CONSTRUCTING MUSICAL DOCUMENTS

In Python With Abjad

Josiah Wolf Oberholtzer

PDX Python
(Thursday 22 October 2015)

FOLLOW ALONG

Project repository

https://github.com/Abjad/abjad

Presentation repository

https://github.com/Abjad/presentations/tree/master/pdxpython

Project documentation

http://abjad.mbrsi.org

Gallery of scores

http://abjad.mbrsi.org/gallery.html

- Classically-trained composer
  - Acoustic chamber music
  - Multi-channel electro-acoustic music
- 2008-2015: Harvard, M.A., PhD, music composition
- 2014: MIT Music and Theater Arts
  - Programmer for the Music21 project
  - http://web.mit.edu/music21/
- 2006-2008: Forced Exposure (a music distributor)
- 2002-2006: Oberlin Conservatory, BMus, music composition
- I love Python

The **Abjad API for Formalized Score Control** extends the Python programming language with an open-source, object-oriented model of common-practice music notation that enables composers to build scores through the aggregation of elemental notation objects.

Python

Needs no introduction.

LaTeX

A venerable automated typesetter.

LilyPond

LaTeX-inspired automated music engraving.

Graphviz

Automated graph / network visualization.

Abjad

A lot of glue.

TABLE OF CONTENTS

A LITTLE HISTORY

Music is poetry.
It’s also math.
A lot of math.

Figure 1: *Pyth(on)agoras*: integer ratios dictating harmony

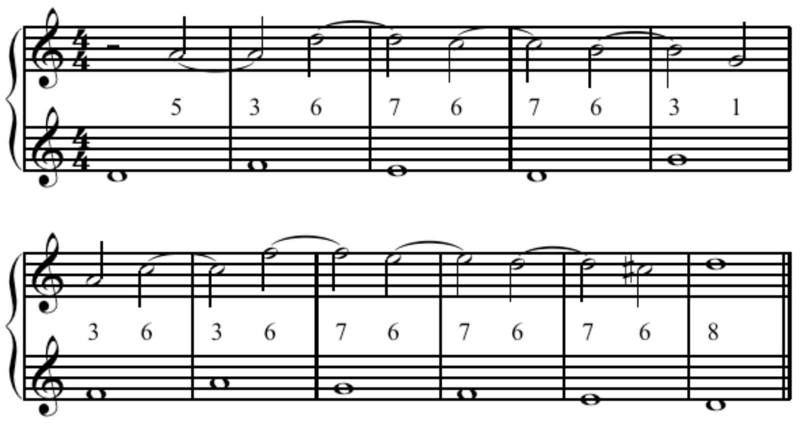
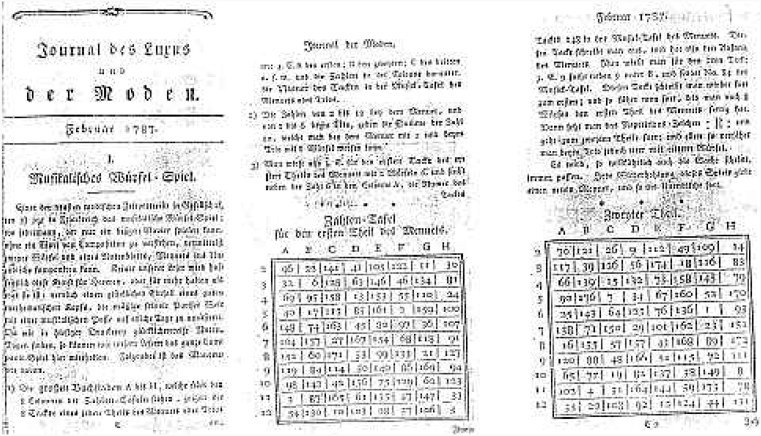


Figure 2: *Species counterpoint*: a constraint-satisfaction-problem dream



Journal des Luxus und der Moden.

Februar 1787.

