooseColor());
ellipse(lowLeafX,lowLeafY,radius,radius);

//set up the variables that the while loop needs.
float leafBound = lowLeafX;
float distanceRadius = 0.;
float leafX = lowLeafX;
float leafY = lowLeafY;
float oneAgoLeafX = lowLeafX;
float oneAgoLeafY = lowLeafY;
float twoAgoLeafX = lowLeafX;
float twoAgoLeafY = lowLeafY;
float oneAgoBranchX = 0.;
float oneAgoBranchY = 0.;
float twoAgoBranchX = 0.;
float twoAgoBranchY = 0.;
lengthLeft = lowLeafY;

//Draw the lowest branch/leaf on the right.
leafY = leafY - radius/3.;
leafX = width/4 + randInt(80,lowLeafXrange);
float branchY = lowBranchY - climbMin - randInt(0,climbRange);
float branchX = width /4;
stroke(branchStrokeWeight);
line(branchX,branchY,leafX,leafY);
noStroke();
fill(chooseColor());
ellipse(leafX,leafY,radius,radius);

//set up the guidelines between the top of the trunk and the left/right branch).
float slopeLeft = 0.0;
float slopeRight = 0.0;
float bLeft = 0.0;
float bRight = 0.0;

slopeLeft = ((trunkTop - lowLeafY) / (width/4 - lowLeafX));
slopeRight = ((leafY - trunkTop) / (leafX - width/4));
bLeft = lowLeafY - slopeLeft*lowLeafX;
bRight = leafY - slopeRight*leafX;

float y= 0.0;
```

```
//draw the guides to check them.
for (float i = lowLeafX; i < width/4; i++) {
  y = slopeLeft*i + bLeft;
  stroke(0);
  strokeWeight(1);
  point(i,y);
}

for (int i = width/4; i <= leafX; i++) {
  y = slopeRight*i + bRight;
  stroke(0);
  strokeWeight(1);
  point(i,y);
}
//println("slopeRight is "+slopeRight);


//update the variables for the while loop that will draw the rest of the leaves.
float leafBoundRight = leafX;
lengthLeft = lengthLeft - (lengthLeft - leafY);
oneAgoLeafX = leafX;
oneAgoLeafY = leafY;
oneAgoBranchX = branchX;
oneAgoBranchY = branchY;
twoAgoBranchX = lowBranchX;
twoAgoBranchY = lowBranchY;

//draw the rest of the leaves up the trunk.
int tooShort = 0;
int i = 0;
//println("BEFORE THE LOOP, LENGTH LEFT IS "+lengthLeft);
while (lengthLeft > trunkTop+radius) {
  //use modulo to track which side the branch is on (important for boundary
      corrections).
  int side = (i+2)%2;
  //use modulo to alternate the leaves from left to right of one another along
      each side.)
  int angleDir = (i+4)%4;
  //println("Side is "+side);
  //println(" - CYCLE "+i+" _");
  //println("At the beginning of the cycle, the remaining trunk lenght is"+
      lengthLeft);

  //pick where the ellipse goes.
  //since you know you're starting from the leafBounds (the lower leaves' x values
      ), you can pick theta first.
    float theta = 0.0;
```

```
    if (side == 0) {
   if (angleDir == 0){
     theta = randInt(1,(90-angleRange));
   }
   if (angleDir == 2) {
     theta = randInt(91,(91+angleRange));
   }
  }
  if (side == 1) {
   if (angleDir == 1){
     theta = randInt(91,(91+angleRange));
   }
   if (angleDir == 3) {
     theta = randInt(1,(89-angleRange));
   }
  }
  float distanceRadiusMax = findDistanceRadiusMax(twoAgoLeafX,leafBound,
      leafBoundRight, fromTrunk,climbRange,radius,side,theta);
  distanceRadius = randInt(2*radius,distanceRadiusMax);
  //println("From a radial range of "+distanceRadiusMax+", the program has chosen
      to draw the next leaf "+distanceRadius+"away.");
  //float angleRange = findMinTheta(twoAgoLeafX,twoAgoLeafY,twoAgoBranchX,
      twoAgoBranchY,distanceRadius,radius,side);
  //pick the angle between the former leaf-trunk line and the former leaf-new leaf
       line.


  //println("Theta is "+theta);
//  float radiusMin = findRadiusMin(theta,twoAgoBranchX,twoAgoBranchY,twoAgoLeafX,
    twoAgoLeafY,side);
  //println("The distance to the new leaf is "+distanceRadius);
  //use the angle and the radius to calculate the (x,y) of the new leaf.
  float changeX = distanceRadius*cos(radians(theta));
  float changeY = distanceRadius*sin(radians(theta));
  leafX = twoAgoLeafX + changeX;
  leafY = twoAgoLeafY - changeY;

  //check to see if the new values collide with a branch, and make theta larger if
       this is the case.
  float slope = 0.0;
  float b = 0.0;
  if (side == 0 ){
    slope = ((twoAgoBranchY - twoAgoLeafY) / (twoAgoBranchX - twoAgoLeafX));
  }
  if (side == 1) {
    slope = ((twoAgoBranchY - twoAgoLeafY) / (twoAgoLeafX - twoAgoBranchX));
  }
  //println("The equation of the branch is y="+slope+"x +"+b);
```

```
b = twoAgoBranchY - slope*twoAgoBranchX;
while (leafY+radius >= (slope*leafX + b - fromBranch)) {
  //println("leafY = "+leafY+" and is less than" +(slope*leafX + b)+".");
  if (side == 0) {
  theta++;
  }
  if (side == 1) {
    theta--;
  }
  //println("theta got incremeneted.");
  changeX = distanceRadius*cos(radians(theta));
  changeY = distanceRadius*sin(radians(theta));
  leafX = twoAgoLeafX + changeX;
  leafY = twoAgoLeafY - changeY;
  //println("After adjustment, leafY+radius is "+(leafY+radius)+".");
 // stroke(0);
 // strokeWeight(2);
 // line(leafBound,0,leafBound, height);
}
//check to see if values collide with leafBound, and change theta appropriately.
if (side == 0) {
  while (leafX <= leafBound) {
    theta--;
    changeX = distanceRadius*cos(radians(theta));
    changeY = distanceRadius*sin(radians(theta));
    leafX = twoAgoLeafX + changeX;
    leafY = twoAgoLeafY - changeY;
    stroke(0);
  }
}
if (side == 1) {
  while (leafX >= leafBoundRight) {
    theta++;
    changeX = distanceRadius*cos(radians(theta));
    changeY = distanceRadius*sin(radians(theta));
    leafX = twoAgoLeafX + changeX;
    leafY = twoAgoLeafY - changeY;
     line(leafY+radius,leafBoundRight,leafY-radius,leafBoundRight);
  }
}

//check to see if values collide with the trunk, and change theta appropriately.
if (side == 0) {
  while (leafX+radius >= width/4. - trunkStrokeWeight) {
    theta++;
    changeX = distanceRadius*cos(radians(theta));
    changeY = distanceRadius*sin(radians(theta));
    leafX = twoAgoLeafX + changeX;
```

```
        leafY = twoAgoLeafY - changeY;
      }
    }
    if (side == 1) {
      while (leafX-radius <= width/4. + trunkStrokeWeight) {
        theta--;
         changeX = distanceRadius*cos(radians(theta));
        changeY = distanceRadius*sin(radians(theta));
        leafX = twoAgoLeafX + changeX;
        leafY = twoAgoLeafY - changeY;
      }
    }


    //println("The first circle was at ("+twoAgoLeafX+","+twoAgoLeafY+"), and the
        second circle was drawn "+changeX+" to the right and "+changeY+" up, at ("+
        leafX+","+leafY+").");

    //Choose where the branch meets the trunk.
    branchX = width / 4.;
    branchY = leafY + branchMin + randInt(0,branchRange);
    //Draw the branch.
    stroke(0);
    strokeWeight(branchStrokeWeight);
    line(branchX,branchY,leafX,leafY);
    //Draw the new leaf.
    noStroke();
    fill(chooseColor());
    ellipse(leafX,leafY,radius,radius);
    //update the variables for the next cycle;
    lengthLeft = leafY;
    twoAgoLeafX = oneAgoLeafX;
    twoAgoLeafY = oneAgoLeafY;
    oneAgoLeafX=leafX;
    oneAgoLeafY=leafY;
    //println("At the end of the cycle, the reminaing trunk length is "+lengthLeft);
    i++;
    }
  fill(chooseColor());
  ellipse(width/4.,trunkTop,radius,radius);
  delay(100);
  oldSecond = second();
    //
   // leafY = chooseLeafY(leafY);
   //pick where the branch goes.
    //draw the branch.
    //draw the ellipse.
    //change the side counter.
```

```
   // if (sideOverride() == 0 )
  //update the length left.
  //update twoAgo.

//tree selection system.
//mm.addFrame();
}
//FIN.





//FUNCTIONS.

//a function that returns a random integer, by scaling the float result of random
    ().
int randInt(float low,float high) {
  float randLow = low/100000;
  float randHigh = high/100000;
  float randOut= random(randLow,randHigh);
  int result = int(100000*randOut);
  return result;
}

//a function that chooses between two colors randomly - with the off chance that
    it might choose black.
color chooseColor() {
  int choice = randInt(0,2);
  if (choice == 0) {
    if (isBlack() == 1) {
      return 0;
    } else {
    return ellipseRed;
    }
  } else {
    return ellipseRedLight;
  }
}

//isBlack() chooses whether or not a leaf will be black - it will be rare!
int isBlack() {
  int choice = randInt(0,10);
  if (choice > 8) {
    return 1;
  } else {
    return 0;
```

```
  }
}
//returns: how long the radius can be without hitting the trunk or going out of
    bounds.
int findDistanceRadiusMax(float twoAgoLeafX, float leafBound, float leafBoundRight
    , float fromTrunk, float climbRange, float radius, int side, float theta) {
  float toTrunk = 0.;
  float toLeafBound = 0.;
  float climb = randInt(0,climbRange);
  if (side == 0) {
    if (theta > 90 ) {
    toLeafBound = twoAgoLeafX - leafBound - radius;
    //println("findRadiusRange returned a radius, to leafBound, of "+toLeafBound);
    return int(toLeafBound);
    }
    if (theta < 90) {
      toTrunk = width / 4 - twoAgoLeafX - fromTrunk - radius - trunkStrokeWeight;
      //println("findRadiusRange returned a radius, to the trunk, of "+toTrunk);
      return int(toTrunk);
    }
  }
  if (side == 1) {
    if (theta > 90) {
    toTrunk = twoAgoLeafX - width / 4 - fromTrunk - radius - trunkStrokeWeight;
    //println ("findRadius returned a radius, to the trunk, of "+toTrunk);
    return int(toTrunk);
    }
    if (theta < 90) {
    toLeafBound = leafBoundRight - twoAgoLeafX - radius;
    //println ("findRadius returned a radius, to leafBound, of "+toLeafBound);
    return int(toLeafBound);
    }

  return 0;
  }
  return 0;
}
```

-

A.5 Catalogue of Possible Entrances Into and Exits from Clarinet Multiphonics

```
import os
from abjad import *
```

```
from WoodwindFingering import *

multiphonics = {'cresc':
                    {2: (pitchtools.NamedChromaticPitchSet( ["ds'", "fs''"] ),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'five'), left_hand = ('R', 'thumb'),
                            right_hand = ('e',) ), '' ),
                    3: (pitchtools.NamedChromaticPitchSet( ["f'", "gs''"] ),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'four', 'five'), left_hand = ('R', 'thumb') ), '
                            prominent gs' ),
                    4: (pitchtools.NamedChromaticPitchSet( ["c'", "ds''", "a''"] )
                        ,
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'four', 'five', 'six'), left_hand = ('
                            thumb',), right_hand=('f',)), 'dissonant/expressive'),
                    5: (pitchtools.NamedChromaticPitchSet( [ "c'", "e''", "as''"
                        ]),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'four', 'five', 'six'), left_hand = ('R',
                            'cis'), right_hand=('fis',)), 'dissonant/expressive'),
                    7: (pitchtools.NamedChromaticPitchSet( [ "d'", "f''", "b''" ]
                        ),
                        WoodwindFingering('clarinet', center_column = ('one', '
                            three', 'four', 'five', 'six'), left_hand = ('R',)), '
                            intense/long cresc.'),
                    8: (pitchtools.NamedChromaticPitchSet( [ "d'", "fs''", "cqf
                        '''" ] ),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'four', 'five', 'six'), left_hand = ('R',)
                            , right_hand = ('gis', 'four')), 'comes out easy,
                            almost no cresc.'),
                    },
                'soft':
                    {1: (pitchtools.NamedChromaticPitchSet( [ "e'", "as'", "gqf''"
                         ] ),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'four'), left_hand = ('R', 'thumb', 'cis')
                            , right_hand = ('fis',)), ''),
                    2: (pitchtools.NamedChromaticPitchSet( [ "e'", "bf'", "g''" ]
                        ),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'five', 'six'), left_hand = ('R', 'thumb',
                             'cis')), ''),
                    6: (pitchtools.NamedChromaticPitchSet( [ "f'", "c''", "aqf''"
                        ] ),
                        WoodwindFingering('clarinet', center_column = ('one', 'two
                            ', 'three', 'five', 'six'), left_hand = ('R', 'thumb
```

