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 [37], Igor, Berlioz, Lilypond [38], GUIDO [39] NoteAbility [40], FOMUS [41, 42] 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, 12–17], harmonic tension and resolution [18–20], melody [21, 22], meter [23], rhythm [24–26], timbre [27–29], temperament [30, 31] and ornamentation [32, 33]. 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 [34].

[5] Software production exists as an organizationally designed feedback loop between production values and implementation [35], 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 [36]. 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 [43] 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 [44], PWGL [45] and BACH [46] 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 standalone 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 [47], JMSL's extension of Java [48] and Common Music's extension of Lisp [49]. Other composition modeling systems, such as athenaCL [50] and PILE/AC Toolbox [51] 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 [52]. Commercial notation software developers attempted to establish a data interchange standard for optical score recognition (NIFF) [53]. 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 [54]. [6]

3. ABJAD BASICS

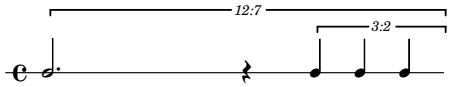
Abjad is not a stand-alone application. Nor is Abjad a programming language. Abjad instead adds a computational model of music notation to the Python programming language. By designing Abjad as an extension to one of the most widely-used programming languages in the world, we hope to make a considerable 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 [55].

[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.

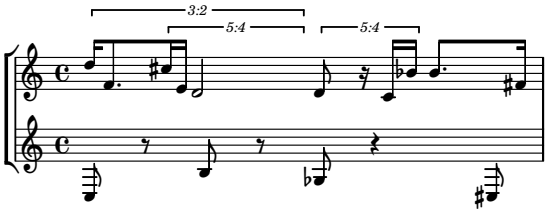
[11] The importance Abjad attaches to composers' ability to notate the objects they work with during composition should be underscored before proceeding to more detailed examples. Abjad reflects this priority in its illustration protocol, which conventionalizes the way developers equip Abjad classes 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.

to one of these three families. Abjad models notes, rests and chords as classes that can be added into the container-like elements of music notation, such as tuplets, measures, voices, staves and complete scores. [12] Spanners model notational constructs that cross different levels of hierarchy in the score tree, such as beams, slurs and glissandi. Indicators model objects attached to a single component, such as articulations, dynamics and time signatures. [13] Composers arrange components hierarchically into a score tree with spanners and indicators attached to components in the tree.

5. BOTTOM-UP CONSTRUCTION

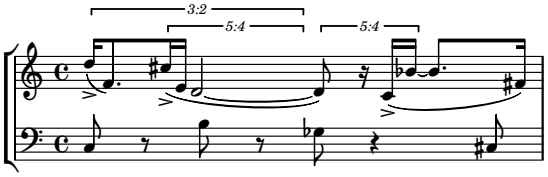
Abjad lets composers build scores from the bottom up. When working bottom-up, composers create individual notes, rests and chords to be grouped into tuplets, measures or voices that may then be included in even higher-level containers, such as staves and scores. [14] [15] Abjad implements this style of component aggregation via a collection of methods which append, extend or insert into container-like components as though they were lists: [16]

```
>>> outer_tuplet_one = Tuplet((2, 3), "d''16 f'8.")
>>> inner_tuplet = Tuplet((4, 5), "cs''16 e'16 d'2")
>>> outer_tuplet_one.append(inner_tuplet)
>>> outer_tuplet_two = Tuplet((4, 5))
>>> outer_tuplet_two.extend("d'8 r16 c'16 bf'16")
>>> tuplets = [outer_tuplet_one, outer_tuplet_two]
>>> upper_staff = Staff(tuplets, name='Upper Staff')
>>> note_one = Note(10, (3, 16))
>>> upper_staff.append(note_one)
>>> note_two = Note(NamedPitch("fs'"), Duration(1, 16))
>>> upper_staff.append(note_two)
>>> lower_staff = Staff(name='Lower Staff')
>>> lower_staff.extend("c8 r8 b8 r8 gf8 r4 cs8")
>>> staff_group = StaffGroup()
>>> staff_group.extend([upper_staff, lower_staff])
>>> score = Score([staff_group])
>>> show(score)
```