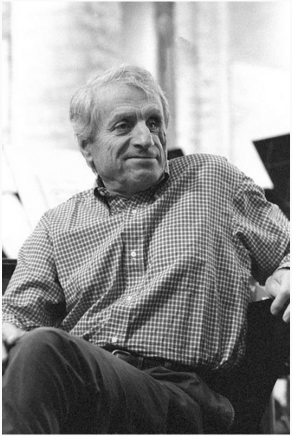
I.

Musikalisches Würfel-Spiel.

Zahlen-Tafel für den ersten Theil des Menuets.

Zweyter Theil.

Figure 3: Random minuets via Mozart's Dice Game



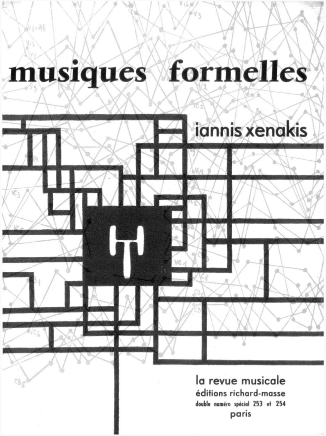


Figure 4: Iannis Xenakis (1922-2001)

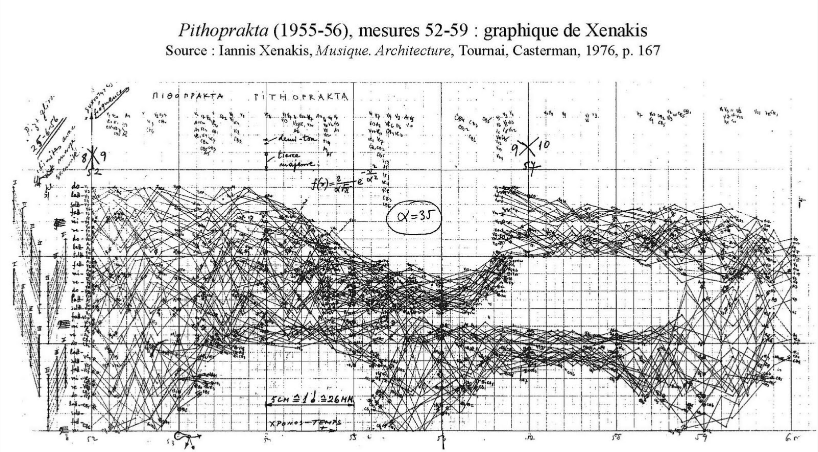


Figure 5: Stochastic string orchestra trajectories

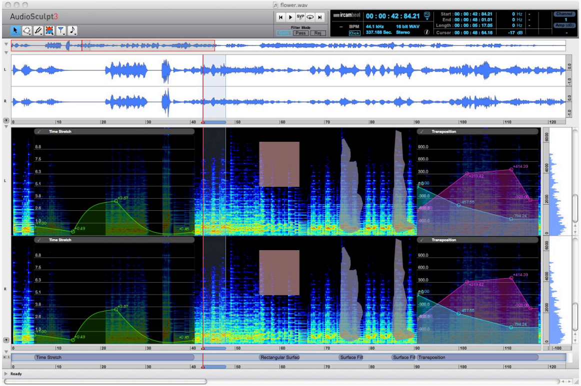


Figure 6: IRCAM's AudioSculpt

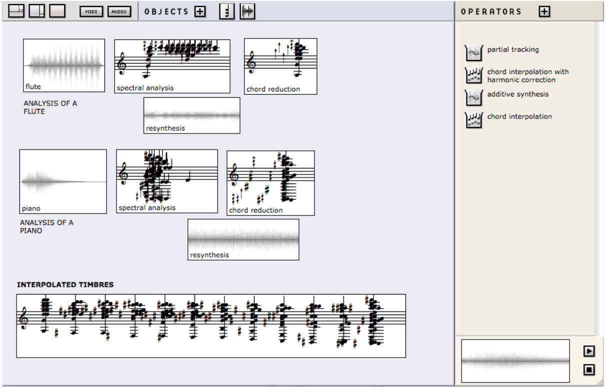


Figure 7: OpenMusic: Lisp hidden behind boxes

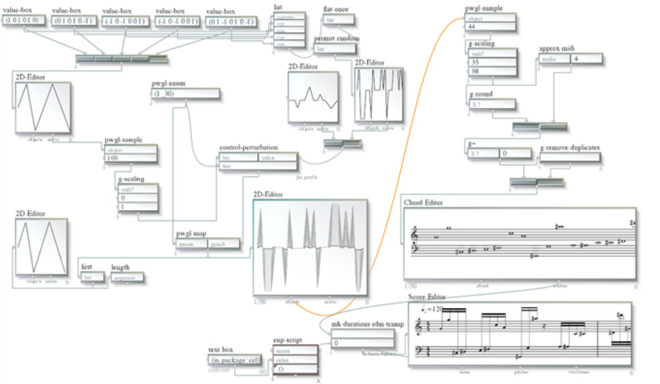


Figure 8: PWGL: yet more Lisp hidden behind boxes

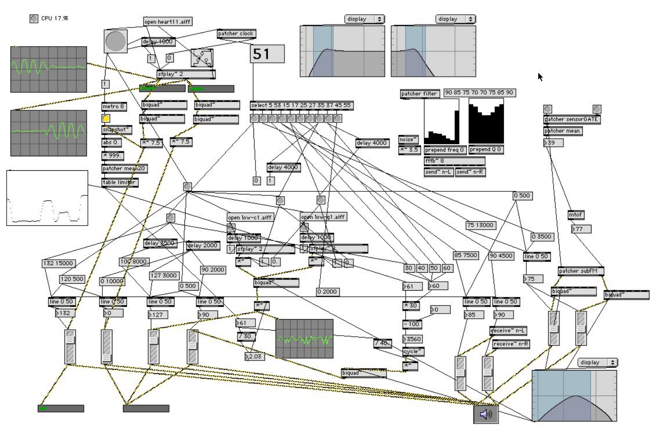


Figure 9: Max/MSP: spaghetti code as cautionary tale or design goal?

Typesetting is hard

Let’s do it anyway and probably do a bad job.

Programming is hard

Let’s pretend it doesn’t exist. That’s a good idea.

Reinventing the wheel is irresistable

I can’t help myself!

- C into Finale via MIDI (1997)
- Mathematica into Sibelius via MIDI (2001)
- Mathematica into SCORE (2003)
- Mathematica into LilyPond (2004)
- Python into Adobe Illustrator (2004)
- Python into LilyPond (2005)
- Max/MSP into MS Access into Adobe Illustrator (2008)
- Public release on GoogleCode (2008)
- Migration to GitHub (2011)
- Abjad 2.16 released (2015)

Table 1: Abjad's Software Stack

Python				
Abjad				
~~SCORE~~	LilyPond	Steinberg?	...	...

AUTOMATED TYPESETTING IS A GODSEND

LilyPond, LaTeX and Graphviz