```
        ',), right_hand = ('gis',)), ''),
    8: (pitchtools.NamedChromaticPitchSet( [ "eqs'", "dqf''", "aqs
        ''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'four', 'five', 'six'), left_hand = ('R', 'thumb',)
            , right_hand = ('f',)), ''),
    },
'diad':
    {1: (pitchtools.NamedChromaticPitchSet( [ "ds'", "c''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'four', 'five', 'six'), left_hand = ('R', 'thumb',)
            , right_hand = ('f',)), ''),
    6: (pitchtools.NamedChromaticPitchSet( [ "eqf'", "gqf''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'three', 'four', 'five'), left_hand = ('R', 'thumb
            ', 'cis'), right_hand = ('gis',)), ''),
    7: (pitchtools.NamedChromaticPitchSet( [ "g'", "b''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'five', 'six'), left_hand = ('R', 'thumb'),
            right_hand=('three', 'four')), 'prominent 4th'),
    9: (pitchtools.NamedChromaticPitchSet( [ "fqs'", "cqs'''" ] ),
        WoodwindFingering('clarinet', center_column = ('two', '
            three', 'four', 'five', 'six'), left_hand = ('R', '
            thumb'), right_hand = ('gis',) ), 'beats'),
    12: (pitchtools.NamedChromaticPitchSet( [ "g'", "d'''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'three', 'four'), left_hand = ('R',) ), 'dissonant/
            beats'),
    },
'shrill':
    {1: (pitchtools.NamedChromaticPitchSet( [ "c'", "fs''", "c
        '''", "fs'''", "as'''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'three', 'four', 'five', 'six'), left_hand = ('
            thumb', 'cis',) ), 'three pitches'),
    13: (pitchtools.NamedChromaticPitchSet( [ "f", "d'''", "f'''",
         "as'''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'three', 'four', 'five', 'six'), left_hand = ('
            thumb',), right_hand = ('f',)), 'resonant'),
    14: (pitchtools.NamedChromaticPitchSet( [ "g", "b''", "f'''",
        "a'''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
            ', 'three', 'four', 'five', 'six'), left_hand = ('
            thumb',)), 'resonant'),
    18: (pitchtools.NamedChromaticPitchSet( [ "e", "g''", "c'''",
        "e'''", "g'''", "a'''" ] ),
        WoodwindFingering('clarinet', center_column = ('one', 'two
```

```
                        ', 'three', 'four', 'five', 'six'), left_hand = ('
                        thumb',), right_hand = ('e',)), 'buzzy'),
                }
            }

def make_multiphonic_markup(fingering, size=0.65, thickness=0.3, padding = 3,
    graphical=False):
    diagram = fingering()
    graphical = markuptools.MarkupCommand('override', schemetools.SchemePair('
        graphical', False))
    size = markuptools.MarkupCommand('override', schemetools.SchemePair('size',
        size))
    thickness = markuptools.MarkupCommand('override', schemetools.SchemePair('
        thickness', thickness))
    padded_markup = markuptools.MarkupCommand('pad-markup', schemetools.Scheme(
        padding ), [graphical, size, thickness, diagram])
    markup = markuptools.Markup(padded_markup, direction=Down)
    return markup

def make_possible_entrances_and_exits(pitch_list, fingering):
    possibilities = []
    for pitch in pitch_list:
        in_measure = in_measure_from_pitch_list_and_fingering(pitch, pitch_list,
            fingering)
        out_measure = out_measure_from_pitch_list_and_fingering(pitch, pitch_list,
             fingering)
        possibilities.extend([in_measure, out_measure])
    return possibilities

def in_measure_from_pitch_list_and_fingering(pitch, pitch_list, fingering):
    in_measure = Measure((4,4), [])
    in_measure.append( Chord([pitch], (1,2)) )
    chord = Chord(pitch_list, (1,2))
    markup = make_multiphonic_markup(fingering)
    markup.attach(chord)
    in_measure.append(chord)
    tietools.TieSpanner(in_measure[:])
    return in_measure

def out_measure_from_pitch_list_and_fingering(pitch, pitch_list, fingering):
    out_measure = Measure((4,4), [])
    chord = Chord(pitch_list, (1,2))
    markup = make_multiphonic_markup(fingering)
    markup.attach(chord)
    out_measure.append(chord)
    out_measure.append( Chord([pitch], (1,2)) )
    tietools.TieSpanner(out_measure[:])
    return out_measure
```

```
def multiphonic_dictionary_to_possibilities(dictionary):
    measures = []
    for value in dictionary.values():
        possibilities = make_possible_entrances_and_exits(value[0], value[1])
        measures.extend(possibilities)
    return measures

def multiphonics_to_possibilities(multiphonics):
    measures = []
    for value in multiphonics.values():
        print 'here.'
        measures = multiphonic_dictionary_to_possibilities(value)
        measures.extend(measures)
    return measures

def make_and_edit_measures(multiphonics):
    cresc_measures = multiphonic_dictionary_to_possibilities(multiphonics['cresc
        '])
    indexes = [ 5,6,7, 8, 27, 28 ]
    indexes = [x-1 for x in indexes]
    print_measures = []
    for index in indexes:
        measure = cresc_measures[index]
        print_measures.append( measure )
    soft_measures = multiphonic_dictionary_to_possibilities(multiphonics['soft'])
    indexes =  [2, 4, 10, 11, 12, 13, 14, 16, 17, 19, 20]
    indexes = [x-1 for x in indexes]
    print_softs = []
    for index in indexes:
        measure = soft_measures[index]
        print_softs.append( measure )
    print_measures.extend(print_softs)
    diad_measures = multiphonic_dictionary_to_possibilities(multiphonics['diad'])
    indexes =  [12, 15]
    indexes = [x-1 for x in indexes]
    print_diads = []
    for index in indexes:
        measure = diad_measures[index]
        print_diads.append( measure )
    print_measures.extend(print_diads)
    shrill_measures = multiphonic_dictionary_to_possibilities(multiphonics['shrill
        '])
    indexes =  [1, 2, 7, 8, 37, 38]
    indexes = [x-1 for x in indexes]
    print_shrills = []
    for index in indexes:
        measure = shrill_measures[index]
```

```
        print_shrills.append( measure )
    print_measures.extend(print_shrills)
    return print_measures

print_measures = make_and_edit_measures(multiphonics)
```

A.6 Clarinet Solo Material Based on Multiphonic Catalogue

```
from abjad import *
from ratioChains import *
import os
from itertools import permutations
from multiphonics import *
from random import *

def tupletize_duration_recursively_by_ratio_at_depth(duration, ratio, depth):
    tuplet = tuplettools.make_tuplet_from_duration_and_ratio(duration, ratio)
    for x in range(depth):
        last_tuplet_created = tuplettools.
            get_first_tuplet_in_proper_parentage_of_component(tuplet.leaves[-1])
        last_tuplet_created[1:] = [ tuplettools.
            make_tuplet_from_duration_and_ratio( duration, ratio ) ]
    return tuplet

def replace_leaves_with_pitches(voice, pitches):
    if markuptools.get_markup_attached_to_component(voice[0]):
        diagram = markuptools.remove_markup_attached_to_component(voice[0])
    for leaf in voice.leaves:
        chord = Chord(leaf)
        chord[:] = []
        chord.extend(pitches)
        componenttools.move_parentage_and_spanners_from_components_to_components([
            leaf], [chord])
    if isinstance(voice[0], Chord):
        markup = diagram
        markup.attach(voice[0])

def change_note_heads_if_air_note(choice, voice):
    if hasattr(choice.override.note_head, 'style'):
        for note in voice:
            note.override.accidental.stencil = False
            note.override.note_head.style = 'harmonic'

def make_diaphragm_bounce_voice(duration, depth, choice):
```

```
    tuplet = tupletize_duration_recursively_by_ratio_at_depth(duration, [1,1,1],
        depth)
    voice = Voice([tuplet])
    if isinstance(choice, Chord) and len(choice) > 1:
        diagram = markuptools.remove_markup_attached_to_component(choice)[0]
        pitches = choice.written_pitches
    else:
        change_note_heads_if_air_note(choice, voice)
        pitches = choice.written_pitches
    replace_leaves_with_pitches(voice, pitches)
    if isinstance(choice, Chord) and len(choice) > 1:
        diagram.attach(voice[0][0])
    voice.override.tuplet_bracket.stencil = False
    voice.override.tuplet_number.stencil = False
    voice.override.stem.stencil = False
    voice.override.stem.stencil = False
    return voice

def choose_multiphonic(multiphonics):
    group = sample(multiphonics.values(), 1)[0]
    multiphonic = sample(group.values(), 1)[0]
    return multiphonic



def apply_multiphonic(voice):
    multiphonic = choose_multiphonic(multiphonics)
    pitches = multiphonic[0]
    fingering = multiphonic[1]
    diagram = fingering()
    replace_leaves_with_pitches(voice,pitches)
    markup = markuptools.Markup(diagram,direction=Down)(voice.leaves[0])

def format_bounce(voice):
    for x in range( len(voice.leaves) ):
        if x % 2 is 0:
            marktools.Articulation('>')(voice.leaves[x])

def make_air_chord(bar):
    note = [x for x in bar if len(x) == 1][0]
    #make the air note
    air_note = Chord(note.written_pitches,(1,2))
    air_note.override.note_head.style = 'harmonic'
    air_note.override.accidental.stencil = False
    note.override.accidental.stencil = 'false'
    return air_note

def choose_fermata():
    prefixes = ['short', '', 'long', 'verylong']
```

```
    prefix = sample(prefixes,1)[0]
    mark_string = prefix + ’fermata’
    return mark_string

def make_rest():
    rest = Rest(”r4”)
    fermata_string = choose_fermata()
    marktools.LilyPondCommandMark(fermata_string, ’after’)(rest)
    return rest

def replace_leaf_with_bounce(copies):
    candidates = [x for x in copies if not isinstance(x, Rest)]
    choice = sample(candidates,1)[0]
    bounce_depth = randint(6,11)
    if isinstance(choice, Note):
        voice = make_diaphragm_bounce_voice( choice.duration, bounce_depth, choice
            )
        change_note_heads_if_air_note(choice, voice)
    else:
        voice = make_diaphragm_bounce_voice(choice.duration, bounce_depth, choice)
    format_bounce(voice)
    componenttools.move_parentage_and_spanners_from_components_to_components([
        choice], [voice])

def bar_to_bars_with_rests_and_air_notes(bar):
    out_bars = []
    leaves = list(componenttools.copy_components_and_covered_spanners(bar.leaves))
    rest = make_rest()
    leaves.append( rest )
    air_note = make_air_chord(bar)
    leaves.append(air_note)
    for permutation in permutations(leaves):
        copies = componenttools.copy_components_and_covered_spanners(permutation)
        new_bar = Measure((7,4), copies)
        replace_leaf_with_bounce(new_bar.leaves)
        out_bars.append(new_bar)
    return out_bars

def add_air_and_rest_to_bars(bars):
    out_bars = []
    for bar in bars:
        permuted = bar_to_bars_with_rests_and_air_notes(bar)
        out_bars.extend(permuted)
    return out_bars

def tie_groups_in_bar(bar):
    for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(bar, (
        Note, Chord)):
```

```
        tietools.TieSpanner(group)

def liberate_notes_from_voice(bar):
    for component in bar:
        if isinstance(component, Voice):
            componenttools.replace_components_with_children_of_components([
                component])

def delete_abutting_rests(voice):
    for x in reversed(range(len(voice) - 1)):
        if isinstance(voice[x], Rest) and isinstance(voice[x+1], Rest):
            del(voice[x])

def choose_and_skip(choices, to_skip):
    take_out = set([to_skip])
    choices = set(choices)
    choices = list(choices.difference(take_out) )
    return sample(choices, 1)[0]

def add_dynamic_to_first_in_bar_based_on_last(last, bar):
    dynamic = choose_and_skip( ['ppp', 'pp', 'p', 'mp'], last)
    if isinstance(bar[0], Rest) and isinstance(bar[1], Tuplet):
        contexttools.DynamicMark(dynamic)(bar[1][0])
    elif isinstance(bar[0], Rest) and isinstance(bar[1], Chord):
        contexttools.DynamicMark(dynamic)(bar[1])
    elif isinstance(bar[0], Tuplet):
        contexttools.DynamicMark(dynamic)(bar[0][0])
    else:
        contexttools.DynamicMark(dynamic)(bar[0])
    return dynamic

def add_arrow_spanner_to_leaves(leaves):
    print leaves
    arrow = spannertools.TextSpanner(leaves)
    arrow.override.text_spanner.bound_details__right__arrow = True
    arrow.override.text_spanner.style = schemetools.Scheme('solid-line', quoting
        ="'")

def add_arrow_spanner_to_leaves_in_bar(bar):
    for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(bar.
        leaves, (Chord, Tuplet)):
        if len(group) > 1:
            add_arrow_spanner_to_leaves(group[:])

def make_voice_from_chart(print_measures):
    for bar in print_measures:
        bar.automatically_adjust_time_signature = True
        out_bars = add_air_and_rest_to_bars(print_measures)
```