Ties, slurs and other spanners attach to score components via Abjad's top-level `attach()` function. The same is true for articulations, clefs and other indicators:

```
>>> upper_leaves = upper_staff.select_leaves()
>>> lower_leaves = lower_staff.select_leaves()
>>> attach(Tie(), upper_leaves[4:6])
>>> attach(Tie(), upper_leaves[-3:-1])
>>> attach(Slur(), upper_leaves[:2])
>>> attach(Slur(), upper_leaves[2:6])
>>> attach(Slur(), upper_leaves[7:])
>>> attach(Articulation('accent'), upper_leaves[0])
>>> attach(Articulation('accent'), upper_leaves[2])
>>> attach(Articulation('accent'), upper_leaves[7])
>>> attach(Clef('bass'), lower_leaves[0])
>>> show(score)
```



When does it make sense for composers to work with Abjad in the bottom-up way outlined here? Instantiating components by hand in the way shown above resembles notating by hand and composers may choose to work bottom-up when doing the equivalent of sketching in computer code: when making the first versions of a figure or gesture, when trying out combinations of small bits of notation or when inserting one or two items at a time into a larger structure. For some composers this may be a regular or even predominant way of working. Other composers may notice patterns in their own compositional process when they work bottom-up and may find ways to formalize these patterns into classes or functions that generalize their work; the next section describes some ways composers might do this.

6. TOP-DOWN CONSTRUCTION

What are the objects of music composition? For most composers, individual notes, rests and chords constitute only the necessary means to achieve some larger, musically interesting result. For this reason, a model of composition needs to describe groups of symbols on the page: notes taken in sequence to constitute a figure, gesture or melody; chords taken in sequence as a progression; attack points arranged in time as the scaffolding of some larger texture. These entities, and the others like them, might result from a flash of compositional intuition that then requires detailed attention and elaboration.

Abjad invites composers to implement factories as a way of generalizing and encapsulating parts of one's own compositional process. In this way, composers can extend the system as they work to implement their own models of composition. Abjad also provides a variety of factory functions and factory classes that exemplify this way of working. These range from simple note-generating functions, like `make_notes()`, which combine sequences of pitches and rhythms to generate patterned selections of notes and rests, to more complexly-configured maker classes for creating nuanced rhythmic patterns or entire scores.

As an example, consider the `rhythmmakertools` package included with Abjad. The classes provided in this package are factory classes which, once configured, can be called like functions to inscribe rhythms into a series of beats or other time divisions. The example below integrates configurable patterns of durations, tupletting and silences:

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     left_classes=(Rest, Note),
...     left_counts=(1,),
...     )
>>> talea = rhythmmakertools.Talea(
```

[12] Abjad uses the term *leaf* to refer generally to notes, rests, and chords. This term, borrowed from graph theory, reflects the fact that notes, rests and chords cannot contain other elements of music notation.

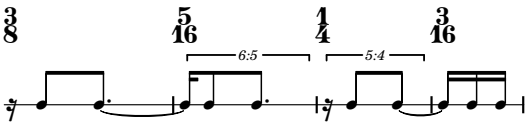
[13] Abjad indicators are scoped. Indicator scoping models how all notes in a voice can be marked *forte* until another note is marked *piano*. The scoping behavior of indicators is generalized to allow composers to annotate score structures by attaching arbitrary non-notating objects like dictionaries or custom classes to any component.

[14] Unlike many notation packages, Abjad does not require composers to structure music into measures. All of Abjad's containers can hold leaves notes, rests and chords directly.

[15] Notes, rests and chords may be initialized with pitches named according to either American or European conventions, with the pitch numbers of American pitch-class theory or from combinations of Abjad pitch and duration objects.

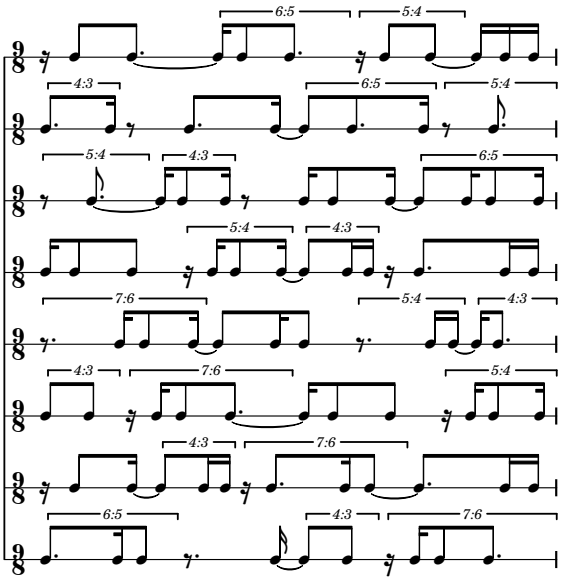
[16] Abjad's container interface derives from Python's mutable sequence protocol, which specifies an interface to list-like objects.

```
...     counts=(1, 2, 3),
...     denominator=16,
...     )
>>> tie_specifier = rhythmmakertools.TieSpecifier(
...     tie_across_divisions=True,
...     )
>>> rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
...     burnish_specifier=burnish_specifier,
...     extra_counts_per_division=(0, 1, 1),
...     talea=talea,
...     tie_specifier=tie_specifier,
...     )
>>> divisions = [(3, 8), (5, 16), (1, 4), (3, 16)]
>>> show(rhythm_maker, divisions=divisions)
```



Once instantiated, factory classes like this can be used over and over again with different input:

```
>>> rhythmic_score = Score()
>>> for i in range(8):
...     selections = rhythm_maker(divisions, seeds=i)
...     measure = Measure((9, 8), selections)
...     staff = Staff(context_name='RhythmicStaff')
...     staff.append(measure)
...     rhythmic_score.append(staff)
...     divisions = sequencetools.rotate_sequence(
...         divisions, 1)
...
>>> show(rhythmic_score)
```

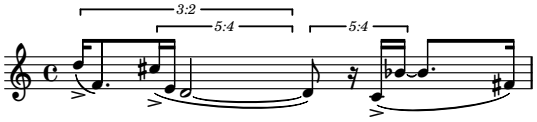


7. SELECTING OBJECTS IN THE SCORE

Abjad allows composers to select and operate on collections of objects in a score. Composers can select objects in several ways: by name, numeric indices or iteration. A single operation, such as transposing pitches or attaching articulations, can then be mapped onto the entirety of a selection.

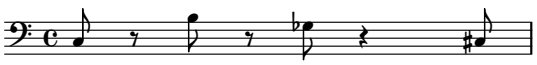
Consider the two-staff score created earlier. Because both staves were given explicit names, the upper staff can be selected by name:

```
>>> upper_staff = score['Upper Staff']
>>> show(upper_staff)
```



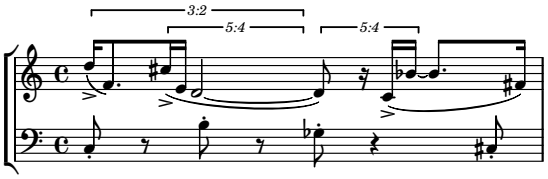
Using numeric indices, the lower staff can be selected by indexing the second child of the first child of the score:

```
>>> lower_staff = score[0][1]
>>> show(lower_staff)
```



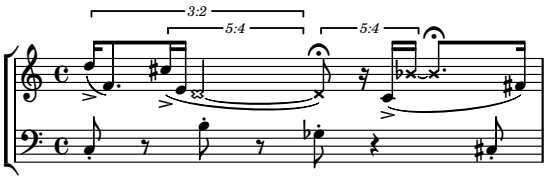
The top-level `iterate()` function exposes Abjad's score iteration interface. This interface provides a collection of methods for iterating the components in a score in different ways. For example, all notes can be selected from a single staff:

```
>>> for note in iterate(lower_staff).by_class(Note):
...     attach(Articulation('staccato'), note)
...
>>> show(score)
```



Groups of tied notes can be selected from an entire score: [17]

```
>>> for logical_tie in iterate(score).by_logical_tie():
...     if 1 < len(logical_tie):
...         attach(Fermata(), logical_tie.tail)
...         for note in logical_tie:
...             override(note).note_head.style = 'cross'
...
>>> show(score)
```



8. PROJECT TESTING AND MAINTENANCE

Abjad has benefited enormously from programming best practices developed by the open-source community. As described previously, the extension of an existing language informs the project as a first principle. The following other development practices from the open-source community have also positively impacted the project and might be helpful in the development of other music software systems.

The literature investigated in preparing this report remains overwhelmingly silent on questions of software testing. None of the sources cited in this article reference software test methodologies. The same appears to be true for the larger list of sources included in [55]. [18] Why should this be the case? One possibility is that authors of music software systems have, in fact, availed themselves of important

[17] Abjad uses the term *logical tie* to refer to a selection of one or more leaf components which represent all of the components tied together by zero or more ties; a single note, untied, is considered a *trivial* logical tie.

[18] Chris Ariza's work developing AthenaCL [50] and the development of Music21 [56] led by Michael Cuthbert are important exceptions. Both projects are implemented in Python and both projects feature approaches to testing aligned to those outlined above.

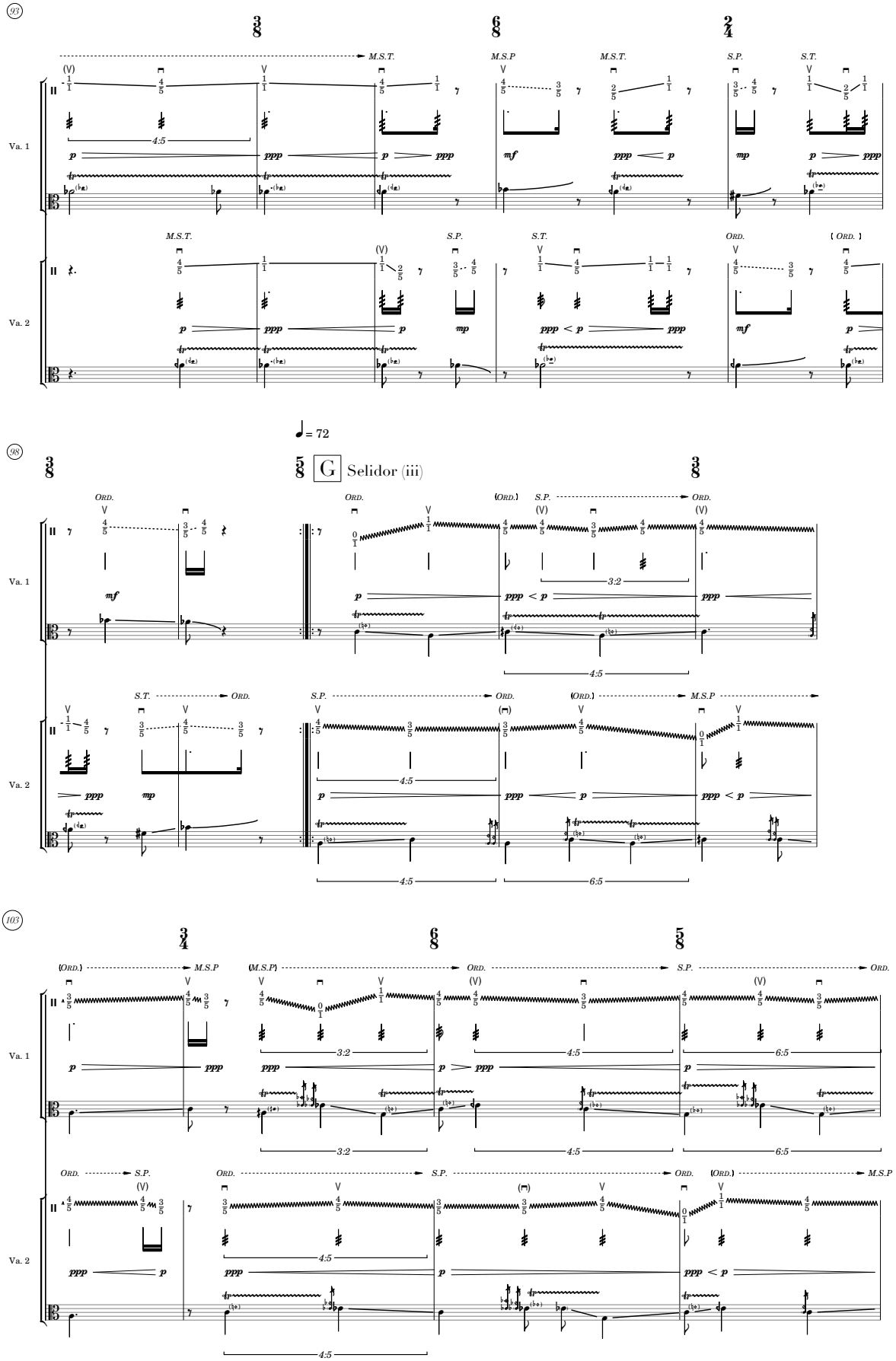


Figure 1. Page 8 from Josiah Wolf Oberholtzer's *Invisible Cities (ii): Armilla* for two violas (2015), created with tools extending Abjad. Source for this score is available at `https://github.com/josiah-wolf-oberholtzer/armilla`.

improvements in software test methods developed over the previous decades but have, for whatever reasons, remained quiet on the matter in the publication record. Perhaps the culture of software best practices now widely followed in the open-source community simply has not yet arrived in the field of music software systems development (and especially in the development of systems designed for non-commercial applications).

The use of automated regression testing in Abjad's development makes apparent the way in which tests encourage efficient development and robust project continuance. [19] The presence of automated regression tests acts as an incentive to new contributors to the system (who can test whether proposed changes to the system work correctly with existing features) and greatly increases the rate at which experienced developers can refactor the system during new feature development. Abjad currently comprises about 178,000 lines of code, and the Abjad repository, hosted on GitHub, [20] lists more than 8.7 million lines of code committed since the start of the project. This refactor ratio of about 50:1 means that each line of code in the Abjad codebase has been rewritten dozens of times. This freedom to refactor is possible only because of the approach to automated regression testing Abjad has borrowed from the larger open-source community.

