Technical Report No: 2004/02

Using Dynamic Aspects in Music Composition Systems

Patrick Hill
Simon Holland
Robin C. Laney

9th February 2004

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

The Open University

Using Dynamic Aspects in Music Composition Systems

Patrick Hill
The Open University
Walton Hall
Milton Keynes. MK7 6AA
PatrickHill@bcs.org.uk

Simon Holland
The Open University
Walton Hall
Milton Keynes. MK7 6AA
s.holland@open.ac.uk

Robin C. Laney
The Open University
Walton Hall
Milton Keynes. MK7 6AA
r.c.laney@open.ac.uk

ABSTRACT
Aspect-oriented programming (AOP) attempts to modularise crosscutting concerns in software. Initial approaches to AOP have used static weaving techniques in which crosscutting implementation, encapsulated by aspects, is merged into . Research into dynamic aspects suggests various ways in which crosscutting implementations may be dynamically woven into code, enabling aspects to be defined and composed at run-time.

It has been suggested, in [14], that AOP might be usefully applied at the end-user level in applications that support multidimensional creative processes, and in particular, of music composition. In this paper we extend this argument to suggest that dynamic aspects are essential to this application. We motivate our argument with a high-level description of crosscutting that exists within music composition, and ways in which these crosscutting concerns, and requirements for their management, have arisen from our initial use of static aspects in music composition. We then evaluate some of the ways in which current research into dynamic aspects might be utilised in addressing these requirements.

Categories and Subject Descriptors
D.2.2 **[Software Engineering]**: Design Tools and Techniques - *Object-oriented design methods, Aspect-oriented design*; D.2.1 **[Software Engineering]:** Requirements / Specifications **-** *Methodologies, Separation of Concerns*; D.2.3 **[Software Engineering]:** Coding Tools and Techniques; D.3.3 **[Programming Languages]:** Language Constructs and Features - *Aspects, Dynamic Aspects;* J.5 **[Computer Applications]:** Performing Arts - *Music*

General Terms
Design, Languages, Human Factors.

Keywords
Aspect-oriented programming, Dynamic Aspects, music composition.

1. INTRODUCTION
Aspect-oriented programming (AOP) is a technique that aims to assist software developers in the separation and composition of various dimensions of concern across a range of software engineering tasks. Early compositional AOP tools, such as AspectJ, operate by statically composing, or *weaving,* crosscutting concerns, expressed as *aspects*, with basic concern code, expressed as *classes*. However, static weaving, by definition, assumes that aspectual relationships are largely invariant and that they can be determined at design-time and are therefore tied to a particular class-graph [19].

In contrast, dynamic aspects variously offer the potential to defer binding of particular aspect implementations until run time. Unlike the statically woven aspects of AspectJ, dynamic aspects persist at run-time and it therefore becomes possible to dynamically modify aspectual relationships, enabling aspects to be added, withdrawn, or replaced depending upon dynamic context.

In [14] we argue that music composition can be viewed in terms of composition of various dimensions of musical concern and that analogies exist with AOP. We suggest that AOP may be used at the *end-user* level in systems that support music composition. In this paper we further suggest that aspectual relationships in musical composition are largely dynamic, and that dynamic aspects could prove a useful technology in the development of aspect-oriented music composition tools.

2. SEPARATION AND COMPOSITION OF CONCERNS IN MUSIC
Music composition is a creative process in which the separation and composition of dimensions of concern are important and pervasive problems. Multidimensional tangling and scattering exists not only within the structure, representation and manipulation of musical data, but also in the cognitive processes of composition [14].

It is common experience that music is not merely a random stream of sound events. Rather, the composer typically works with a limited set of musical resources that are manipulated, in various ways, to form a logical and coherent whole [28]. The ‘musical surface’ of a musical composition, which is perceived by the listener in terms of *pitch*, *duration*, *loudness*

and *timbre*[1] [20], can be viewed as the result of the composer's weaving together of a 'tangled web' of musical structures and dimensions [8].

Although the processes of software engineering and music composition are clearly different, there are some parallels between the two. We can, for example, draw a broad analogy between the notation that is traditionally output as part of a composition process and the set of instructions executed by a computer as the result of a software engineering process. In both cases, the outputs are, largely, sets of low-level performance instructions resulting from the composition of high-level abstractions.

For example, consider the musical gesture of *crescendo*, ie. 'getting louder over time'. A crescendo might be readily achieved by simply increasing the value of the 'loudness dimension', eg. by striking piano keys with more force or blowing a trumpet more forcefully[2]. However, there are other dimensions that may also be modified in order to obtain a crescendo effect. For example, the composer might choose to

- introduce additional instruments (timbre),
- use different pitches (pitch),
- modify the arrangement of harmonies (pitch /timbre)

and so forth.

