http://cnmat.berkeley.edu/publications/new-tools-aspect-oriented-programming-music-and-media-programming-environments

New Tools for Aspect-Oriented Programming in Music and Media Programming Environments

John MacCallum, Adrian Freed, David Wessel
Center for New Music and Audio Technologies
Department of Music
University of California, Berkeley
{john,adrian,wessel}@cnmat.berkeley.edu

ABSTRACT

Media/arts programming is often experimental and exploratory in nature and requiring a flexible development environment to enable continually changing requirements and to facilitate iterative design in which the development of software impacts the design of a work of art, which in turn produces new requirements for the software. We discuss *agile development* as it relates to media/arts programming. We present *aspect-oriented programming* and its implementation in Max/MSP using Open Sound Control and the odot library as tool for mobilizing the benefits of agile development.

1. INTRODUCTION

Media/arts programming is often speculative in nature and its practice is closely related to that of *agile development* [1] in the software engineering community. The following principles constitute (with some slight modifications for media/arts development) agile programming [2, 3]:

1. The person for whom the development is being done (often oneself) should be satisfied through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes embrace change as the artist adapts his/her vision of the project based on iterations of the software.
3. Deliver working software frequently.
4. Artists and developers must work together often throughout the project.
5. Working systems are the primary measure of progress.
6. Agile processes promote sustainable development. The artists and developers should be able to maintain a constant pace indefinitely.
7. Continuous attention to technical excellence and good design enhances agility.
8. Simplicity—the art of maximizing the amount of work done—is essential.

Copyright: ©2014 John MacCallum et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Of these principles, 2, 3, 6, and 7 are perhaps the most important for our purposes. Since the creation of a work of art is a speculative process, a clear and well-defined specification for a piece of software is rarely possible. Iterative development where the software is tested and evaluated in the context of the piece being created is essential to allow the software and the concept of the work of art to co-evolve. To promote this type of exchange, the development environment must be one that is flexible, not brittle, and one that welcomes potentially drastic change as the result of incremental use and evaluation. Aspect-oriented programming can limber up the development environment when used sparingly and judiciously.

1.1 Aspect-Oriented Programming

Before providing a description of aspect-oriented programming, it is useful to introduce some terms that are used throughout the literature [4].

1.1.1 Terminology

Cross-cutting concerns Aspects of a program that cut across or are interwoven among many different parts of a program (i.e., logging, debugging, etc.).

Advice Additional behavior applied to data in the context of an aspect.

Join point A point in the control flow of a program—in dataflow languages like Max/MSP, PD, or Ptolemy II [5] [1] these points are the inlets and outlets of data flow actors.

Pointcut A set of join points that may have *advice* associated with them.

1.1.2 Description

"Aspect-oriented programming (AOP) is a programming methodology [which separates out] cross-cutting concerns [. . .] from the main code of the actions to which the concerns apply."[6] Some examples of cross-cutting concerns that are useful in the context of real-time media/arts programming are:

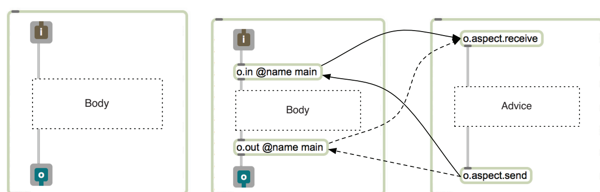
- Logging
- Visualization
- Structural analysis
- Commentary
- Scaffolding

[1] http://ptolemy.eecs.berkeley.edu

- Stream capture
- Performance profiling
- Debugging, printing, and tracing
- Input/Output validation and assertion
- Dynamic code injection without modifying existing (possibly running) code

Most modules in a complex program will need some of the features itemized above at some point in the development process. AOP enables those elements of the program to be injected when necessary and easily removed when they are no longer needed without modifying the code to which they are applied. AOP frees programmers from the need to foresee functionality and often obviates the need for programmers to revisit code to extend or modify its behavior. By providing each module with the proper hooks, we can quickly and unobtrusively add additional behavior ("advice") at various points in the program ("join points") without modifying existing code. Further, AOP obviates the need to remember many ad hoc systems for mundane functionality such as printing to a debugging console.