Testing benefits project continuance when the original developers of a music software system can no longer develop the system. Automated regression tests help make possible a changing of the guard from one set of developers to another. Automated tests serve as a type of functional specification of how a software system should behave after revision. While automated tests alone will not ensure the continued development of any software system, adherence to the testing practices of the open-source community constitutes the most effective hedge available to music software systems developers to fend against project abandonment in the medium and long term.

9. DISCUSSION & FUTURE WORK

The design and development priorities for Abjad outlined here derive from the fact that the developers of Abjad are all composers who use the system to make their own scores. Abjad is not implemented for the type of music information storage and retrieval functions that constitute an important part of musicology-oriented music software systems. Nor is Abjad designed for use in realtime contexts of performance or synthesis. Abjad is designed as a composers' toolkit for the formalized control of music notation and for modeling the musical ideas that composers use notation to explore and represent. Although Abjad embeds well in other music software systems, future work planned for Abjad itself does not prioritize file format conversion, audio synthesis, realtime applications or graphic user interface integration. Future work will instead extend Abjad for object-oriented control over parts of the document preparation process required of complex scores with many parts. Future work will also extend and reinforce the inventory of factory classes and factory functions introduced in this report. We hope this will encourage composers working with Abjad to transition from working with lower-level symbols of music notation to modeling higher-level ideas native to one's own language of composition.

[19] Abjad comprises an automated battery of 9,119 unit tests and 8,528 documentation tests. Unit tests are executed by `pytest`. Documentation tests are executed by the `doctest` module included in Python's standard library. Parameterized tests ensure that different classes implement similar behaviors in a consistent way. Developers run the entire battery of tests at the start of every development session. No new features are accepted as part of the Abjad codebase without tests authored to document changes to the system. Continuous integration testing is handled by Travis CI to ensure that all tests pass after every commit from every core developer and newcomer to the project alike. For `pytest` see *http://pytest.org*. For Travis CI see *https://travis-ci.org*.

[20] https://github.com/Abjad/abjad

10. ACKNOWLEDGEMENTS

Our sincere thanks go out to all of Abjad's users and developers for their comments and contributions to the code. We would also like to thank everyone behind the LilyPond and Python projects, as well as the wider open-source community, for fostering the tools that make Abjad possible.