```
            shuffle(out_bars)
            voice = Voice(out_bars)
        tietools.remove_tie_spanners_from_components_in_expr(voice)
        voice.override.script.padding = 2
        voice.override.time_signature.stencil = False
        voice.override.stem.stencil = False
        voice.override.tuplet_bracket.stencil = False
        voice.override.tuplet_number.stencil = False
        return voice

    def decorate_voice(voice):
        last = 'mp'
        for bar in voice:
            liberate_notes_from_voice(bar)
            tie_groups_in_bar(bar.leaves)
            add_arrow_spanner_to_leaves_in_bar(bar)
            last = add_dynamic_to_first_in_bar_based_on_last(last, bar)
        for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(voice.
            leaves, (Chord, Chord)):
            if len(group) > 1:
                tietools.TieSpanner(group[:])
                delete_abutting_rests(voice)

    def make_score_from_voice(voice):
        staff = Staff([voice])
        staff.override.time_signature.stencil = False
        staff.override.bar_line.stencil = False
        score = Score([staff])
        score.set.proportional_notation_duration = schemetools.SchemeMoment(1, 56)
        contexttools.set_accidental_style_on_sequential_contexts_in_expr(score,'forget
            ')
        return score

    def make_lilypond_file(score):
        lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
        lilypond_file.default_paper_size = 'legal', 'landscape'
        lilypond_file.global_staff_size = 18
        lilypond_file.layout_block.indent = 0
        lilypond_file.layout_block.ragged_right = True
        lilypond_file.paper_block.top_margin = 15
        lilypond_file.paper_block.bottom_margin = 3
        lilypond_file.paper_block.left_margin = 15
        lilypond_file.paper_block.right_margin = 15
        lilypond_file.paper_block.markup_system_spacing__basic_distance = 15
        lilypond_file.paper_block.ragged_bottom = True
        lilypond_file.paper_block.system_system_spacing = layouttools.
            make_spacing_vector(0, 0, 8, 0)
        return lilypond_file
```

```
def make_clarinet_solo_document(print_measures):
    voice = make_voice_from_chart(print_measures)
    decorate_voice(voice)
    score = make_score_from_voice(voice)
    lilypond_file = make_lilypond_file(score)
    show(lilypond_file)

make_clarinet_solo_document(print_measures[:10])
```

A.7 Prepared Piano Part for *The World All Around*

```
from abjad import *
from random import *
from clarinetSolo import *

def choose_and_skip(choices, to_skip):
    take_out = set([to_skip])
    choices = set(choices)
    choices = list(choices.difference(take_out) )
    choice =  sample(choices, 1)[0]
    return choice

def replace_note_below_split_with_rest(note, split_pitch):
    if split_pitch.chromatic_pitch_number > note.written_pitch.
        chromatic_pitch_number:
            duration = note.written_duration
            rest = leaftools.make_tied_leaf( Rest, duration )
            index = note.parent.index( note )
            note.parent[ index: index+1 ] = rest

def remove_chord_pitches_below_split(chord, split_pitch):
    index = chord.parent.index( chord )
    for note in reversed(chord):
        if split_pitch.chromatic_pitch_number > note.written_pitch.
            chromatic_pitch_number:
            note_index = chord.written_pitches.index( note )
            chord.pop(note_index)
    if 0 == len(chord.written_pitches):
        rest = leaftools.make_tied_leaf( Rest, chord.written_duration )
        chord.parent[ index:index+1 ] = rest

def replace_note_above_split_with_rest(note, split_pitch):
    if split_pitch.chromatic_pitch_number <= note.written_pitch.
        chromatic_pitch_number:
            duration = note.written_duration
            rest = leaftools.make_tied_leaf( Rest, duration )
```

```
            index = note.parent.index( note )
            note.parent[ index: index+1 ] = rest

def remove_chord_pitches_above_split(chord, split_pitch):
    index = chord.parent.index( chord )
    popped = 0
    for note in reversed(chord):
        if split_pitch.chromatic_pitch_number <= note.written_pitch.
            chromatic_pitch_number:
            note_index = chord.written_pitches.index( note )
            chord.pop(note_index)
            popped = 1
    if 0 == len(chord.written_pitches):
        rest = leaftools.make_tied_leaf( Rest, chord.written_duration )
        chord.parent[ index:index+1 ] = rest
    elif popped:
        remove_number_label_from_chord(chord)

def remove_pitches_below_split_in_components(voice, split_pitch):
    for note in iterationtools.iterate_notes_in_expr(voice.leaves):
        replace_note_below_split_with_rest(note, split_pitch)
    for chord in iterationtools.iterate_chords_in_expr(voice.leaves):
        remove_chord_pitches_below_split(chord, split_pitch)

def remove_pitches_above_split_in_components(voice, split_pitch):
    for note in iterationtools.iterate_notes_in_expr(voice.leaves):
        replace_note_above_split_with_rest(note, split_pitch)
    for chord in iterationtools.iterate_chords_in_expr(voice.leaves):
        remove_chord_pitches_above_split(chord, split_pitch)

def split_components_to_piano_staff_at_pitch(components, split_pitch = pitchtools.
    NamedChromaticPitch("c'")):
    piano_staff = scoretools.PianoStaff()
    #piano_staff.engraver_consists.append("#Span_stem_engraver")
    treble_staff = Staff()
    treble_staff.name = "treble"
    bass_staff = Staff()
    bass_staff.name = "bass"
    copies = componenttools.copy_components_and_covered_spanners( components )
    treble_voice = Voice(copies)
    #treble_voice.override.script.padding =
    copies = componenttools.copy_components_and_covered_spanners( components )
    bass_voice = Voice(copies)
    #bass_voice.override.script.padding =
    remove_pitches_below_split_in_components(treble_voice, split_pitch)
    remove_pitches_above_split_in_components(bass_voice, split_pitch)
    bass_staff.extend(bass_voice)
    treble_staff.extend(treble_voice)
```

```
    contexttools.ClefMark('bass')(bass_staff)
    #marktools.LilyPondCommandMark("autoBeamOff")(bass_staff[0])
    #marktools.LilyPondCommandMark("voiceOne",)(bass_staff[0])
    piano_staff.extend([treble_staff,bass_staff])
    return piano_staff

def format_piano_staff(piano_staff):
    #piano_staff.override.bar_line.stencil = False
    p#iano_staff.override.span_bar.stencil = False
    piano_staff.override.beam.transparent = True
    piano_staff.override.tuplet_bracket.stencil = False
    piano_staff.override.tuplet_number.stencil = False
    piano_staff.override.dots.transparent = True
    piano_staff.override.rest.transparent = True
    piano_staff.override.tie.transparent = True
    piano_staff.override.stem.transparent = True
    piano_staff.override.flag.stencil = False
    #piano_staff[0].override.bar_line.stencil = False
    #piano_staff[1].override.bar_line.stencil = False

def get_pitch_class_string(abbreviation):
    base = abbreviation[0]
    base = base.upper()
    return markuptools.MarkupCommand("left-align", "\\teeny", base)

def add_markup_to_illegible_note(note):
    padding = 3
    if note.written_pitch.chromatic_pitch_number >= 31:
        class_abbreviation = str(note.written_pitch.named_chromatic_pitch_class)
        letter = get_pitch_class_string(class_abbreviation)
        padded_markup = markuptools.MarkupCommand('pad-markup', schemetools.Scheme
            ( padding ), letter)
        markup = markuptools.Markup(padded_markup, direction=Up)(note)
    elif note.written_pitch.chromatic_pitch_number <= -27:
        class_abbreviation = str(note.written_pitch.named_chromatic_pitch_class)
        letter = get_pitch_class_string(class_abbreviation)
        padded_markup = markuptools.MarkupCommand('pad-markup', schemetools.Scheme
            ( padding ), letter)
        markup = markuptools.Markup(padded_markup, direction=Down)(note)

def add_markup_to_illegible_notes_in_leaves(leaves):
    notes = [x for x in leaves if isinstance(x, Note)]
    for note in notes:
        add_markup_to_illegible_note(note)

def derive_unprepared_pitches(preparation_groups):
    all_prepared_pitches = [pitchtools.NamedChromaticPitch(x).
        chromatic_pitch_number for x in sequencetools.flatten_sequence(
```

```
        preparation_groups)]
    all_piano_pitches = set( [x - 39 for x in range(88)] )
    unprepared_pitches = list( all_piano_pitches.difference(all_prepared_pitches)
        )
    unprepared_pitches.sort()
    unprepared_pitches = [pitchtools.NamedChromaticPitch(x).chromatic_pitch_name
        for x in unprepared_pitches]
    return unprepared_pitches

def list_prepared_notes(preparation_groups):
    staff = Staff()
    for group in preparation_groups:
        pitches = [pitchtools.NamedChromaticPitch(x) for x in group]
        pitches.sort()
        pitches.reverse()
        notes = [Note(x,(1,4)) for x in pitches]
        voice = Voice(notes)
        staff.append(voice)
    return staff

def attach_dynamic(note, previous_dynamic):
    choices = ['ppp', 'pp', 'p', 'mp', 'mf', 'f']
    choice = choose_and_skip(choices, previous_dynamic)
    previous_dynamic = choice
    contexttools.DynamicMark(choice)(note)
    return previous_dynamic

def format_staff(staff):
    for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(staff, (
        Note, Note)):
        for chain in tietools.iterate_tie_chains_in_expr(group):
            marktools.LilyPondCommandMark('laissezVibrer', 'after')(chain[0])
            add_markup_to_illegible_note(chain[0])
            chain[0].override.note_head.duration_log = 2
            for note in chain[1:]:
                note.override.note_head.transparent = True
                note.override.note_head.no_ledgers = True
                note.override.accidental.stencil = False

def make_chain(staff, pitch_strings, total, previous_dynamic, to_skip):
    pitch_string = choose_and_skip(pitch_strings, to_skip)
    pitch = pitchtools.NamedChromaticPitch(pitch_string)
    duration = rest_intervals_as_durations.pop(0)
    total += duration
    chain = leaftools.make_tied_leaf(notetools.Note, duration, pitches = pitch )
    previous_dynamic = attach_dynamic(chain[0], previous_dynamic)
    to_skip = chain[0].written_pitch.chromatic_pitch_name
    staff.extend( chain )
```

```
    return total, to_skip, previous_dynamic, chain

def add_quarter_of_form_to_staff(staff, total, check_duration, previous_dynamic,
    pitch_strings, unprepared_treble, unprepared_wait_times):
    colors = ['red', 'green', 'blue', 'yellow']
    to_skip = pitch_strings[0]
    choice = sample(unprepared_wait_times, 1)[0]
    unpreparation_count = 0
    wait_time = sample(unprepared_wait_times, 1)[0]
    while total < check_duration:
        unpreparation_count += 1
        if rest_intervals_as_durations == []:
            break
        if unpreparation_count == wait_time:
            total, to_skip, previous_dynamic, chain = make_chain( staff,
                unprepared_treble, total, previous_dynamic, to_skip)
            #markuptools.Markup('unprepared!',direction=Up)(chain[0])
            unpreparation_count = 0
            wait_time = sample(unprepared_wait_times, 1)[0]
        else:
            total, to_skip, previous_dynamic, chain = make_chain( staff,
                pitch_strings, total, previous_dynamic, to_skip)
        staff.extend( chain )
    return total, previous_dynamic

def make_staff(rest_intervals_as_durations, preparation_groups, unprepared_treble)
    :
    previous_dynamic = 'f'
    shuffle(rest_intervals_as_durations)
    quarter = ( sum(rest_intervals_as_durations) / 4)
    staff = Staff()
    total = Duration(0,1)
    unprepared_wait_times = [0]
    for x in range(4):
        if x == 1:
            unprepared_wait_times = [5,6,7]
        elif x== 2:
            unprepared_wait_times = [2,3,4]
        elif x == 3:
            unprepared_wait_times = [1,2]
        pitch_strings = preparation_groups[3-x]
        total, previous_dynamic = add_quarter_of_form_to_staff(staff, total,
            quarter * (x + 1), previous_dynamic, pitch_strings, unprepared_treble,
             unprepared_wait_times)
    componenttools.split_components_at_offsets(staff.leaves, [Duration(7,4)],
        cyclic = True)
    contexttools.TimeSignatureMark( (7,4) )(staff)
    return staff
```