- What-You-See-Is-What-You-Mean
- Available on the command-line
- Often extensible / modular / allow scripting
- Take plain-text as input
- Give back beautiful graphics as output

Oh, and Python isn't half-bad at…

- Writing out plain text files, and…
- Opening shells to command-line programs

SOME LIVE CODING

ABJAD'S OBJECT MODEL

Abjad models musical score as a tree of components

Containers, leaves, spanners & indicators

Relationships between objects are modeled explicitly

Parentage, lineage, logical tie, logical voice

Primitive objects are also modeled explicitly

Duration, Offset, Pitch, PitchClass, Interval, Octave, Accidental

Top-level functions expose higher-level interfaces

Inspection, iteration, selection, mutation, persistence

PARSERS

- PLY-powered
- Pervasive throughout the system
- LilyPond syntax parsing
    - Includes a Scheme parser for LilyPond’s embedded Scheme-Lisp
- IRCAM-inspired RTM-parsing
- *Reduced-LilyPond*-parsing for pedagogical examples

show(), play() and graph()

Illustratable visualization or sonification

attach(), detach()

Indicator and spanner attachment

inspect_()

Reveals inspection interface,
Accesses score-context-derived info
(How much work should properties do?)

iterate()

Reveals interation interface

mutate()

Reveals mutation interface

override(), set_()

Override and set LilyPond typographic overrides

persist()

Reveals persistence interface,
Exports objects as PNG, PDF, LilyPond, MIDI, etc.