11. REFERENCES

[1] L. Polansky, M. McKinney, and B. E.-A. M. Studio, "Morphological Mutation Functions," in *Proceedings of the International Computer Music Conference*, 1991, pp. 234–41.

[2] Y. Uno and R. Huebscher, "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*, 1994, p. 7.

[3] C. Dobrian, "Algorithmic Generation of Temporal Forms: Hierarchical Organization of Stasis and Transition," in *Proceedings of the International Computer Music Conference*, 1995.

[4] S. Abrams, D. V. Oppenheim, D. Pazel, J. Wright *et al.*, "Higher-level Composition Control in Music Sketcher: Modifiers and Smart Harmony," in *Proceedings of the International Computer Music Conference*, 1999.

[5] M.-J. Yoo and I.-K. Lee, "Musical Tension Curves and its Applications," *Proceedings of International Computer Music Conference*, 2006.

[6] S. Horenstein, "Understanding Supersaturation : A Musical Phenomenon Affecting Perceived Time," *Proceedings of International Computer Music Conference*, 2004.

[7] G. Boenn, M. Brain, M. De Vos, and et. al., "Anton: Composing Logic and Logic Composing," in *Logic Programming and Nonmonotonic Reasoning*. Springer, 2009, pp. 542–547.

[8] A. Acevedo, "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, 2005, p. 96. [Online]. Available: http://books.google.com/books?hl=en&lr=&id=zXegsi7tj00C&oi=fnd&pg=PA96&dq=Fugue+Composition+with+Counterpoint+Melody+Generation+Using+Genetic+Algorithms&ots=Z4DpBjPMN-&sig=ocgSVVHiDIB7GJfmznh1Z3OheUA

[9] T. Anders and E. R. Miranda, "Constraint Programming Systems for Modeling Music Theories and Composition," *ACM Computing Surveys*, vol. 43, no. 4, pp. 30:1–30:38, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978809

[10] K. Balser and B. Streisberg, "Counterpoint Compositions in Non-tempered Systems: Theory and Algorithms," *Proceedings of International Computer Music Conference*, 1990.

[11] D. E. Jones, "A Computational Composer's Assistant for Atonal Counterpoint," *Computer Music Journal*, vol. 24, no. 4, pp. pp. 33–43, 2000. [Online]. Available: http://www.jstor.org/stable/3681553

[12] M. E. Bell, "A MAX Counterpoint Generator for Simulating Stylistic Traits of Stravinsky, Bartok, and Other Composers," *Proceedings of International Computer Music Conference*, 1995.

[13] M. Farbood and B. Schoner, "Analysis and Synthesis of Palestrina-style Counterpoint using Markov Chains," in *Proceedings of the International Computer Music Conference*, 2001, pp. 471–474.

[14] D. Cope, "Computer Analysis and Computation Using Atonal Voice-Leading Techniques," *Perspectives of New Music*, vol. 40, no. 1, pp. 121–146, 2002. [Online]. Available: http://www.jstor.org/stable/833550

[15] M. Laurson and M. Kuuskankare, "Extensible Constraint Syntax Through Score Accessors," in *Journées d'Informatique Musicale*, 2005, pp. 27–32.

[16] L. Polansky, A. Barnett, and M. Winter, "A Few More Words About James Tenney: Dissonant Counterpoint and Statistical Feedback," *Journal of Mathematics and Music*, vol. 5, no. 2, pp. 63–82, 2011.

[17] K. Ebcioglu, "Computer Counterpoint," *Proceedings of International Computer Music Conference*, 1980.