```
def choose_sustain_string_based_on_last(last):
    if last == 'sustainOn' or last == 'sustainOff\\sustainOn':
        choices = ['sustainOff', 'sustainOff\\sustainOn' ]
    elif last == 'sustainOff':
        choices =  ['sustainOn']
    choice = sample(choices, 1)[0]
    return choice

def choose_corda_string_based_on_last(last):
    choices = [ 'unaCorda', 'treCorde' ]
    choice = choose_and_skip(choices, last)
    return choice

def add_corda_mark_to_note_based_on_last(note, last, bass):
    choice = choose_corda_string_based_on_last(last)
    index = note.parent.index(note)
    mark = marktools.LilyPondCommandMark(choice, 'after')(bass[ index ])
    last = choice
    return last

def add_sustain_mark_to_note_based_on_last(note,last, bass):
    choice = choose_sustain_string_based_on_last(last)
    index = note.parent.index(note)
    mark = marktools.LilyPondCommandMark(choice, 'after')(bass[ index ])
    last = choice
    return last

def add_pedal_marks_to_piano_staff(staff, piano_staff):
    treble = piano_staff[0]
    bass = piano_staff[1]
    bass.set.pedal_sustain_style = 'mixed'
    marktools.LilyPondCommandMark('sustainOn', 'after')(bass[0])
    marktools.LilyPondCommandMark('unaCorda', 'after')(bass[0])
    last_sustain = 'sustainOn'
    last_corda = 'unaCorda'
    chains = list(tietools.iterate_tie_chains_in_expr(staff))
    chains = chains[1:]
    for chain in chains:
        add_sustain = randint(0,1)
        add_corda = randint(0,1)
        if add_sustain:
            last_sustain = add_sustain_mark_to_note_based_on_last(chain[0],
                last_sustain, bass)
        if add_corda:
            last_corda = add_corda_mark_to_note_based_on_last(chain[0], last_corda
                , bass)
```

```
def make_piano_staff(rest_intervals_as_durations, preparation_groups,
    unprepared_treble):
    staff = make_staff(rest_intervals_as_durations, preparation_groups,
        unprepared_treble)
    piano_staff = split_components_to_piano_staff_at_pitch(staff[:], split_pitch =
         pitchtools.NamedChromaticPitch(”c’”))
    format_piano_staff(piano_staff)
    format_staff(piano_staff[0])
    format_staff(piano_staff[1])
    add_pedal_marks_to_piano_staff(staff, piano_staff)
    return piano_staff

#! d’’’, e’’, c’’’’ are a unison.

preparation_groups = [ \
#celesta
[ ”a’’’’”, ”g’’’’”, ”f’’’’”, ”e’’’’”, ”ef’’’’”, ”d’’’’”, ”b’’’”, ”bf’’’”, ”a’’’”,”
    af’’’”, ”g’’’”, ”ef’’’”, ”e’’’”, ”cs’’’”, ”c’’’”, ”b’’”], \
#between
[”d’’”, ”fs’’”, ”f’’”, ”a’’” ], \
#pitched but not piano
[ ”d,”, ”d”, ”g”, ”g’”, ”af’”, ”bf’”, ”c’’”, ”b’”, ”e’’”, ”fs’’’”], \
#percussion
[ ”d,,”, ”af”, ”a”, ”bf”, ”b”, ”c’”, ”cs’”, ”d’”, ”ef’”, ”cs’’”, ”ef’’”, ”af’’”, ”
    f’’’”] \
]

unprepared_pitches = derive_unprepared_pitches(preparation_groups)
unprepared_treble = [x for x in unprepared_pitches if Note(x).written_pitch.
    chromatic_pitch_number > -8]
piano_staff = make_piano_staff(rest_intervals_as_durations, preparation_groups,
    unprepared_treble)
#score = Score([piano_staff])
#show(score)

\normalsize
```

A.8 Harp Part for *The World All Around*

```
    from categorizedPreparations import *
#for each bar,
#get the set of pitches in the clarinet part and transpose them down by a whole
    step to get concert pitches.
#compose four interval taleas, one per quarter of the form. Some intervals should
    stay the same, others should change.
interval_sets = [ \
[6,1,0],\
```

```
[6,1, 2, 0, 8],\
[6,1, 2, 0, 8, 3],\
[1, 2, 0, 11]\
]

#choose a new set of pitches based on the chosen intervals.
#choose an octave for those pitches.
#choose one pitch or an octave.
#if it’s one pitch, make it a harmonic or not.
#use location of rests in clarinet parts to place harp events (shuffled)
#per bar, if an event exists,
#get the pitches for that bar,
#choose two pitches and make them within a tenth. (wrapping)
def choose_pitch_based_on_bar_set(bar_set, interval_set):
    base_pitch = sample(bar_set, 1)[0]
    interval = sample(interval_set, 1)[0]
    up = randint(0,1)
    if up:
        out_pitch = pitchtools.NamedChromaticPitch( base_pitch + interval)
    else:
        out_pitch = pitchtools.NamedChromaticPitch( base_pitch - interval)
    return out_pitch

def reoctavize_pitch(pitch, last):
    class_name = pitch.chromatic_pitch_class_name
    octave_choices = [ ’,,’, ’,’, ’’, ”’”, ”’’”, ”’’’”, ”’’’’”]
    choice = choose_and_skip(octave_choices, last)
    last = choice
    pitch = class_name + choice
    return last, pitchtools.NamedChromaticPitch(pitch)

def choose_and_reoctavize_pitch_based_on_bar_set(pitch_set, interval_set,
    last_octave):
    pitch = choose_pitch_based_on_bar_set(pitch_set, interval_set)
    last, shifted_pitch = reoctavize_pitch(pitch, last_octave)
    return last, shifted_pitch

def add_flageolet_to_note_based_on_pitch(note):
    if note.written_pitch.chromatic_pitch_number < 0:
        marktools.Articulation(’flageolet’, Down)(note)
    else:
        marktools.Articulation(’flageolet’, Up)(note)

def make_chain(staff, total, previous_dynamic, interval_set, pitch_set,
    harp_rest_intervals, previous_octave):
    previous_octave, pitch = choose_and_reoctavize_pitch_based_on_bar_set(
        pitch_set, interval_set, previous_octave)
    duration = harp_rest_intervals.pop(0)
```

```
    total += duration
    print total
    chain = leaftools.make_tied_leaf(notetools.Note, duration, pitches = pitch )
    previous_dynamic = attach_dynamic(chain[0], previous_dynamic)
    harmonic_chances = [0,0,0,1]
    harmonic = sample(harmonic_chances, 1)[0]
    if harmonic and chain[0].written_pitch.chromatic_pitch_number > 0:
        add_flageolet_to_note_based_on_pitch(chain[0])
    staff.extend( chain )
    return total, previous_dynamic, previous_octave, chain

def add_harp_quarter_of_form_to_staff(staff, total, check_duration,
    previous_dynamic, interval_set, pitch_set, harp_rest_intervals,
    previous_octave):
    while total < check_duration:
        if harp_rest_intervals == []:
            break
        total, previous_dynamic, previous_octave, chain = make_chain( staff, total
            , previous_dynamic, interval_set, pitch_set, harp_rest_intervals,
            previous_octave)
        staff.extend( chain )
    return total, previous_dynamic, previous_octave

def make_harp_staff(harp_rest_intervals, interval_sets, pitch_sets_by_bar):
    previous_dynamic = 'f'
    previous_octave = ',,'
    shuffle(harp_rest_intervals)
    quarter = ( sum(harp_rest_intervals) / 4)
    staff = Staff()
    total = Duration(0,1)
    for x in range(4):
        measure_index = int( total / Duration(7,4) )
        if total % Duration(7,4) == 0:
            measure_index -= 1
        pitch_set = pitch_sets_by_bar[ measure_index ]
        interval_set = interval_sets[ x ]
        total, previous_dynamic, previous_octave =
            add_harp_quarter_of_form_to_staff(staff, total, quarter * (x + 1),
            previous_dynamic, interval_set, pitch_set, harp_rest_intervals,
            previous_octave)
    componenttools.split_components_at_offsets(staff.leaves, [Duration(7,4)],
        cyclic = True)
    contexttools.TimeSignatureMark( (7,4) )(staff)
    return staff

def make_harp_double_staff(harp_rest_intervals, interval_sets, pitch_sets_by_bar):
    staff = make_harp_staff(harp_rest_intervals, interval_sets, pitch_sets_by_bar)
    double_staff = split_components_to_piano_staff_at_pitch(staff[:], split_pitch
```

```
        = pitchtools.NamedChromaticPitch(”c’”))
    format_piano_staff(double_staff)
    format_staff(double_staff[0])
    format_staff(double_staff[1])
    double_staff.override.script.padding = 1
    return double_staff

harp_staff = make_harp_double_staff(harp_rest_intervals, interval_sets,
    pitch_sets_by_bar)
```

A.9 Formatted Score for *The World All Around*

```
from abjad import *
from ratioChains import *
import os
from harp import *

def format_score(clarinet_staff, piano_staff, harp_staff):
    #clarinet_staff = Staff(”c’ d’ e’ f’”)
    contexttools.InstrumentMark(’Clarinet’, ’clar.’,)(clarinet_staff)
    #piano_treble = Staff(”c’ d’ e’ f’”)
    #piano_bass = Staff(”c’ d’ e’ f’”)
    #piano_treble.name = ’piano treble’
    #piano_bass.name = ’piano bass’
    #piano_staff = scoretools.PianoStaff()
    #piano_staff.extend([piano_treble,piano_bass])
    contexttools.InstrumentMark(’Piano’, ’Pno.’, target_context = scoretools.
        PianoStaff)(piano_staff)
    contexttools.InstrumentMark(
        ’Harp’, ’Hp.’,
        target_context = scoretools.PianoStaff
        )(harp_staff)
    score = Score([])
    score.append(clarinet_staff)
    score.append(piano_staff)
    score.append(harp_staff)
    marktools.BarLine(’|.’)(clarinet_staff[0][-1])
    marktools.BarLine(’|.’)(piano_staff[0][-1])
    score.override.rehearsal_mark.padding = 3
    score.override.bar_number.stencil = False
    score.set.proportional_notation_duration = schemetools.SchemeMoment(1, 64)
    score.override.spacing_spanner.uniform_stretching = True
    score.override.spacing_spanner.strict_note_spacing = True
    score.override.time_signature.stencil = False
    contexttools.set_accidental_style_on_sequential_contexts_in_expr(score,’forget
        ’)
    return score
```

```
def make_lilypond_file(score):
    lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
    lilypond_file.default_paper_size = ’tabloid’, ’portrait’
    lilypond_file.global_staff_size = 14
    lilypond_file.layout_block.indent = 0
    lilypond_file.layout_block.ragged_right = True
    lilypond_file.paper_block.top_margin = 15
    lilypond_file.paper_block.left_margin = 20
    lilypond_file.paper_block.right_margin = 15
    lilypond_file.paper_block.markup_system_spacing__basic_distance = 5
    lilypond_file.paper_block.ragged_bottom = False
    lilypond_file.paper_block.system_system_spacing = layouttools.
        make_spacing_vector(0, 0, 8, 0)
    directory = os.path.abspath(os.getcwd())
    fontTree = directory+’/fontTree.ly’
    lilypond_file.file_initial_user_includes.append(fontTree)
    return lilypond_file

def make_piece(clarinet_staff, piano_staff, harp_staff):
    score = format_score(clarinet_staff, piano_staff, harp_staff)
    lilypond_file = make_lilypond_file(score)
    show(lilypond_file)