new()

Storage-formattable object templating

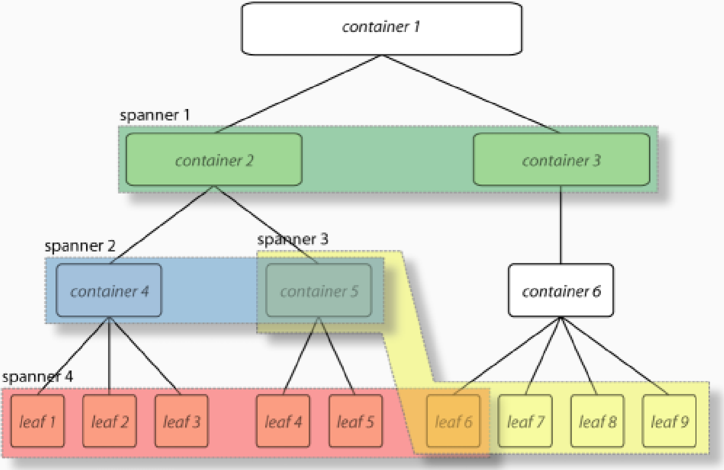
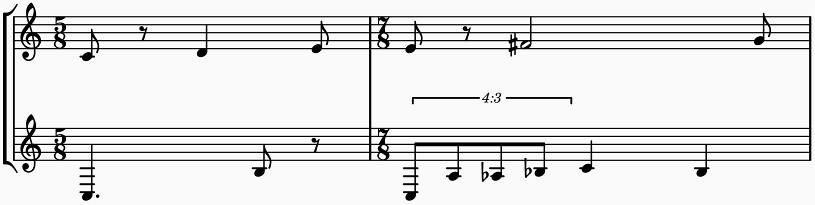


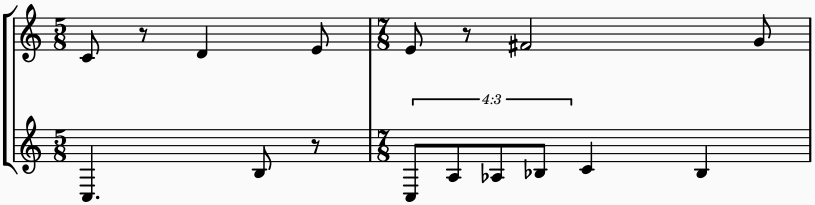
Figure 10: Spanners introducing cyclicity

A TWO VOICE EXAMPLE

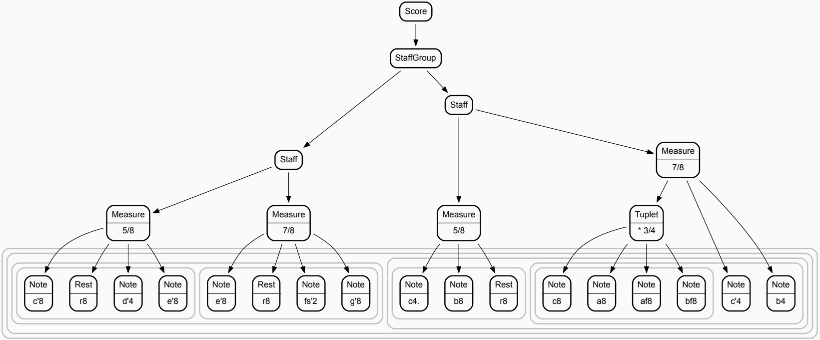
```
>>> upper_staff_string = "abj: | 5/8 c'8 r8 d'4 e'8 || 7/8 e'8 r8 fs'2 g'8 |"
>>> lower_staff_string = "abj: | 5/8 c4. b8 r8 || 7/8 3/4 { c8 a8 af8 bf8 } c'4 b4 |"
>>> upper_staff= Staff(upper_staff_string, name='Upper Staff')
>>> lower_staff = Staff(lower_staff_string, name='Lower Staff')
>>> staff_group = StaffGroup(name='Staff Group')
>>> staff_group.extend([upper_staff, lower_staff])
>>> score = Score(name='Score')
>>> score.append(staff_group)
>>> show(score)
```



```
>>> show(score)
```



```
>>> graph(score)
```