[18] A. F. Melo and G. Wiggins, "A Connectionist Approach to Driving Chord Progressions Using Tension," in *Proceedings of the AISB*, vol. 3, no. 1988, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9086&rep=rep1&type=pdf

[19] G. Wiggins, "Automated Generation of Musical Harmony: What's Missing?" in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1999. [Online]. Available: http://www.doc.gold.ac.uk/~mas02gw/papers/IJCAI99b.pdf

[20] C. D. Foster, "A Consonance Dissonance Algorithm for Intervals," *Proceedings of International Computer Music Conference*, 1995.

[21] D. Hornel, "SYSTHEMA - Analysis and Automatic Synthesis of Classical Themes," *Proceedings of International Computer Music Conference*, 1993.

[22] M. Smith and S. Holland, "An AI Tool for Analysis and Generation of Melodies," *Proceedings of International Computer Music Conference*, 1992.

[23] M. Hamanaka, K. Hirata, and S. Tojo, "Automatic Generation of Metrical Structure Based on GTTM," *Proceedings of International Computer Music Conference*, 2005.

[24] P. Nauert, "Division- and Addition-based Models of Rhythm in a Computer-Assisted Composition System," *Computer Music Journal*, vol. 31, no. 4, pp. 59–70, Dec. 2007. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/comj.2007.31.4.59

[25] B. Degazio, "A Computer-based Editor for Lerdahl and Jackendoff's Rhythmic Structures," *Proceedings of International Computer Music Conference*, 1996.

[26] N. Collins, "A Microtonal Tempo Canon Generator After Nancarrow and Jaffe," in *Proceedings of the International Computer Music Conference*, 2003.

[27] I. Xenakis, "More Thorough Stochastic Music," *Proceedings of International Computer Music Conference*, 1991.

[28] D. P. Creasey, D. M. Howard, and A. M. Tyrrell, "The Timbral Object - An Alternative Route to the Control of Timbre Space," *Proceedings of International Computer Music Conference*, 1996.

[29] N. Osaka, "Toward Construction of a Timbre Theory for Music Composition Composition," *Proceedings of International Computer Music Conference*, 2004.

[30] J. C. Seymour, "Computer-assisted Composition in Equal Tunings: Tonal Congnition and the *Thirteen Tone March*," *Proceedings of International Computer Music Conference*, 2007.

[31] A. Gräf, "On Musical Scale Rationalization," *Proceedings of International Computer Music Conference*, 2006.

[32] C. Ariza, "Ornament as Data Structure : An Algorithmic Model Based on Micro-Rhythms of Csángó Laments and Funeral Music Music of the Csángó," *Proceedings of International Computer Music Conference*, 2003.

[33] W. Chico-Töpfer, "AVA: An Experimental, Grammar/Case-based Composition System to Variate Music Automatically Through the Generation of Scheme Series," *Proceedings of International Computer Music Conference*, 1998.

[34] N. Collins, "Musical Form and Algorithmic Composition," *Contemporary Music Review*, vol. 28, no. 1, pp. 103–114, Feb. 2009.

[35] J.-C. Derniame, B. A. Kaba, and D. Wastell, *Software Process: Principles, Methodology, and Technology*. Springer, 1999.

[36] I. Xenakis, *Formalized Music: Thought and Mathematics in Composition*. Pendragon Press, 1992.

[37] L. Smith, "SCORE- A Musician's Approach to Computer Music," *Journal of the Audio Engineering Society*, vol. 20, no. 1, pp. 7–14, 1972.