make_piece(clarinet_staff, piano_staff, harp_staff)
```

Appendix B

Score Examples

B.1 Arvo Pärt's *Cantus in Memory of Benjamin Britten* (1977—80) for Bell and String Orchestra, as Rendered with the Abjad API for Formalized Score Control

Cantus in Memory of Benjamin Britten (1980)

Arvo Pärt

2

17 C

Camp. *p*

Vl. I

Vl. II

Va. *mp*

Vc.

Cb.

21

Camp.

Vl. I *p*

Vl. II

Va.

Vc. *mp*

Cb.

25 D

Camp. *mp*

Vl. I

Vl. II *mp*

Va.

Vc.

Cb. *mf*

29 E

Camp.

Vl. I *mp*

Vl. II

Va. *mf*

Vc. *mf*

Cb.

34 F

Camp. *mf*

Vl. I

Vl. II *mf*

Va. *f*

Vc.

Cb.

3

39 G

Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

mf *f* *f*

44

Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

f *f* *f*

49 H

Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

ff *ff* *ff* *ff*

54 J

Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

ff *ff*

4

59

K

Camp.

fff

div.

uniti

Vl. I

Vl. II

Va.

Vc.

Cb.

fff

fff

fff

fff

64

L

Camp.

uniti

Vl. I

Vl. II

uniti

div.

Va.

Vc.

Cb.

fff

69

Camp.

ff

Vl. I

Vl. II

Va.

Vc.

Cb.

M

73

Camp.

Vl. I

Vl. II

uniti

Va.

uniti

Vc.

Cb.

77 N
Camp.
f
Vl. I
Vl. II
Va.
Vc.
Cb.

81
Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

O
85
Camp.
mf
Vl. I
Vl. II
Va.
uniti
Vc.
espr.
Cb.
espr.

89 P
Camp.
Vl. I
Vl. II
Va.
Vc.
molto espr.
Cb.
molto espr.

93
Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

6

Q

97

Camp.

Vl. I

Vl. II

Va.

Vc.

Cb.

uniti

101

R

(non dim.)

(non dim.)

(non dim.)

(non dim.)

(non dim.)

105

pp

B.2 Iannis Xenakis's *Windugen* (1976) for Twelve Cellos, as Rendered with the Abjad API for Formalized Score Control

Windungen (1976)
18-04-2013

Iannis Xenakis

Cello 1
Cello 2
Cello 3
Cello 4
Cello 5
Cello 6
Cello 7
Cello 8
Cello 9
Cello 10
Cello 11
Cello 12

8

1
2
3
4
5
6
7
8
9
10
11
12

12

1
2
3
4
5
6
7
8
9
10
11
12

15

1 2 3 4 5 6 7 8 9 10 11 12

3

18

1 2 3 4 5 6 7 8 9 10 11 12

4

21

1
2
3
4
5
6
7
8
9
10
11
12

24

1
2
3
4
5
6
7
8
9
10
11
12

27 5

1 2 3 4 5 6 7 8 9 10 11 12

30

1 2 3 4 5 6 7 8 9 10 11 12

6
33
1
2
3
4
5
6
7
8
9
10
11
12
35
1
2
3
4
5
6
7
8
9
10
11
12

7

37

1
2
3
4
5
6
7
8
9
10
11
12

40

1
2
3
4
5
6
7
8
9
10
11
12

8

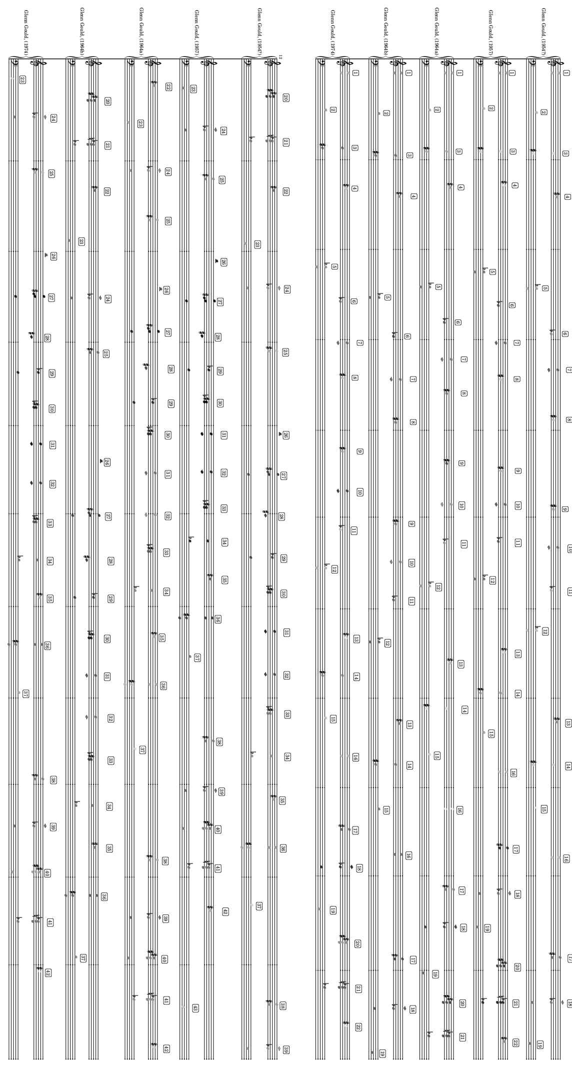
42

1 2 3 4 5 6 7 8 9 10 11 12

44

1 2 3 4 5 6 7 8 9 10 11 12

B.3 Glenn Gould's Performances of the First Movement of Webern's op. 27 Piano Variations



Glenn Gould, (1954?)

Glenn Gould, (1957)

Glenn Gould, (1964a)

Glenn Gould, (1964b)

Glenn Gould, (1974)

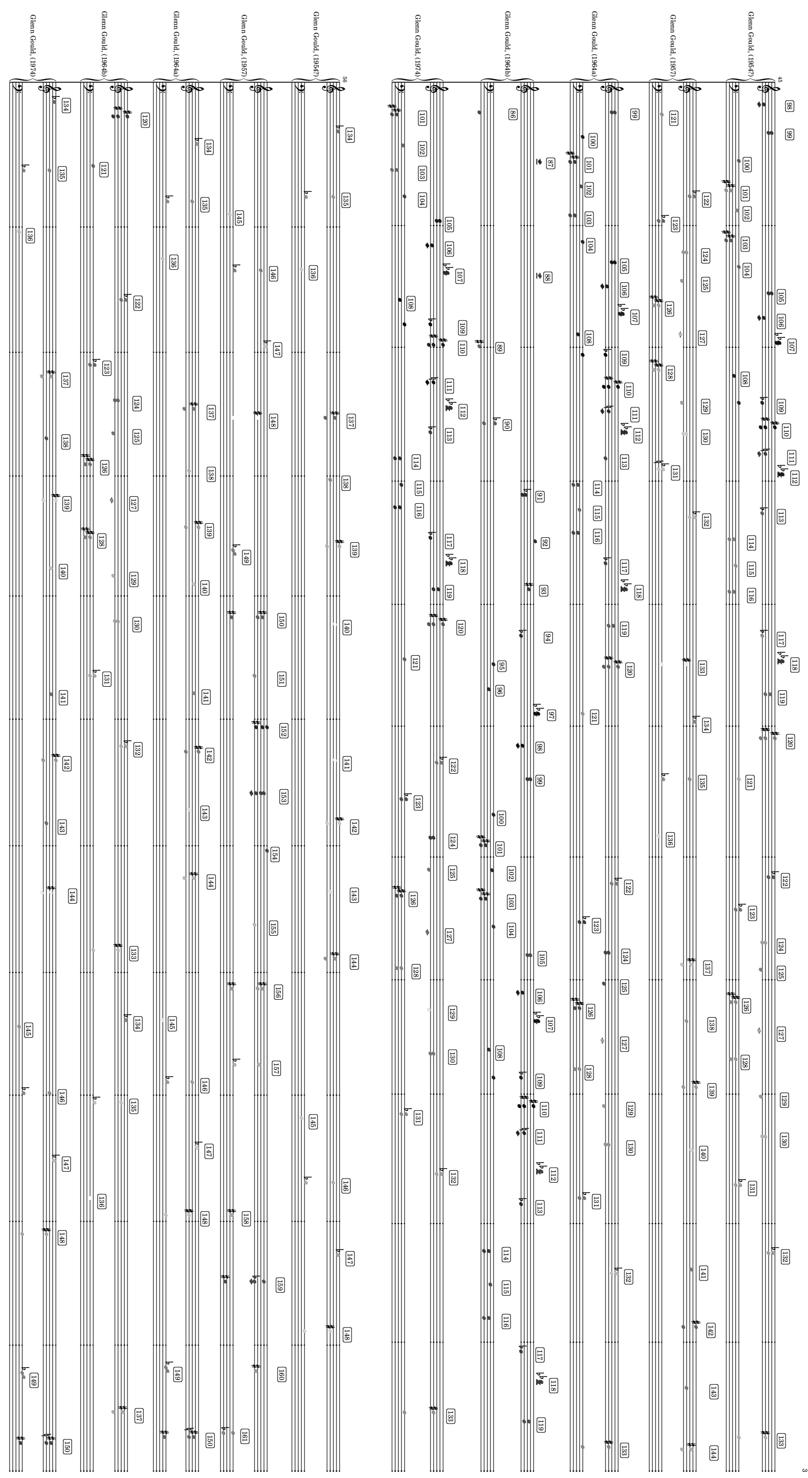
Glenn Gould, (1954?)

Glenn Gould, (1957)

Glenn Gould, (1964a)

Glenn Gould, (1964b)

Glenn Gould, (1974)


Glenn Gould, (1954?)
Glenn Gould, (1957)
Glenn Gould, (1964a)
Glenn Gould, (1964b)
Glenn Gould, (1974)
Glenn Gould, (1954?)
Glenn Gould, (1957)
Glenn Gould, (1964a)
Glenn Gould, (1964b)
Glenn Gould, (1974)

Glenn Gould, (1954?)

Glenn Gould, (1957)

Glenn Gould, (1964a)

Glenn Gould, (1964b)

Glenn Gould, (1974)

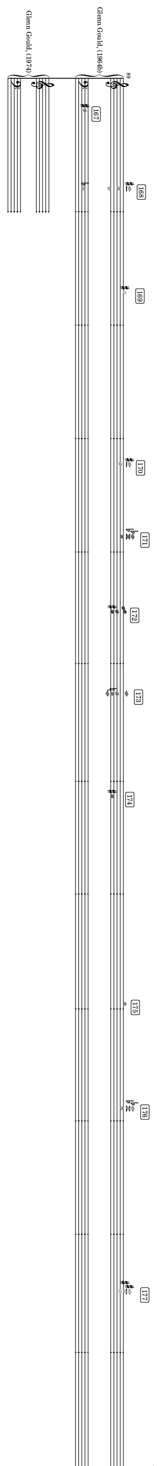
Glenn Gould, (1954?)

Glenn Gould, (1957)

Glenn Gould, (1964a)

Glenn Gould, (1964b)

Glenn Gould, (1974)


Glenn Gould, (1964b)
Glenn Gould, (1974)
167
168
169
170
171
172
173
174
175
176
177

B.4 *The World All Around* (2013) for Prepared Piano, Eb Clarinet, and Harp

The World All Around
06-04-2013

Jeff Treviño

2

Clar. *ppp* F

Pno. *mp*

Hp. *pp*

Clar. *mp* E

Pno. *ppp* *una corda*

Hp. *f*

Clar. *p* C♯

Pno. *f* *tre corde*

Hp. *ppp*

Clar. *pp* C♯ G♯

Pno. *mp*

Hp. *mf* *pp* D *p*

3

Clar.
mp
G♯
Pno.
mf
pp
una corda
Hp.
mp

Clar.
p
C♯
F♯
Pno.
f
tre corde
Hp.
ppp

Clar.
pp
G♯
Pno.
p
una corda
f
tre corde
Hp.
mf

Clar.
ppp
G♯
Pno.
mp
Hp.
mp
p

4

Clar.

Pno.

Hp.

F

una corda

C♯

A

tre corde

5

Clar.

p

E

Pno.

ppp

mp

una corda

Hp.

mf

Clar.

pp

Pno.

mf

Hp.

mp

C

Clar.

ppp

Pno.

ppp

tre corde

Hp.

f

Clar.

pp

Pno.

mf

Hp.

ppp

E

6

Clar.

mp

Pno.

pp

Hp.

f

f

mf

Clar.

pp

E

Pno.

f

Hp.

f

f

Clar.

p

Pno.

ppp

Hp.

mf

ppp

Clar.

mp

4

G♯

Pno.

mp

Hp.

pp

7

Clar.

Pno.

Hp.

Clar.

Pno.

Hp.

Clar.

Pno.

Hp.

Clar.

Pno.

una corda

tre corde

Hp.

8

Clar.
ppp
F
Pno.
p
Hp.
mf

Clar.
pp
C
Pno.
mf
Hp.
mp

Clar.
ppp
G♯
B
Pno.
mp
f
una corda
tre corde
Hp.
p

Clar.
mp
F
A
Pno.
ppp
una corda
Hp.
mp

9

Clar.
pp
Pno.
mf
Hp.
mf

Clar.
mp
F
Pno.
Hp.
ppp

Clar.
p
F
Pno.
mp
tre corde
pp
mf
Ped.
Hp.
f

Clar.
mp
Pno.
Hp.

10

Clar.

Pno.

una corda

Hp.

Clar.

Pno.

Hp.

Clar.

Pno.

tre corde

Hp.

Clar.

Pno.

Hp.

11

Clar.

mp

Pno.

p

Hp.

ppp

Bibliography

Abelson, H. and G.J. Sussman (1983). “Structure and Interpretation of Computer Programs”. In: (cit. on p. 3).

Abrams, Steven et al. (1999). “Higher-level Composition Control in Music Sketcher: Modifiers and Smart Harmony”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 14).

Acevedo, A.G. (2005). “Fugue Composition with Counterpoint Melody Generation Using Genetic Algorithms”. In: *Computer music modeling and retrieval: Second International Symposium, CMMR 2004, Esbjerg, Denmark, May 26-29, 2004: revised papers*. Springer-Verlag New York Inc, p. 96. ISBN: 3540244581. URL: http : / / books . google . com / books ? hl = en \ & lr = \ & amp ; id = zXegsi7tj00C \ & oi = fnd \ & pg = PA96 \ & dq = Fugue + Composition + with + Counterpoint + Melody + Generation + Using + Genetic + Algorithms \ & ots = Z4DpBjPMN - \ & amp ; sig = ocgSVVHiDIB7GJfmznh1Z3OheUA (cit. on pp. 15, 31).

Agon, Carlos, Gérard Assayag, and Jean Bresson, eds. (2008). *The OM Composer’s Book. 2*. IRCAM-Centre Pompidou (cit. on p. 20).

Agon, Carlos, Gérard Assayag, Jean Bresson, and Miller Puckette (2006). *The OM Composer’s Book*. IRCAM-Centre Pompidou (cit. on p. 20).

Agon, Carlos, Marco Stroppa, and Gerard Assayag (2000). “High-level Musical Control of Sound Synthesis in OpenMusic”. In: *Proceedings of the International Computer Music Conference*, pp. 332–335 (cit. on p. 19).

Alberman, David (2005). “Abnormal Playing Techniques in the String Quartets of Helmut Lachenmann”. In: *Contemporary Music Review* 24.1, pp. 39–51 (cit. on p. 33).

anarkomposer SCM Repository. URL: http://anarkomposer.svn.sourceforge.net/ (cit. on p. 32).

Anders, Torsten and Eduardo R Miranda (2008a). “Constraint-based Composition in Realtime”. In: *Proceedings of International Computer Music Conference* (cit. on p. 22).

— (2008b). “Higher-order Constraint Applicators for Music Constraint Programming”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 22).

Anders, Torsten and Eduardo R. Miranda (Oct. 2011). “Constraint Programming Systems for Modeling Music Theories and Composition”. In: *ACM Computing Surveys* 43.4, 30:1–30:38. ISSN: 0360-0300. DOI: 10.1145/1978802.1978809. URL: http://doi.acm.org/10.1145/1978802.1978809 (cit. on p. 15).

Ariza, Christopher (2003). “Ornament as Data Structure : An Algorithmic Model Based on Micro-Rhythms of Csángó Laments and Funeral Music Music of the Csángó”. In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

— (2005a). *An Open Design for Computer-Aided Algorithmic Composition: athenacl*. Vol. 54. Dissertation. com, p. 258. ISBN: 1581122926. URL: http://books.google.com/books?hl=en\&lr=\&id=XukW-mq76mcC\&oi=fnd\&pg=PR3\&dq=An+Open+Design+for+Computer-Aided+Algorithmic+Composition:+athenacl\&ots=bHedXym8ZP\&sig=9i2RQINqIVr2Y7sjxeD9e74myxA (cit. on p. 12).

— (2005b). “Navigating The Landscape Of Computer-aided Algorithmic Composition Systems: A Definition , Seven Descriptors , and a Lexicon Of Systems And Research”. In: *Proceedings of the International Computer Music Conference*, pp. 765–772 (cit. on p. 12).

Aspray, W.F. (1985). “The Scientific Conceptualization of Information: A Survey”. In: *Annals of the History of Computing* 7.2, pp. 117–140 (cit. on p. 6).

Assayag, Gérard et al. (1999). “Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic”. English. In: *Computer Music Journal* 23.3, pp. 59–72. ISSN: 01489267. URL: http://www.jstor.org/stable/3681240 (cit. on p. 19).

Attali, Jacques (1985). *Noise: The Political Economy of Music*. Manchester University Press (cit. on p. 36).

AVID. *Plugins for Sibelius*. URL: http://www.sibelius.com/download/plugins/index.html?help=write (cit. on p. 16).

Azziggoti, Luciano (2012). *Personal Communication* (cit. on p. 21).

Bača, Trevor (2010). *Personal Communication* (cit. on p. 25).

— (2011). “Chapter 4: Containers”. In: *Abjad Reference Manual* (cit. on p. 39).

— (2013). *Personal Communication* (cit. on p. 22).

Backus, John (1978). “Multiphonic Tones in the Woodwind Instruments”. In: *The Journal of the Acoustical Society of America* 63, p. 591 (cit. on p. 33).

Balachandran, Sudarshan and Lonce Wyse (2012). “Computer-mediated Visual Communication in Live Musical Performance: What’s the Score?” In: *Arts and Technology*. Ed. by Anthony L. Brooks. Vol. 101. Springer, pp. 54–62. ISBN: 978-3-642-33328-6. DOI: 10.1007/978-3-642-33329-3_7. URL: http://dx.doi.org/10.1007/978-3-642-33329-3_7 (cit. on p. 21).

Balser, Klaus and Bernd Streisberg (1990). “Counterpoint Compositions in Non-tempered Systems: Theory and Algorithms”. In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Barlow, Clarence (2011). *On Musiquantics*. Unpublished English translation from the German *Von der Musiquantenlehre* (cit. on p. 39).

Barrett, G.D. and M. Winter (2010). “LiveScore: Real-Time Notation in the Music of Harris Wulfson”. In: *Contemporary Music Review* 29.1, pp. 55–62 (cit. on p. 21).

Beazley, David M and et. al. (1996). “SWIG: An Easy-to-use Tool for Integrating Scripting Languages with C and C++”. In: *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pp. 129–139 (cit. on p. 31).

Bell, Malcolm E. (1995). “A MAX Counterpoint Generator for Simulating Stylistic Traits of Stravinsky, Bartok, and Other Composers”. In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Berger, Jonathan (1994). *Playing with” Playing with Signs”: A Critical Response to Kofi Agawu* (cit. on p. 32).