Score
StaffGroup
Staff
Staff
Measure
7/8
Measure
5/8
Measure
7/8
Measure
5/8
Tuplet
* 3/4
Note
c'8
Rest
r8
Note
d'4
Note
e'8
Note
e'8
Rest
r8
Note
fs'2
Note
g'8
Note
c4.
Note
b8
Rest
r8
Note
c8
Note
a8
Note
af8
Note
bf8
Note
c'4
Note
b4

```
>>> attach(Tempo((1, 4), 56), upper_staff[0][0])
>>> attach(Hairpin('p < f'), upper_staff[:])
>>> to_tie_together = (upper_staff[0][-1], upper_staff[1][0])
>>> attach(Tie(), to_tie_together)
>>> show(score)
```

INSPECTING COMPONENTS

```
>>> inspect_(score).get_duration()
Duration(3, 2)
```

```
>>> for component in inspect_(upper_staff[0]).get_parentage():
...     component
...
Measure((5, 8), "c'8 r8 d'4 e'8 ~")
<Staff-"Upper Staff"{2}>
<StaffGroup-"Staff Group"<<2>>>
<Score-"Score"<<1>>>
```

```
>>> inspect_(lower_staff[1]).get_timespan()
Timespan(start_offset=Offset(5, 8), stop_offset=Offset(3, 2))
```

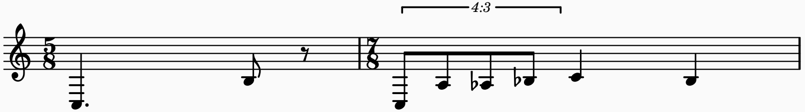
INDICATOR SCOPE

- Arbitrary objects can be attached to components
- They can be attached with *scope*
- Scoped objects *persist* until replaced
- Indicator scope can apply at different context levels

```
>>> inspect_(score[0][1][1][-1]).get_effective(Tempo)
Tempo(reference_duration=Duration(1, 4), units_per_minute=56)
```

NAMED COMPONENTS, SELECTING LEAVES

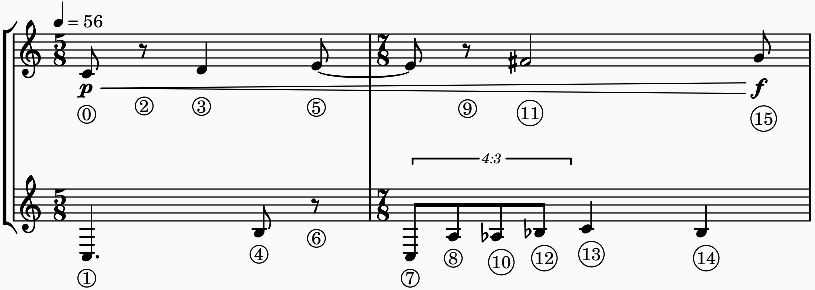
```
>>> lower_staff = score['Lower Staff']
>>> show(lower_staff)
```




```
>>> lower_leaves = lower_staff.select_leaves()
>>> inspect_(lower_leaves[-1]).get_effective(Tempo)
Tempo(reference_duration=Duration(1, 4), units_per_minute=56)
```

```
>>> iterator = iterate(score).depth_first()
>>> for i, component in enumerate(iterator):
...     print(component)
...     if 6 < i:
...         break
...
<Score-”Score”<<1>>>
<StaffGroup-”Staff Group”<<2>>>
<Staff-”Upper Staff”{2}>
Measure((5, 8), ”c’8 r8 d’4 e’8 ~”)
c’8
r8
d’4
e’8
```

```
>>> iterator = iterate(score).by_timeline_and_logical_tie()
>>> for index, logical_tie in enumerate(iterator):
...     attach(Markup(index).circle(), logical_tie.head)
...
>>> show(score)
```



```
>>> for leaf in iterate(score).by_class(scoretools.Leaf):
...     detached_markup = detach(Markup, leaf)
...
>>> show(score)
```