[38] H.-W. Nienhuys and J. Nieuwenhuizen, "LilyPond, A System for Automated Music Engraving," in *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*. Citeseer, 2003, pp. 167–172.

[39] H. H. Hoos, K. Hamel, K. Renz, and J. Kilian, "The GUIDO Notation Format- A Novel Approach for Adequatly Representing Score-level Music," *Proceedings of International Computer Music Conference*, 1998.

[40] K. Hamel, "NoteAbility Reference Manual," 1997.

[41] D. Psenicka, "FOMUS , a Music Notation Software Package for Computer Music Composers," *Proceedings of the International Computer Music Conference*, pp. 75–78, 2006.

[42] ——, "FOMUS , a Music Notation Software Package for Computer Music Composers," *Proceedings of the International Computer Music Conference*, pp. 69–72, 2009.

[43] AVID. Plugins for Sibelius. [Online]. Available: http://www.sibelius.com/download/plugins/index.html?help=write

[44] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue, "Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic," *Computer Music Journal*, vol. 23, no. 3, pp. pp. 59–72, 1999. [Online]. Available: http://www.jstor.org/stable/3681240

[45] M. Laurson, M. Kuuskankare, and V. Norilo, "An Overview of PWGL, a Visual Programming Environment for Music," *Computer Music Journal*, vol. 33, no. 1, pp. 19–31, 2009.

[46] A. Agostini and D. Ghisi, "Real-time Computer-aided Composition with BACH," *Contemporary Music Review*, vol. 32, no. 1, pp. 41–48, 2013.

[47] L. Polansky, "HMSL (Hierarchical Music Specification Language): A Theoretical Overview," *Perspectives of New Music*, vol. 28, no. 2, 1990.

[48] N. Didkovsky and P. Burk, "Java Music Specification Language, an introduction and overview," in *Proceedings of the International Computer Music Conference*, 2001, pp. 123–126.

[49] H. Taube, "Common music: A music composition language in common lisp and clos," *Computer Music Journal*, pp. 21–32, 1991.

[50] C. Ariza, "An Open Design for Computer-aided Algorithmic Composition: athenaCL," Ph.D. dissertation, New York University, 2005. [Online]. Available: 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

[51] P. Berg, "PILE - A Language for Sound Synthesis," *Computer Music Journal*, vol. 3, no. 1, pp. 30–41, 1979. [Online]. Available: http://www.jstor.org/stable/3679754

[52] E. Selfridge-Field, *Beyond MIDI: The Handbook of Musical Codes*. The MIT Press, 1997.

[53] N. Consortium and et al., "NIFF 6a: Notation Interchange File Format," NIFF Consortium, Tech. Rep., 1995.

[54] M. Good, "MusicXML for Notation and Analysis," in *The Virtual Score: Representation, Retrieval, Restoration*, ser. Computing in Musicology, W. B. Hewlett and E. Selfridge-Field, Eds. MIT Press, 2001, no. 12, pp. 113–124.

[55] J. R. Trevino, "Compositional and Analytic Applications of Automated Music Notation via Object-oriented Programming," Ph.D. dissertation, University of California, San Diego, 2013.

[56] C. Ariza and M. Cuthbert, "Modeling Beats, Accents, Beams, and Time Signatures Hierarchically with Music21 Meter Objects," in *Proceedings of the International Computer Music Conference*, 2010. [Online]. Available: http://web.mit.edu/music21/papers/2010MeterObjects.pdf