Blackwell, Alan F. (2006). “Metaphors We Program By: Space, Action and Society in Java”. In: *Proceedings of PPIG 2006*, pp. 7–21 (cit. on p. 39).

Bobrow, Daniel G. et al. (1986). “CommonLoops: Merging Lisp and Object-oriented Programming”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPLSA ’86. Portland, Oregon, USA: ACM, pp. 17–29. ISBN: 0-89791-204-7. DOI: 10.1145/28697.28700. URL: http://doi.acm.org/10.1145/28697.28700 (cit. on p. 9).

Boenn, Georg et al. (2009). “Anton: Composing Logic and Logic Composing”. In: *Logic Programming and Nonmonotonic Reasoning*. Springer, pp. 542–547 (cit. on p. 14).

Bresson, Jean and Carlos Agon (2010). “Processing Sound and Music Description Data Using OpenMusic”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 19).

Brün, Herbert (1969). “Infraudibles”. In: *Music by Computers*. Ed. by von Förster Heinz. J. Wiley (cit. on p. 15).

Burraston, Dave et al. (2004). “Cellular Automata in MIDI-based Computer Music”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 31).

Buse, Raymond P. L. and Westley R Weimer (2010). “Learning a Metric for Code Readability”. In: *IEEE Transactions on Software Engineering* 36.4, pp. 546–558 (cit. on p. 28).

Buteau, Chantal and John Vipperman (2009). “Melodic Clustering within Motivic Spaces: Visualization in OpenMusic and Application to Schumann’s Träumerei”. In: *Mathematics and Computation in Music*, pp. 59–66 (cit. on p. 19).

Butt, John (2002). *Playing with History: The Historical Approach to Musical Performance*. Cambridge University Press (cit. on p. 84).

Cage, John (2011). “Experimental Music: Doctrine”. In: *Silence: Lectures and Writings*. Wesleyan University Press (cit. on p. 13).

Cannam, Chris et al. (2006). “The Sonic Visualiser: A Visualisation Platform for Semantic Descriptors from Musical Signals”. In: *Proceedings of the 7th International Conference on Music Information Retrieval*, pp. 324–327 (cit. on p. 84).

Cantor, Don (1971). “A Computer Program that Accepts Common Musical Notation”. In: *Computers and the Humanities* 6.2, pp. 103–109 (cit. on pp. 17, 27).

Cardelli, L., P. Wegner, et al. (1985). “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM computing surveys* 17.4, pp. 471–522 (cit. on p. 3).

Cardew, Cornelius (1961). “Notation–Interpretation, Etc.” In: *Tempo* 58, pp. 21–24 (cit. on p. 37).

Cassidy, Aaron (2004). “Performative Physicality and Choreography as Morphological Determinants”. In: (cit. on p. 33).

Cetta, Pablo (2011). *Personal Communication* (cit. on p. 39).

Chico-Töpfer, Wolfgang (1998). "AVA: An Experimental, Grammar/Case-based Composition System to Variate Music Automatically Through the Generation of Scheme Series". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Collins, Nick (2003). "A Microtonal Tempo Canon Generator After Nancarrow and Jaffe". In: *Proceedings of the International Computer Music Conference* (cit. on p. 15).

— (Feb. 2009). "Musical Form and Algorithmic Composition". In: *Contemporary Music Review* 28.1, pp. 103–114. ISSN: 0749-4467 (cit. on pp. 15, 17).

Collins, Nicolas (2011). "Beyond Notation: Communicating Music". In: *Leonardo Music Journal* 21, pp. 5–6 (cit. on p. 21).

Consortium, NIFF and et al. (1995). *NIFF 6a: Notation Interchange File Format*. Tech. rep. NIFF Consortium (cit. on p. 17).

Cook, Nicholas (2007). "Performance Analysis and Chopin's Mazurkas". In: *Musicae Scientiae* 11.2, pp. 183–208 (cit. on p. 84).

— (2010). "The Ghost in the Machine: Towards a Musicology of Recordings". In: *Musicae Scientiae* 14.2, pp. 3–21 (cit. on p. 84).

Cope, D. (2002). "Computer Analysis and Computation Using Atonal Voice-Leading Techniques". In: *Perspectives of New Music* 40.1, pp. 121–146. ISSN: 0031-6016. URL: http://www.jstor.org/stable/833550 (cit. on p. 15).

Cope, David (2010). *Personal Communication* (cit. on p. 40).

Courtot, Francis (1990). "A Constraint-based Logic Program for Generating Polyphonies". In: *Proceedings of International Computer Music Conference* (cit. on p. 22).

Creasey, David P., David M. Howard, and Andrew M. Tyrrell (1996). "The Timbral Object - An Alternative Route to the Control of Timbre Space". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Creasy, R. J. (Sept. 1981). "The Origin of the VM/370 Time-Sharing System". In: *IBM Journal of Research and Development* 25.5, pp. 483 –490. ISSN: 0018-8646. DOI: 10.1147/rd.255.0483 (cit. on p. 9).

Cuthbert, Michael (2013). *Personal Communication* (cit. on p. 98).

Dannenberg, Roger B. (1993). "Music Representation Issues, Techniques, and Systems". English. In: *Computer Music Journal* 17.3, pp. 20–30. ISSN: 01489267. URL: http://www.jstor.org/stable/3680940 (cit. on p. 35).

Davies, S. (1994). *Musical Meaning and Expression*. Cornell University Press (cit. on p. 12).

Degazio, Bruno (1996). "A Computer-based Editor for Lerdahl and Jackendoff's Rhythmic Structures". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Derniame, Jean-Claude, Badara A Kaba, and David Wastell (1999). *Software Process: Principles, Methodology, and Technology*. Springer (cit. on p. 13).

Desainte-Katherine, M. and R. Strandh (1991). "The Architecture of a Musical Composition System Based on Constraint Resolution and Graph Rewriting". In: *Proceedings of International Computer Music Conference* (cit. on p. 22).

Didkovsky, Nick and Georg Hajdu (2008). "MaxScore: Music Notation in Max/MSP". In: *Proceedings of the International Computer Music Conference*, pp. 483–486 (cit. on p. 19).

Diener, Glendon (1989). "Nutation: Structural Organization Versus Graphical Generality in a Common Music Notation Program". In: *Proceedings of International Computer Music Conference* (cit. on pp. 19, 37).

— (1991a). "Addendum: A Hierarchical Approach to Music Notation". In: *The Well-tempered Object: Musical Applications of Object-oriented Software Technology*. Ed. by Steven Travis Pope. MIT Press (cit. on p. 19).

— (1991b). "TTrees: A Tool for the Compositional Environment". In: *The Well-tempered Object: Musical Applications of Object-oriented Software Technology*. Ed. by Steven Travis Pope. MIT Press (cit. on p. 19).

Dijkstra, Edsger W. (Mar. 1968). "Letters to the Editor: Go to Statement Considered Harmful". In: *Commun. ACM* 11.3, pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. URL: http://doi.acm.org/10.1145/362929.362947 (cit. on p. 8).

Dobrian, Christopher (1995). "Algorithmic Generation of Temporal Forms: Hierarchical Organization of Stasis and Transition". In: *Proceedings of the International Computer Music Conference* (cit. on p. 14).

Dunbar-Hester, C. (Dec. 2009). "Listening to Cybernetics: Music, Machines, and Nervous Systems, 1950-1980". In: *Science, Technology & Human Values* 35.1, pp. 113–139. ISSN: 0162-2439. DOI: 10.1177/0162243909337116. URL: http://sth.sagepub.com/cgi/doi/10.1177/0162243909337116 (cit. on p. 6).

Ebcioglu, K. (1980). "Computer Counterpoint". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Echevarría, David (2013). *Personal Communication* (cit. on p. 32).

Essl, Georg (2006). “Circle Maps as Simple Oscillators for Complex Behavior”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 31).

Evarts, John (1968). “The New Musical Notation – A Graphic Art?” In: *Leonardo* 1.4, pp. 405–412 (cit. on p. 37).

Farbood, Mary and Bernd Schoner (2001). “Analysis and Synthesis of Palestrina-style Counterpoint using Markov Chains”. In: *Proceedings of the International Computer Music Conference*, pp. 471–474 (cit. on p. 15).

Foster, Campbell D. (1995). “A Consonance Dissonance Algorithm for Intervals”. In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Gartland-Jones, Andrew and Peter Copley (Sept. 2003). “The Suitability of Genetic Algorithms for Musical Composition”. In: *Contemporary Music Review* 22.3, pp. 43–55. ISSN: 0749-4467. DOI: 10 . 1080 / 0749446032000150870. URL: http : / / www . informaworld . com / openurl ? genre = article \ &doi = 10 . 1080 / 0749446032000150870 \ &magic = crossref | |D404A21C5BB053405B1A640AFFD44AE3 (cit. on p. 31).

Good, Michael (2001). “MusicXML for Notation and Analysis”. In: *The Virtual Score: Representation, Retrieval, Restoration*. Ed. by Walter B. Hewlett and Eleanor Selfridge-Field. Computing in Musicology 12. MIT Press, pp. 113–124 (cit. on p. 17).

Gosling, J. (2000). *The Java Language Specification*. Prentice Hall (cit. on p. 11).

Goulthorpe, Mark (2011). “Digital Recursions”. In: *Testing to Failure: Design and Research in MIT’s Department of Architecture*. Ed. by Sarah M. Hirschman. SA+P Press (cit. on p. 130).

Gräf, Albert (2006). “On Musical Scale Rationalization”. In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Gurevich, M. and J. Treviño (2007). “Expression and Its Discontents: Toward an Ecology of Musical Creation”. In: *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*. ACM, pp. 106–111 (cit. on p. 27).

Hamanaka, Masatoshi, Keiji Hirata, and Satoshi Tojo (2005). “Automatic Generation of Metrical Structure Based on GTTM”. In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Hamel, Keith (1997). *NoteAbility Reference Manual* (cit. on p. 16).

Harley, James (2004). *Xenakis: His Life in Music*. New York: Routledge (cit. on p. 57).

Hedelin, Fredrik (Nov. 2008). "Formalising Form: An Alternative Approach To Algorithmic Composition". In: *Organised Sound* 13.03, p. 249. ISSN: 1355-7718. DOI: 10.1017/S1355771808000344. URL: http://www.journals.cambridge.org/abstract_S1355771808000344 (cit. on p. 32).

Herrera, Perfecto et al. (2005). "MUCOSA: A Music Content Semantic Annotator". In: *Proceedings of the 6th International Conference on Music Information Retrieval*, pp. 77–83 (cit. on p. 84).

Hiller L. A., Jr. and R. A. Baker (1965). "Automated Music Printing". English. In: *Journal of Music Theory* 9.1, pp. 129–152. ISSN: 00222909. URL: http://www.jstor.org/stable/843151 (cit. on p. 14).