Thus the basic 'crescendo' concern may be scattered among other musical dimensions. Moreover, the particular crescendo implementation used might depend upon musical context. For example, while some sections of a musical piece might be written for a full orchestra, other sections might be written for strings only. Thus the 'introduction of additional instruments' approach to crescendo might itself be limited in the instruments that may be used.

We choose this example as one that is readily understood and that does not mandate in-depth discussion of musical technicalities. There are numerous other documented instances of crosscutting in music. Examples include separation of metre and melody [17], the impact of tempo on performance [10], and the interrelationship between orchestration and composition [24]

While software is typically composed through the use of automated tools; compilers, configuration management etc, even with computer assistance, the music composer is often forced to express high-level musical ideas in terms of tangled, low-level musical detail, by manually weaving together various dimensions. The requirement to express music in such a detailed 'note-list' representation rather than as higher-level constructs is incompatible with the creative process itself [21]. Moreover, musical composition does not appear to be a linear process. Rather, composers tend to sketch out and elaborate ideas iteratively and across multiple, possibly incomplete, dimensions [29][23][30]. During the composition process, certain musical elements may be created which the composer wishes to preserve and use, but not necessarily in the present composition [30]

[1] Timbre describes those qualities of a sound that enable the listener, for example, to distinguish between a trumpet and a violin.

[2] In practice, these techniques would also affect timbre.

From the viewpoint of the music analyst, just as it would be difficult to identify aspects from reverse-engineered bytecode produced by AspectJ, musical intent is often obscured in the musical score produced by the composer. As Raes [27] points out, musical scoring systems, conventional or otherwise, do not express the 'conception' or 'flow of ideas' within a musical composition. This is not to underestimate the power of algorithmic musical composition systems but note that here too, musical dimensions are often scattered or tangled.

We believe that AOP techniques might be usefully applied to the domain of computer assisted music composition as a way to weave together separately described musical elements that express musical *intent*. An AOP-based music creation environment could enable the composer to work in an iterative experimental fashion that supports the creative process.

3. WHY ARE DYNAMIC ASPECTS IMPORTANT TO MUSICAL COMPOSITION APPLICATIONS?

Our initial research has shown that static aspects, of the type implemented by AspectJ, may be used, with some success, to help separate musical concerns and compose them into a musical piece. For example, we have used AspectJ to construct the first few bars of Widor's famous organ Toccata, using a core program that represents a sequence of chords, and aspects that implement crosscutting concerns, such as changes of key, temporal position and duration of chords, and transformation of the chords into the left-hand and right-hand parts of the original score. In considering the extension of this system to generate the entire piece, it was observed that some general requirements could not be met by AspectJ.

- Music is often based on the variation and juxtaposition of a small number of musical elements [28]. From an AOP perspective, this means that aspectual relationships do not necessarily persist for the entire duration of a musical piece. This is in some ways analogous to the 'Jumping Aspects' problem [5], in that a particular aspect behaviour might be desirable only within certain musical contexts. Gybels [12] observes that some crosscut languages, particularly that used in AspectJ, are not Turing Complete, and are therefore unable to evaluate dynamic expressions. Clearly, 'enabling conditions' based on dynamic context might be encoded into the advice, but this could lead to over-complicated code and unclear separation of concerns between pointcut and advice.

  For example, consider the case where within the same piece of music some crescendi are realised by additional instrumentation, while others are realised by a simple increase of 'loudness'. A 'pointcut' on a crescendo would require different 'advice' depending on context.

  Dynamically installable aspects might be used to separate concerns such that the selection of 'which' crescendo implementation is used is described separately from the 'aspect' that invokes the

selected crescendo implementation at crescendo pointcuts.

- By definition, static aspects cannot be defined and applied interactively at run-time. Music composition is a creative art that involves experimentation and iteration [23][29][30]. The development of an interactive music composition system that enables the composer to selectively define, apply, refine, and withdraw aspectual relationships presupposes a dynamic aspect platform.

- Given such an interactive music composition system, it is possible that certain aspects could be constructed that may have application in a range of musical compositions, not only the one in which they were defined. In a similar way to the 'buy-don't-build' [7] methodology espoused by proponents of component based software development, the ability to encapsulate musical aspects as components that may be subsequently 'plugged-in' and reused in other musical composition projects is attractive.

- Greater separation of concerns might be achieved with the facility to compose aspects with other aspects. While this does not necessarily require dynamic aspects, it is not currently possible using, for example, AspectJ.

4. DYNAMIC ASPECT SYSTEMS

In this section we overview ways in which current dynamic aspect systems might help in the development of a musical composition system as outlined above.

4.1 Event-Based Dynamic AOP

Whereas systems such as AspectJ determine pointcuts from source-code inspection, event-based systems, such as EAOP [11], Axon [1] and PROSE [26] utilise various techniques to generate events at runtime. These events, which function as joinpoints, are intercepted and used to invoke separately defined crosscutting implementations.