2. SIMPLE EXAMPLE



(a) Max abstraction (b) Max abstraction with hooks

Figure 1: On the left, we see the typical dataflow through an abstraction in Max. On the right, the use of o.in and o.out to forward incoming and outgoing data to aspects.

In figure 1a, we see the skeleton of a module in Max, while in figure 1b o.in and o.out provide join points where advice can be applied. Data enters a module and is forwarded by o.in to o.aspect.receive where advice is applied. After processing, it is sent back *to where it came from*, in this case o.in, by o.aspect.receive. o.in then forwards the (possibly modified) data to the body of the Max module. o.out behaves identically to o.in with the exception of its contextual information.

3. APPLICATIONS

As programs grow in complexity, the need to understand what is going on inside of submodules nested deep in the program hierarchy can present serious difficulties. For example, we may suspect that something is going wrong in a module and wish to see the values that are being sent into it as a way to determine whether the problem occurs inside or outside the module. This requires one of two things: *a*) modification to the existing module which, in the case of Max and PD, carries with it the loss of any stored internal state and the potential of introducing new bugs or behavior that can impede the search for the bug, or *b*) foresight during the initial implementation of the module resulting in debugging components that will aid in our current situation.The former is cumbersome, and the latter, when generalized to all types of debugging situations, requires the kind of forward thinking design that is particularly difficult in dynamic and speculative work.

One of the tenets of AOP is that the programmer should be *oblivious* to future aspects that may be applied at a later date, obviating the need for the programmer in our example to predict future debugging situations. One must still ensure that each inlet and outlet of the module is connected to an o.in or o.out, but assuming those hooks are in place, we may inspect the input to a module with the following steps (see figure 2).

1. Create a new patch and instantiate o.aspect.receive.
2. Filter the stream of bundles based on the port type (and possibly other contextual information).
3. Display the data in an appropriate way if it matches the contextual criteria.
4. When the visualization is no longer necessary, save the aspect in case it may be useful in the future, and simply close the window. If it becomes useful again, reopen it.

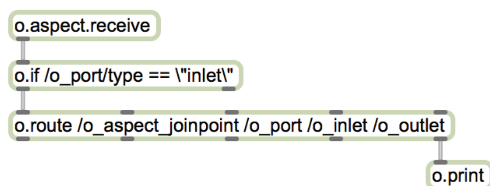


Figure 2: o.aspect.print

4. MAX/MSP IMPLEMENTATION

The implementation of AOP in Max that we introduce here makes use of Open Sound Control [7] as a rich, composite data type, and the odot library [8] for providing contextual information and high-level processing of OSC data. AOP is implemented in Max using a pair of Max "abstractions" or *shims* called o.in and o.out which are placed immediately after and before each inlet and outlet in a module, respectively. o.in and o.out are thin wrappers around o.port which collects contextual information from its environment such as the name of the module it is in and the name of the enclosing module, as well as the arguments of those modules. This data is added to the OSC bundle along with a "return address" and sent, using Max's built-in "send" object, to the global location "o.aspect".

We provide two additional abstractions that aid in writing aspects: o.aspect.receive which simply wraps the built-in "receive" object with the argument "o.aspect", and o.aspect.send which uses the return address to set the "forward" object to send to that location (see figure 3).

To create an aspect in Max/MSP, one creates a new patch and instantiates o.aspect.receive and o.aspect.send (both with no arguments). o.aspect.receive will produce all messages sent to the location "o.aspect". The bundle can then be processed according to the advice that this particular aspect provides. Certain aspects may choose to

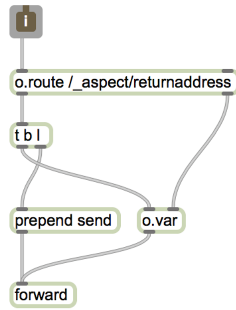


Figure 3: o.aspect.send

inject data into the bundle (e.g., profiling data) in which case the aspect should send the modified bundle back using o.aspect.send. If no modifications are made, this step may be skipped and a copy of the original bundle will be produced after the join point.

The following sections describe the implementation of o.port and o.aspect.joinpoint.