Holzner, S. (1999). *Perl Core Language Little Black Book*. Coriolis Group Books (cit. on p. 11).

Hoos, Holger H et al. (1998). "The GUIDO Notation Format- A Novel Approach for Adequatly Representing Score-level Music". In: *Proceedings of International Computer Music Conference* (cit. on p. 16).

Horenstein, Stephen (2004). "Understanding Supersaturation : A Musical Phenomenon Affecting Perceived Time". In: *Proceedings of International Computer Music Conference* (cit. on p. 14).

Hornel, Dominik (1993). "SYSTHEMA - Analysis and Automatic Synthesis of Classical Themes". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Hörnel, Dominik (1997). "A Neural Organist Improvising Baroque-style Melodic Variations". In: *Proceedings of International Computer Music Conference* (cit. on p. 31).

Ingalls, Daniel H. H. (1978). "The Smalltalk-76 Programming System Design and Implementation". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: ACM, pp. 9–16. DOI: 10.1145/512760.512762. URL: http://doi.acm.org/10.1145/512760.512762 (cit. on p. 10).

Ingalls, D.H.H. (1981). "Design Principles Behind Smalltalk". In: *Byte Magazine* 6.8, pp. 286–298 (cit. on p. 10).

Jones, David Evan (2000). "A Computational Composer's Assistant for Atonal Counterpoint". English. In: *Computer Music Journal* 24.4, pp. 33–43. ISSN: 01489267. URL: http://www.jstor.org/stable/3681553 (cit. on p. 15).

Kanno, Mieko (2007). “Prescriptive Notation: Limits and Challenges”. In: *Contemporary Music Review* 26.2, pp. 231–254 (cit. on p. 33).

Kay, Alan C. (1996). “The Early History of Smalltalk”. In: *HOPL-II: The Second ACM SIGPLAN Conference on the History of Programming Languages*. Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM. Chap. The Early History of Smalltalk, pp. 511–598. ISBN: 0-201-89502-1. DOI: 10.1145/234286.1057828. URL: http://doi.acm.org/10.1145/234286.1057828 (cit. on p. 9).

Kelly, Edward (2011). “Gemnotes – A Real-time Notation System for PD”. In: *Proceedings of the 4th Annual Pure Data Convention (Weimar-Berlin)* (cit. on p. 19).

Kiviat, Philip (1993). “A Brief Introduction to Discrete-event Simulation Programming Languages”. In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: ACM, pp. 369–370. ISBN: 0-89791-570-4. DOI: 10.1145/154766.155400. URL: http://doi.acm.org/10.1145/154766.155400 (cit. on p. 8).

Kowalski, Robert (1979). “Algorithm = Logic + Control”. In: *Communications of the ACM* 22.7, pp. 424–436 (cit. on p. 22).

Krasner, Glen (1991). “Machine Tongues VIII: The Design of a Smalltalk Music System”. In: *The Well-tempered Object: Musical Applications of Object-oriented Software Technology*. Ed. by Steven Travis Pope. MIT Press (cit. on p. 17).

Kröger, Pedro et al. (2008). “Rameau: A System for Automatic Harmonic Analysis”. In: *Information Retrieval* (cit. on p. 31).

Kunze, Tobias and Heinrich Taube (1996). “See—A Structured Event Editor: Visualizing Compositional Data in Common Music”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 84).

Kuuskankare, Mika (2009). “ENP: A System for Contemporary Music Notation”. In: *Contemporary Music Review* 28.2, pp. 221–235 (cit. on p. 20).

— (2012a). “Meta-Score, a Novel PWGL Editor Designed for the Structural, Temporal, and Procedural Description of a Musical Composition”. In: *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference* (cit. on p. 20).

— (2012b). “The Visual SDIF Interface in PWGL”. In: *Proceedings of the 9th International Symposium on Computer Music Modeling and Retrieval* (cit. on p. 20).

Kuuskankare, Mika and Mikael Laurson (2006). “Expressive Notation Package”. English. In: *Computer Music Journal* 30.4, pp. 67–79. ISSN: 01489267. URL: http://www.jstor.org/stable/4617984 (cit. on p. 16).

— (2010). “Connecting Graphical Scores To Sound Synthesis In PWGL”. In: *Proceedings of the 7th Sound and Music Computing Conference* (cit. on p. 20).

Laine, Pauli (1997). “Generating Musical Patterns Using Mutually Inhibited Artificial Neurons”. In: *Proceedings of the International Computer Music Conference* (cit. on p. 31).

Lakoff, George (1980). *Metaphors We Live By*. Chicago: University of Chicago Press. ISBN: 9780226468013. URL: http://www.worldcat.org/title/metaphors-we-live-by/oclc/6042798\&referer=brief_results (cit. on p. 32).

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Prentice Hall (cit. on p. 11).

Laurson, Mikael and Mika Kuuskankare (2003). “Some Box Design Issues in PWGL”. In: *Proceedings of International Computer Music Conference* (cit. on p. 20).

— (2005). “Extensible Constraint Syntax Through Score Accessors”. In: *Journées d’Informatique Musicale*, pp. 27–32 (cit. on p. 15).

— (2006). “Recent Trends in PWGL”. In: *Proceedings of International Computer Music Conference* (cit. on p. 20).

Laurson, Mikael, Vesa Norilo, and Mika Kuuskankare (2005). “PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control”. In: *Computer Music Journal* 29.3, pp. 29–41 (cit. on p. 20).

Liskov, Barbara and Stephen Zilles (1974). “Programming with Abstract Data Types”. In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. Santa Monica, California, USA: ACM, pp. 50–59. DOI: 10.1145/800233.807045. URL: http://doi.acm.org/10.1145/800233.807045 (cit. on p. 5).

Lloyd, John W (1994). “Practical Advantages of Declarative Programming”. In: *Joint Conference on Declarative Programming, GULP-PRODE*. Vol. 94, p. 94 (cit. on p. 22).

Lopez-Montes, José (2011). *Personal Communication* (cit. on p. 39).

Luque, Sergio (Dec. 2009). “The Stochastic Synthesis of Iannis Xenakis”. In: *Leonardo Music Journal* 19, pp. 77–84. ISSN: 0961-1215. DOI: 10.1162/lmj.2009.19.77. URL: http://www.mitpressjournals.org/doi/abs/10.1162/lmj.2009.19.77 (cit. on pp. 15, 31, 58).

Lystad, Mary (1989). “Taming the Wild Things”. In: *Children Today* 18.2, pp. 16–19 (cit. on p. 124).

Magnus, Cristyn (2010). "Evolutionary Sound: a Non-Symbolic Approach to Creating Sonic Art With Genetic Algorithms". PhD thesis. University of California, San Diego (cit. on p. 31).

Magnus, Wilhelm, Abraham Karrass, and Donald Solitar (2004). *Combinatorial Group Theory: Presentations of Groups in Terms of Generators and Relations*. Courier Dover Publications (cit. on p. 57).

Mahling, Andreas (1991). "How to Feed Musical Gestures into Compositions". In: *Proceedings of International Computer Music Conference* (cit. on p. 18).

Marsden, Alan et al. (2007). "Tools for Searching, Annotation and Analysis of Speech, Music, Film and Video—A Survey". In: *Literary and Linguistic Computing* 22.4, pp. 469–488 (cit. on p. 84).

Martin, Robert C. *Principles of OOP*. URL: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (cit. on p. 11).

Mathews, Max V. (1983). "RTSKED, A Real-time Scheduled Language for Controlling a Music Synthesizer". In: *The Journal of the Acoustical Society of America* 74, S60 (cit. on p. 19).

Mathews, Max V. and J. Pasquale (1981). "RTSKED, A Scheduled Performance Language for the Crumar General Development System". In: *Proceedings of International Computer Music Conference* (cit. on p. 19).

McLean, A. and Geraint Wiggins (2010). "Bricolage Programming in the Creative Arts". In: *22nd Psychology of Programming Interest Group*. URL: http://yaxu.org/tmp/ppig.pdf (cit. on p. 30).

Melo, A. F. and Geraint Wiggins (2003). "A Connectionist Approach to Driving Chord Progressions Using Tension". In: *Proceedings of the AISB*. Vol. 3. 1988. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9086\&rep=rep1\&type=pdf (cit. on pp. 15, 31).

Miranda, Eduardo (2007). "Cellular Automata Music: From Sound Synthesis to Musical Forms". In: *Evolutionary Computer Music*. Ed. by Eduardo Reck Miranda. Springer, pp. 170–193. URL: http://www.springerlink.com/index/R20003856G6874Q2.pdf (cit. on p. 31).

Mitroo, J. B., Nancy Herman, and Norman I Badler (1979). "Movies from Music: Visualizing Musical Compositions". In: *ACM SIGGRAPH Computer Graphics*. Vol. 13. 2. ACM, pp. 218–225 (cit. on p. 84).

Morris, Robert (2002). *How Does Using Music Notation Software Affect Your Music?* URL: http://www.newmusicbox.org/articles/How-does-using-music-notation-software-affect-your-music-Robert-Morris/ (cit. on p. 36).

Nance, R.E. and R.G. Sargent (2002). "Perspectives on the Evolution of Simulation". In: *Operations Research* 50.1, pp. 161–172 (cit. on pp. 4, 7, 8).

Nauert, Paul (1994). "A Theory of Complexity to Constrain the Approximation of Arbitrary Sequences of Timepoints". In: *Perspectives of New Music*, pp. 226–263 (cit. on p. 91).

— (Dec. 2007). "Division- and Addition-based Models of Rhythm in a Computer-Assisted Composition System". In: *Computer Music Journal* 31.4, pp. 59–70. ISSN: 0148-9267. DOI: 10.1162/comj.2007.31.4.59. URL: http://www.mitpressjournals.org/doi/abs/10.1162/comj.2007.31.4.59 (cit. on p. 15).

Nienhuys, Han-Wen and Jan Nieuwenhuizen (2003). "LilyPond, A System for Automated Music Engraving". In: *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*. Citeseer, pp. 167–172 (cit. on pp. 16, 24).

Norman, D.A. (2003). *Emotional Design: Why We Love (or Hate) Everyday Things*. Basic books (cit. on p. 4).

Oberholtzer, Josiah (2010). *Personal Communication* (cit. on p. 30).

Orledge, R (Aug. 1998). "Understanding Satie's 'Vexations'". In: *Music and Letters* 79.3, pp. 386–395. ISSN: 0027-4224. DOI: 10.1093/ml/79.3.386. URL: http://ml.oupjournals.org/cgi/doi/10.1093/ml/79.3.386 (cit. on p. 32).

Osaka, Naotoshi (2004). "Toward Construction of a Timbre Theory for Music Composition Composition". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Pärt, Arvo (1980). *Cantus in Memory of Benjamin Britten*. Philharmonia Series 555. Universal Edition (cit. on p. 40).