MARKUP

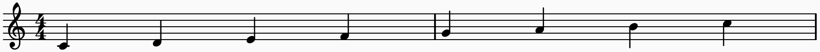
```
>>> city = Markup('Los Angeles').bold()
>>> date = Markup('May - August 2014').italic()
>>> markup = Markup.center_column([city, date])
>>> markup = markup.pad_around(1)
>>> markup = markup.box()
>>> show(markup)
```

Los Angeles
May - August 2014

```
>>> selector = selectortools.Selector().by_leaves().by_run((Note, Chord))
>>> for selection in selector.by_length('>', 1)(upper_staff):
...     attach(Slur(), selection)
...
>>> for leaf in selector[0].flatten()(upper_staff):
...     attach(Articulation('accent'), leaf)
...
>>> show(upper_staff)
```

```
>>> staff = score['Lower Staff']
>>> attach(Clef('percussion'), staff)
>>> override(staff).note_head.style = 'cross'
>>> override(staff).staff_symbol.line_positions = schemetools.SchemeVector(-4, 4)
>>> show(score)
```

```
>>> staff = Staff("c'4 d'4 e'4 f'4 g'4 a'4 b'4 c''4")
>>> show(staff)
```



```
>>> shards = mutate(staff.select_leaves()).split([(5, 16)], cyclic=True)
>>> for i, shard in enumerate(shards):
...     if i % 2:
...         mutate(shard).transpose('+P8')
...     attach(Slur(), shard)
...     attach(Articulation('accent'), shard[0])
...
>>> show(staff)
```

ABOUT THE CODE BASE

- targeted at printed music, not audio
- no floating point anywhere - it’s all rational numbers
- aims for explicitness in its design
- few dependencies
- 496 public classes
- 387 public functions
- 186,963 lines of code
- 9399 unit tests
- 10190 documentation tests
- 100% free & open source
- platform independent
- runs under both Python 2.7, 3.3+ and PyPy

A SMALL CONCERT

2015 Josiah: **Invisible Cities (iii): Ersilia**
for chamber orchestra

2015 Trevor: **Al-kitab al-khamr**
for eleven players

2015 Josiah: **Invisible Cities (ii): Armilla**
for viola duet

2014 Trevor: **Krummzeit**
for seven players

Scores and source code are all available on GitHub.

INTEGRATION

LaTeX

Preprocessing LaTeX input files

Sphinx

Extensions for executing Python inline and embedding graphics

IPython

Embedding graphics and audio in IPython notebooks

Graphviz

Object-oriented toolkit for constructing Graphviz graphs

How to embed graphics into a document?

Use a *sand-boxed* interpreter within your actual interpreter

Python's `code` module

Monkey-patch all output functions

Replace `print()`, `show()` and friends inside the sandbox

Capture objects to be rendered and save them for later

Use an intermediate format for multiple potential outputs

COMPOSITION

https://github.com/josiah-wolf-oberholtzer/armilla

CONCLUSION

The Abjad API for Formalized Score Control extends the Python programming language with an open-source, object-oriented model of common-practice music notation that enables composers to build scores through the aggregation of elemental notation objects.

Documentation

http://projectabjad.org

GitHub Repository

http://github.com/Abjad/abjad

User Mailing List

http://groups.google.com/group/abjad-user

ABJAD: AN OPEN-SOURCE SOFTWARE SYSTEM FOR FORMALIZED SCORE CONTROL

Trevor Bača
Harvard University
trevor.baca@gmail.com

Josiah Wolf Oberholtzer
Harvard University
josiah.oberholtzer@gmail.com

Jeffrey Treviño
Carleton College
jeffrey.trevino@gmail.com

Víctor Adán
vctradn@gmail.com

ABSTRACT

The Abjad API for Formalized Score Control extends the Python programming language with an open-source, object-oriented model of common-practice music notation that enables composers to build scores through the aggregation of elemental notation objects. A summary of widely used notation systems' intended uses motivates a discussion of system design priorities via examples of system use.