4.1 o.port

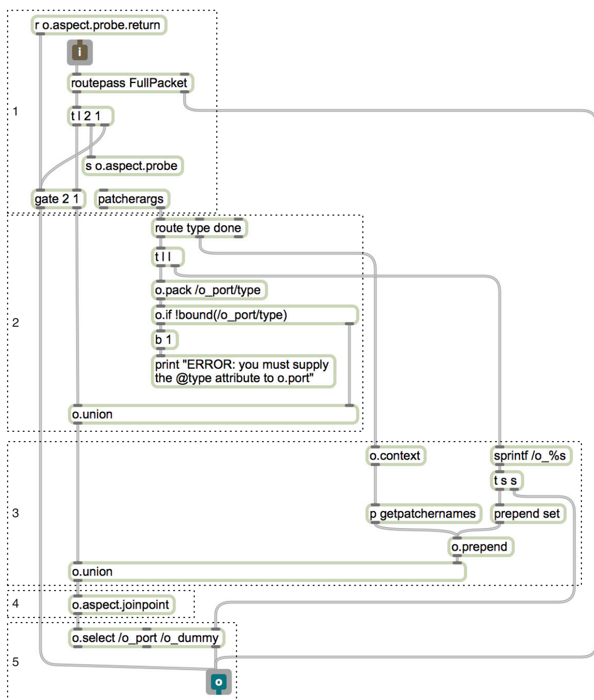


Figure 4: o.aspect.port

o.port (see figure 4) is responsible for gathering contextual information about its environment, blending it into the bundle, passing it to o.aspect.joinpoint, and outputting the result. o.port is a general mechanism that makes use of o.aspect.joinpoint and also serves as the location for implementing other hooks to extend the functionality of a patch.

The following enumerated items correspond to those in figure 4.

1. If the incoming data is not an OSC bundle, it is forwarded on to the enclosing patch. Otherwise, it is processed by o.port. For efficiency, we query the environment to see if any aspects have been instantiated and only proceed if so.
2. The port type, "in" or "out", is bound to /o_port/type and blended into the incoming OSC stream.
3. The context information for this port is blended into the incoming OSC stream.
4. The bundle is passed to the join point.
5. All contextual information is stripped off before outputting.

4.2 o.aspect.joinpoint

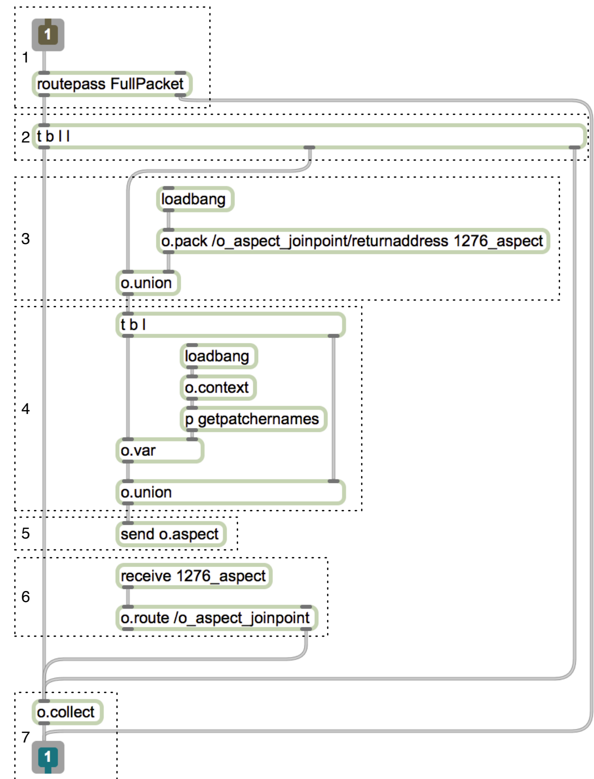


Figure 5: o.aspect.joinpoint

o.aspect.joinpoint (see figure 5) is responsible for dispatching incoming OSC bundles to any pointcuts that may be instantiated. If no pointcuts are in place, a copy of the bundle is simply passed through unchanged.

The following enumerated items correspond to those in figure 5.

1. OSC bundles are processed and all other non-OSC data is simply passed through untouched.
2. A copy of the incoming bundle is stored in o.collect. This will be sent out if no data is returned, or a union operation will be performed producing a bundle containing the original data and any data added by any aspects. After the bundle is passed to o.collect, it is processed by the following steps.
3. A "return address" is created using the unique numerical identifier created using the "#0" lexical substitution variable and blended into the OSC bundle.
4. Contextual information is generated using o.context and blended into the OSC bundle.