Pearce, Marcus, Darrell Conklin, and Geraint Wiggins (2005). "Methods for Combining Statistical Models of Music". In: *Computer Music Modeling and Retrieval*, pp. 295–312. URL: http://www.springerlink.com/index/CPTVYB2CC735HDX8.pdf (cit. on p. 33).

Pearce, Marcus, David Meredith, and Geraint Wiggins (2002). "Motivations and Methodologies for Automation of the Compositional Process". In: *Musicae Scientiae* 6.2, pp. 119–148. ISSN: 1029-8649. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.9989\&rep=rep1\&type=pdf (cit. on p. 15).

Peters, Michael (Aug. 2010). "From Strange to Impossible: Interactive Attractor Music". In: *Contemporary Music Review* 29.4, pp. 395–404. ISSN: 0749-4467. DOI: 10.1080/07494467.2010.587317. URL: http://www.tandfonline.com/doi/abs/10.1080/07494467.2010.587317 (cit. on p. 31).

Phon-Amnuaisuk, S., Andrew Tuson, and Geraint Wiggins (1999). "Evolving Musical Harmonisation". In: *Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Portorož, Slovenia, 1999*. Springer Verlag Wien, p. 229. ISBN: 3211833641. URL: http://books.google.com/books?hl=en\&lr=\&id=clKwynlfZYkC\&oi=fnd\&pg=PA229\&dq=Evolving+Musical+Harmonisation\&ots=bJK6JCWmdd\&sig=SnyQUlrh8ixL-meviZuLX8lsH8A (cit. on p. 31).

Polansky, Larry (1990). "HMSL (Hierarchical Music Specification Language): A Theoretical Overview". In: *Perspectives of New Music* 28.2 (cit. on pp. 17, 18).

— (2010). *Personal Communication* (cit. on p. 15).

Polansky, Larry, Alex Barnett, and Michael Winter (2011). "A Few More Words About James Tenney: Dissonant Counterpoint and Statistical Feedback". In: *Journal of Mathematics and Music* 5.2, pp. 63–82 (cit. on p. 15).

Polansky, Larry, Martin McKinney, and Bregman Electro-Acoustic Music Studio (1991). "Morphological Mutation Functions". In: *Proceedings of the International Computer Music Conference*, pp. 234–41 (cit. on p. 14).

Polfreman, Richard (2002). "Modalys-ER for OpenMusic (MfOM): Virtual Instruments and Virtual Musicians". In: *Organised Sound* 7.3, pp. 325–338 (cit. on p. 19).

Pope, Steven Travis (1991). "Introduction to MODE: The Musical Object Development Environment". In: *The Well-tempered Object: Musical Applications of Object-oriented Software Technology*. Ed. by Steven Travis Pope. MIT Press (cit. on p. 18).

Priestley, Mark (2011). *A Science of Operations: Machines, Logic and the Invention of Programming*. History of Computing. Springer-Verlag London Limited (cit. on p. 7).

Puckette, Miller (1991). “Combining Event and Signal Processing in the MAX Graphical Programming Environment”. English. In: *Computer Music Journal* 15.3, pp. 68–77. ISSN: 01489267. URL: http://www.jstor.org/stable/3680767 (cit. on p. 19).

Puckette, Miller et al. (1996). “Pure Data: Another Integrated Computer Music Environment”. In: *Proceedings of the Second Intercollege Computer Music Concerts*, pp. 37–41 (cit. on p. 19).

Quinlan, Philip T and Richard N Wilton (1998). “Grouping by Proximity or Similarity? Competition Between the Gestalt Principles in Vision”. In: *Perception* 27, pp. 417–430 (cit. on p. 28).

Rastall, Richard (1983). *The Notation of Western Music*. St. Martin’s Press (cit. on p. 33).

Reas, Casey and Ben Fry (2007). *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press (cit. on pp. 21, 114).

Reddy, Michael J (1979). “The Conduit Metaphor: A Case of Frame Conflict in our Language about Language”. In: *Metaphor and Thought* 2, pp. 164–201 (cit. on p. 13).

reStructured Text Primer – Sphinx 1.1.3 Documentation. URL: http://sphinx-doc.org/rest.html (cit. on p. 32).

Rosen, Margit et al. (2011). *A Little Known Story about a Movement, a Magazine and the Computer’s Arrival in Art: New Tendencies and Bit International, 1961-1973*. ZKM/Center for Art, Media, and MIT Press (cit. on p. 100).

Sammet, J.E. (1991). “Some Approaches to, and Illustrations of, Programming Language History”. In: *Annals of the History of Computing* 13.1, pp. 33–50 (cit. on pp. 9, 11).

Sandred, Örjan (2010). “PWMC, a Constraint-Solving System for Generating Music Scores”. In: *Computer Music Journal* 34.2, pp. 8–24 (cit. on p. 22).

Sanner, Michel and et. al. (1999). “Python: A Programming Language for Software Integration and Development”. In: *Journal of Molecular Graphics and Modeling* 17.1, pp. 57–61 (cit. on p. 31).

Sapp, Craig Stuart (2005). “Visual Hierarchical Key Analysis”. In: *Computers in Entertainment (CIE)* 3.4, pp. 1–19 (cit. on p. 84).

— (2011). “Computational Methods for the Analysis of Musical Structure”. PhD thesis (cit. on p. 84).

Schankler, Isaac (2013). *Notational Alternatives: Beyond Finale and Sibelius*. URL: http://www.newmusicbox.org/articles/notational-alternatives-beyond-finale-and-sibelius/ (cit. on p. 24).

Schwarz, Diemo and Matthew Wright (2000). "Extensions and Applications of the SDIF Sound Description Interchange Format". In: *Proceedings of the International Computer Music Conference, Berlin, Germany*. Citeseer, pp. 481–484 (cit. on p. 20).

Schwarz, Jerry (1993). "A Brief Introduction to C++". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: ACM, pp. 349–350. ISBN: 0-89791-570-4. DOI: 10.1145/154766.155383. URL: http://doi.acm.org/10.1145/154766.155383 (cit. on p. 11).

Selfridge-Field, Eleanor (1997). *Beyond MIDI: The Handbook of Musical Codes*. The MIT Press (cit. on p. 16).

Seymour, John Chow (2007). "Computer-assisted Composition in Equal Tunings: Tonal Congnition and the *Thirteen Tone March*". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Shannon, Claude Elwood (1949). "A Mathematical Theory of Communication". In: *Bell System Technical Journal* (cit. on p. 13).

Smith, Leland (1972). "SCORE- A Musician's Approach to Computer Music". In: *Journal of the Audio Engineering Society* 20.1, pp. 7–14 (cit. on p. 16).

Smith, Matt and Simon Holland (1992). "An AI Tool for Analysis and Generation of Melodies". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

Solomos, Makis (2001). "The Unity of Xenakis's Instrumental and Electroacoustic Music: The Case for "Brownian Movements"". In: *Perspectives of New Music*, pp. 244–254 (cit. on p. 58).

Sorensen, Andrew (Feb. 2013). *impromptu home page*. URL: http://impromptu.moso.com.au/ (cit. on p. 36).

Spicer, Michael (2004). "AALIVENET : An Agent-based Distributed Interactive Composition Environment". In: *Proceedings of International Computer Music Conference* (cit. on p. 31).

Standish, Thomas A. (July 1975). "Extensibility in Programming Language Design". In: *ACM SIGPLAN Notices* 10.7, pp. 18–21. ISSN: 0362-1340. DOI: 10.1145/987305.987310. URL: http://doi.acm.org/10.1145/987305.987310 (cit. on p. 30).

Stefik, M. and D.G. Bobrow (1985). "Object-oriented programming: Themes and variations". In: *AI magazine* 6.4, p. 40 (cit. on p. 9).

Stockhausen, Karlheinz and Elaine Barkin (1962). "The Concept of Unity in Electronic Music". In: *Perspectives of New Music* 1.1, pp. 39–48 (cit. on p. 15).

Sutherland, Ivan E. (1964). "Sketch Pad: A Man-Machine Graphical Communication System". In: *Proceedings of the SHARE Design Automation Workshop*. DAC '64. New York, NY, USA: ACM, pp. 6.329–6.346. DOI: 10.1145/800265.810742. URL: http://doi.acm.org/10.1145/800265.810742 (cit. on p. 9).

Tenney, James and Larry Polansky (1980). "Temporal Gestalt Perception in Music". In: *Journal of Music Theory* 24.2 (cit. on p. 18).

Truchet, Charlotte (2004). "Contraintes, Rercherche Locale et Composition Assistée par Ordinateur". PhD thesis. Universite Paris 7 (cit. on p. 33).

Uno, Y. and R. Huebscher (1994). "Temporal-Gestalt Segmentation-Extensions for Compound Monophonic and Simple Polyphonic Musical Contexts: Application to Works by Cage, Boulez, Babbitt, Xenakis and Ligeti". In: *Proceedings of the International Computer Music Conference*, p. 7 (cit. on pp. 14, 15).

Van Rossum, G. and F. L. Drake (2003). *Python Language Reference Manual*. Network Theory Limited (cit. on p. 11).

Van Roy, Peter and Seif Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press (cit. on pp. 3, 5, 7, 22).

Washka, Rodney (2007). "Composing with Genetic Algorithms : GenDash". In: *Evolutionary Computer Music*. Ed. by Eduardo Reck Miranda. Springer, pp. 117–136 (cit. on p. 31).

Wiggins, G. (1999). "Automated Generation of Musical Harmony: What's Missing?" In: *Proceedings of the International Joint Conference on Artificial Intelligence*. URL: http://www.doc.gold.ac.uk/~mas02gw/papers/IJCAI99b.pdf (cit. on p. 15).

Wiggins, G.A. et al. (1998). "Evolutionary Methods for Musical Composition". In: *International Journal of Computing Anticipatory Systems*. ISSN: 0963-5203. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.987\&rep=rep1\&type=pdf (cit. on p. 31).

Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener (1990). *Designing Object-oriented Software*. Prentice-Hall (cit. on pp. 1, 3).

Wright, Matthew et al. (1999). "Audio Applications of the Sound Description Interchange Format Standard". In: *Proceedings of the Audio Engineering Society* (cit. on p. 20).

Wulfson, H., G.D. Barrett, and M. Winter (2007). "Automatic Notation Generators". In: *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*. ACM, pp. 346–351 (cit. on p. 21).

Xenakis, Iannis (1991). "More Thorough Stochastic Music". In: *Proceedings of International Computer Music Conference* (cit. on p. 15).

— (1992). *Formalized Music: Thought and Mathematics in Composition*. Pendragon Press (cit. on p. 13).

Yoo, Min-Joon and In-Kwon Lee (2006). "Musical Tension Curves and its Applications". In: *Proceedings of International Computer Music Conference* (cit. on p. 14).

Zad, Damon Daylamani, Babak N Araabi, and Caru Lucas (2005). "A Novel Approach to Automatic Music Composing : Using Genetic Algorithms". In: *Information Systems*, pp. 551–555 (cit. on p. 31).

Zhong, Ningyan and Yi Zheng (2005). "Constraint-Based Melody Representation". In: *Computer Music Modeling and Retrieval*, pp. 313–329. URL: http://www.springerlink.com/index/VXW8RE4KP7VAH2G9.pdf (cit. on p. 22).