1. INTRODUCTION

Abjad [1] is an open-source software system designed to help composers build scores in an iterative and incremental way. Abjad is implemented in the Python [2] programming language as an object-oriented collection of packages, classes and functions. Composers can visualize their work as publication-quality notation at all stages of the compositional process using Abjad's interface to the LilyPond [3] music notation package. The first versions of Abjad were implemented in 1997 and the project website is now visited thousands of times each month. This paper details some of the most important principles guiding the development of Abjad and illustrates these with examples of the system in use. The priorities outlined here arise in answer to domain-specific questions of music modeling (What are the fundamental elements of music notation? Which elements of music notation should be modeled hierarchically?) as well as in consideration of the ways in which best practices taken from software engineering can apply to the development of a music software system (How can programming concepts like iteration, aggregation and encapsulation help composers as they work?). A background taxonomy motivates a discussion of design priorities via examples of system use.

[1] http://www.projectabjad.org
[2] http://www.python.org
[3] http://www.lilypond.org

2. A TAXONOMY

Many software systems implement models of music but few of these implement a model of notation. [4] Music software systems that model notation can be classified according to their generative tasks. [5]

Many notation systems — such as Finale, Sibelius, SCORE [33], Igor, Berlioz, Lilypond [34], GUIDO [35] NoteAbility [36], FOMUS [37, 38] and Nightingale — exist to help people engrave and format music documents; because these systems provide functions that operate on notational elements (i.e., transposition, spacing and playback), hidden models of common-practice music notation must underly all of these systems, and each system's interface constrains and directs the ways in which users interact with this underlying model of notation. These systems enable users to engrave and format music without exposing any particular underlying model of composition, and without requiring, or even inviting the user to computationally model composi-

[4] 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. Researchers and musical artists have modeled many such extrasymbolic musical entities, such as large-scale form and transition [1–5], texture [6], contrapuntal relationships [7–13], harmonic tension and resolution [14–16], melody [17, 18], meter [19], rhythm [20–22], timbre [23–25], temperament [26, 27] and ornamentation [28, 29]. 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 [30].

[5] Software production exists as an organizationally designed feedback loop between production values and implementation [31], and it is possible to understand a system by understanding the purpose for which it was initially designed. This purpose can be termed a software system's generative task. 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.

While composers working traditionally may allow intuition to substitute for formally defined principles, a computer demands the composer to think formally about music [32]. Keeping in mind generative task as an analytical framework, it is broadly useful to bifurcate a notation system's development into the modeling of 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 composition while focusing more ostensibly on a model of notation, or focusing on the abstract modeling of composition without a considered link to a model of notation. Generative task explains a given system's balance between computational models of composition and notation by assuming a link between intended use and system development.

tion. Such systems might go so far as to enable scripting, as in the case of Sibelius's ManuScript [39] scripting language or Lilypond's embedded Scheme code; although these systems enable the automation of notational elements, it remains difficult to model compositional processes and relationships.

Other systems provide environments specifically for the modeling of higher-level processes and relationships. OpenMusic [40], PWGL [41] and BACH [42] supply an interface to a model of common-practice notation, as well as a set of non-common-practice visual interfaces that enables the user to model composition, in the context of a stand-alone application and with the aid of the above notation editors for final engraving and layout via intermediate file formats. Similarly purposed systems extend text-based programming languages rather than existing as stand-alone applications, such as HMSL's extension of Forth [43], JMSL's extension of Java [44] and Common Music's extension of Lisp [45]. Other composition modeling systems, such as athenaCL [46] and PILE/AC Toolbox [47] provide a visual interface for the creation of compositional structures without providing a model of common-practice notation.

Some composers make scores with software systems that provide neither a model of notation nor a model of composition. Graphic layout programs, such as AutoCAD and Adobe Illustrator, have been designed broadly for the placement and design of graphic elements. While scripting enables automation, composers must model both notation and composition from scratch, and the symbolic scope of potential automations described in the course of modeling ensures that composers spend as much time addressing elemental typographical concerns (e.g., accidental collisions) as would be spent modeling compositional processes and relationships.