5. The bundle is sent to the named global location "o.aspect".
6. If a bundle is received here, the "return address" is removed and it is sent to o.collect which will combine it with the original.
7. The bundle is output into the enclosing patch.

5. A MORE DETAILED EXAMPLE

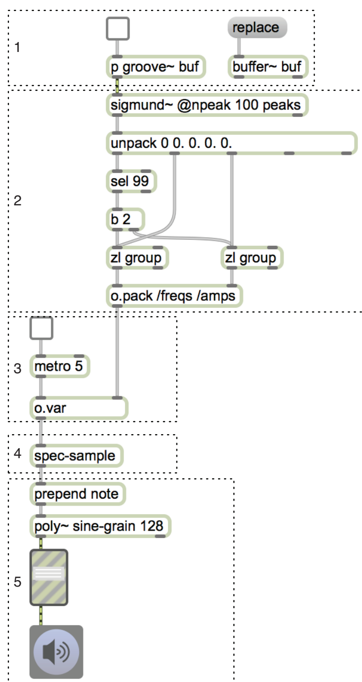


Figure 6: A granular synthesizer that chooses the frequency of each grain from a spectrum produced by sigmund∼ using the amplitudes of each frequency component as a probability distribution.

In this section, we use the implementation of a granular synthesizer to discuss the use of aspects in the development process. The granular synthesizer (figure 6) consists of two parts: code that chooses the frequency of a grain, and the code to sonify the grain. In this example, we take the spectral output of Miller Puckette's sigmund∼ external [2] and choose frequencies at random from it using the amplitudes as probabilities. This ensures that more grains will be set to those components that had a greater amplitude in the spectrum. The sonification can be done with any suitable polyphonic synthesizer—in our case, we use a simple enveloped sine wave. The following is a description of each component of figure 6.

1. Encapsulated logic for playback of sound files with groove∼.
2. Capture output from sigmund∼ and encode as OSC.
3. Drive the granular synthesizer with a clock independent of the rate of output of sigmund∼.
4. Choose a random frequency from the spectrum using the amplitudes as a (categorical) probability distribution.
5. Sonify a grain at the chosen frequency.

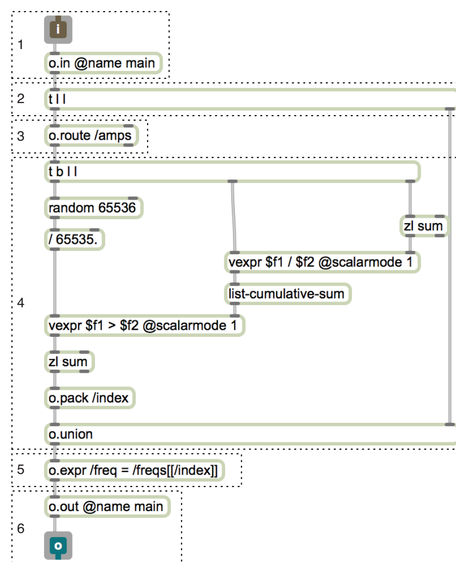


Figure 7: Choose a frequency at random using a list of amplitudes as probabilities.

The module called spec-sample draws a random sample from the spectrum and is implemented as follows (see figure 7).

1. Inlet with o.in.
2. Store a copy of the bundle in order to blend derived data into it.
3. Extract the list of amplitudes.
4. Treat the list of amplitudes as a probability mass function, convert it to a cumulative distribution function, and draw a random sample from it.
5. Assign the frequency corresponding to the random index to the address /freq.
6. Output the bundle, passing through o.out first.

5.1 Visualization

While developing this granular synthesizer, we may want to visualize its output. Normally, we would construct something in the main patch, however, such ad hoc work is often discarded when not needed in order to clean the patch up and optimize it for efficiency. AOP can assist here as seen in figure 8. We instantiate o.aspect.receive and filter any incoming OSC data by looking specifically for bundles that come from outlets and are sent from patches called "spec-vis". We then interleave the frequencies and amplitudes for display with resdisplay [3] and visualize the data bound to /freq using multislider.