Many models of musical notation have been designed 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 [48]. Commercial notation software developers attempted to establish a data interchange standard for optical score recognition (NIFF) [49]. 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 [50]. [6]

3. ABJAD BASICS

Abjad is not a stand-alone application. Nor is Abjad a programming language. Abjad instead adds a computational model of music notation to the Python programming language. By designing Abjad as an extension to one of the most widely-used programming languages in the world, we hope to make a considerable collection of programming best practices available to composers in a straightforward way. Abjad is implemented as a Python package. [7] [8] [9] Composers work with Abjad exactly the same way developers work with all the other packages available for the language. In the most common case this means opening a file, writing code and saving the file:

```
from abjad import *

def make_nested_tuplet(
    tuplet_duration,
    outer_tuplet_proportions,
    inner_tuplet_subdivision_count,
    ):
    outer_tuplet = Tuplet.from_duration_and_ratio(
        tuplet_duration, outer_tuplet_proportions)
    inner_tuplet_proportions = \
        inner_tuplet_subdivision_count * [1]
    last_leaf = outer_tuplet.select_leaves()[-1]
    inspector = inspect_(last_leaf)
    right_logical_tie = inspector.get_logical_tie()
    right_logical_tie.to_tuplet(inner_tuplet_proportions)
    return outer_tuplet
```

The contents of the file can then be used in other Python files or in an interactive session:

```
>>> rhythmic_staff = Staff(context_name='RhythmicStaff')
>>> tuplet = make_nested_tuplet((7, 8), (3, -1, 2), 3)
>>> rhythmic_staff.append(tuplet)
>>> show(rhythmic_staff)
```

This paper demonstrates most examples in Python's interactive console because the console helps distinguish input from output; [10] however, composers work with Abjad primarily by typing notationally-enabled Python code into a collection of interrelated files and managing those files as a project grows to encompass the composition of an entire score. [11]

4. THE ABJAD OBJECT MODEL

Abjad models musical notation with *components*, *spanners* and *indicators*. Every notational element in Abjad belongs

[6] An attempt to survey more comprehensively the history of object-oriented notation systems for composition, in the context of the broader history of object-oriented programming, lies beyond the scope of this paper but has recently been undertaken elsewhere [51].

[7] See the Python Package Index for extensions to Python for purposes as diverse as creative writing and aeronautical engineering. The Python Package Index contains 54,306 packages at the time or writing and is available at *https://pypi.python.org*.

[8] Designing Abjad as an extension to Python makes hundreds of print and Web resources relevant to composers working in Abjad. The global community of developers working in Python also becomes available to composers, and vice versa.

[9] Abjad is an importable Python library. It can be used in whole or in part as a component of any Python-compatible system. For example, Abjad supports IPython Notebook, a Web-based interactive computational environment combining code execution, text, mathematics, plots and rich media into a single document. Notational output from Abjad can be transparently captured and embedded into an IPython Notebook that has loaded Abjad's IPython Notebook extension. See *http://ipython.org/notebook.html*.

[10] Lines preceded by the >>> prompt are passed to Python for interpretation; any output generated by the line of code appears immediately after. The example above creates a tuplet with the tuplet-making function defined earlier and calls Abjad's top-level `show()` function to generate a PDF of the result.

[11] The importance Abjad attaches to composers' ability to notate the objects they work with during composition should be underscored before proceeding to more detailed examples. Abjad reflects this priority in its illustration protocol, which conventionalizes the way developers equip Abjad classes with notation output and the way composers notate objects during composition. Abjad's top-level `show()` function, included in most of the examples in this article, implements the composer-facing part of this protocol and provides a unified way to notate any object in the system. A collection of `__illustrate__()` methods implement the developer-facing part of this protocol and define the way any given class should produce notation. Under the most common pattern, Abjad transforms an object into LilyPond input code and then calls LilyPond to render the input as a PDF.

Figure 11: First International Conference on Technologies for Music Notation and Representation, May 2015, Paris, France

Trevor Bača

- trevor.baca@gmail.com
- trevorbaca.com
- github.com/trevorbaca

Jeffrey Treviño

- jeffrey.trevino@gmail.com
- jeffreytrevino.com
- github.com/jefftrevino

Josiah Wolf Oberholtzer

- josiah.oberholtzer@gmail.com
- josiahwolfoberholtzer.com
- github.com/josiah-wolf-oberholtzer