5.2 Extension and Experimentation

We may wish to add additional behavior to our program, for example, spectral smearing which we could implement by adding a random value to the chosen frequency. We may want to experiment with different families of probability distributions, and ultimately we may wish to discard

[2] http://crca-archive.ucsd.edu/ tapel/software.html

[3] http://cnmat.berkeley.edu/downloads

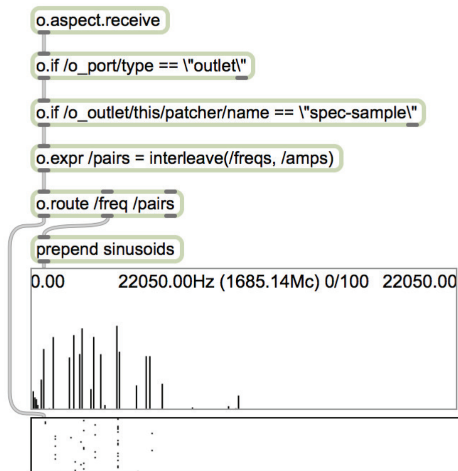


Figure 8: An aspect used to visualize the data computed in spec-sample.

this behavior if it proves to be uninteresting. Rather than perform many edits on a working patch, we can encapsulate this speculative work in an aspect as seen in figure 9.

We first look for OSC bundles that came from the outlet of the spec-sample patch. We then blend in the name of a probability distribution and its parameters which is used to generate a random value that is added to the value of /freq.

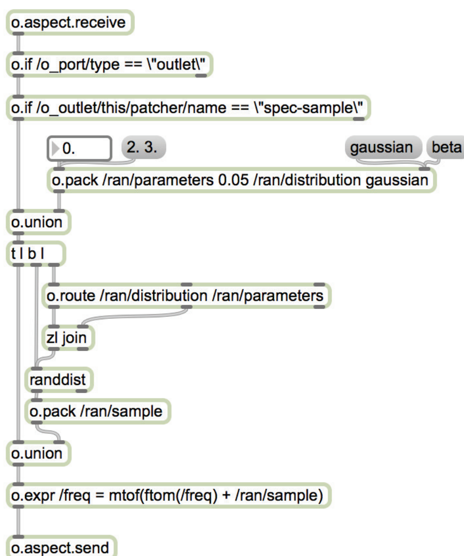


Figure 9: An aspect used to experiment with different probability distributions for smearing the spectrum.

6. CONCLUSIONS AND FUTURE WORK

We have presented an implementation of aspect-oriented programming for data-flow languages such as Max/MSP and PD, which can simplify a variety of tasks in arts/media programming of an agile and speculative nature. We illustrated this implementation with a case study of granular synthesis development.

Aspect-oriented programming is a relatively recent paradigm with a growing community of users exploring where it can be effective. Our contribution in this work is to bring the paradigm to the music and intermedia communities. As well as having noticeable positive effects on our own productivity we have discovered that this style of programming requires extensions to the core programming environments especially in the area of introspection. We have also found interesting opportunities to extend aspect-oriented programming afforded by visual programming environments. For example, it is effective to write aspects that change the colors of Max/MSP object boxes, text and patch chords to contextualize state changes and program flow.

Acknowledgments

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

7. REFERENCES

[1] A. Cockburn, *Agile Software Development: The Cooperative Game*, 2nd ed. Addison-Wesley Professional, 2006.

[2] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds., *Aspect-Oriented Software Development*. Boston: Addison-Wesley, 2005.

[3] "Principles behind the agile manifesto," http://agilemanifesto.org/principles.html, accessed: 2014-03-29.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*. SpringerVerlag, 1997.

[5] I. Akkaya, P. Derler, and E. Lee, "Aspect-oriented fault modelling and anomaly detection in Ptolemy II."

[6] D. Patterson and A. Fox, *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Strawberry Canyon LLC. Kindle Edition, 2014.

[7] M. Wright and A. Freed, "Open sound control: A new protocol for communicating with sound synthesizers," in *Proceedings of the International Computer Music Conference, (Thessaloniki, Hellas)*, 1997, pp. 101–104.

[8] A. Freed, J. MacCallum, and A. Schmeder, "A dynamic, instance-based, object-oriented programming in max/msp using open sound control message delegation," in *Proceedings of the International Computer Music Conference*, 2011.