EAOP performs source-code modification to produce a framework that instruments the source application code such that execution events are raised to an *event monitor*. Aspects, which are coded as pieces of Java code, are invoked by the event monitor upon receipt of particular events. An 'aspect tree', which specifies how events are routed to aspect code, is maintained by the monitor. Dynamic AOP is achieved by the ability to modify the aspect tree at run-time. Aspects may be composed with other aspects through the use of EAOP's composition operators.

In contrast, both Axon and PROSE utilise the services of the Java debugger interface (JVMDI) to raise events at appropriate points, such as method calls, in the programs execution. This approach does not require source-code modification. Axon uses an API to programmatically define aspects in terms of dynamic associations between pointcuts and advisory units. Advisory units, analogous to AspectJ's advice, are written as plain Java classes. In PROSE, Java classes representing aspects are defined as subclasses of the PROSE class Aspect. As such, each aspect class contains one or more Crosscut objects, each of which equates to the combination of an AspectJ pointcut **and** an advice. Pointcuts are defined using *specializers*, which are specified using an AspectJ-like pointcut syntax. Aspects may be added or removed through interaction with PROSE's ExtensionManager interface. An interesting feature of this approach is the ability to manipulate aspect instances from outside of the JVM in which the application is running.

Event-based AOP has an immediate appeal for musical applications, since parallels can be drawn with the event-based nature of the industry-standard Musical Instruments Digital Interface (MIDI) model that is used by many musical software systems and the general observation that music is perceived as discrete audio events in time. Advantages of event-based AOP include the clear separation of aspect and application code. Axon goes further and separates pointcuts from advice. However, the event generation systems employed by the three systems overviewed above are not ideal. Typically, events are 'over eager'; they are generated irrespective of whether they are required by aspects. EAOP's source code modification conflicts with a general AOP requirement of non-invasiveness [1] but the use of the JVMDI imposes a performance overhead that might render it unsuitable for realtime applications.

4.2 Meta Programming & Reflection

Metaprogramming and reflection techniques, in principle, enable a piece of software to both discover and modify its internal structure. Thus metaprogramming enables programs to reason about themselves. To do this, a programming language or environment must expose internal structure such that it can be programmatically manipulated as data, through the process of *reification*. Meta Object Protocols (MOPs) enable structural program elements to be manipulated as object-oriented encapsulations of reified data and appropriate methods. Indeed, AOP has its roots in MOP research [31], indeed AOP itself is a computational reflection mechanism [16].

Meta Programming has been used as an underlying technology that enables AOP. Examples of such enabling technologies include MethodWrappers [4], AOP/ST [6], and Handi-Wrap [2], all of which support the dynamic composition of method code with additional 'wrapper' code that might constitute advice.

AspectS [15] utilises MethodWrappers [4] to implement a dynamic AOP system for Squeak Smalltalk. AspectS defines a set of base classes from which aspects are derived. Aspects are themselves written in Smalltalk, and are dynamically woven or unwoven, by automatically modifying the relevant Smalltalk Class descriptions (as described by the aspect's joinpoint) such that subsequent calls to the original method are redirected to a new wrapped method. The wrapped method, which is dynamically compiled into the Smalltalk system, invokes the aspect's advice and the original method in ways that are analogous to AspectJ's `before()`, `after()` and `around()` advices. It is noted however [16] that although AspectS focuses on message passing, other types of joinpoint, such as member variable access, are not easily achieved through the use of Smalltalk's MOP.

Gybels [12] proposes the use of a Logic Meta Programming Language as an Aspect language. His "Andrew" system uses the Smalltalk Open Unification Language (SOUL) to implement a dynamic aspect system with an aspect model similar to that of AspectJ. SOUL is a variant of PROLOG, but its implementation enables dynamic interaction between itself and Smalltalk. For example, SOUL facts may contain arbitrary dynamic Smalltalk expressions and the result of these expressions may be bound to SOUL logic variables. Andrew implements pointcuts in terms of SOUL predicates that relate to typical AOP joinpoints such as method invocation, methods reception, member variable assignment and so on. Aspects are implemented as the combination of pointcuts and advice. Although Andrew implements `before()` and `after()` advice, there is currently no support for `around()` advice and as such it is currently not possible for aspects to choose not to execute methods, as they can in AspectJ. The use of logic language for aspect, and particularly pointcut, definition is both intuitive and flexible. This is enhanced by the language symbiosis between SOUL and Smalltalk.

Metaprogramming based AOP is heavily dependent on the reflective nature of the underlying language, which may explain why many such approaches target the highly reflective Smalltalk language. A Java equivalent of AspectS, for example, would not be possible because Java's reflection capability is limited to introspection and does not permit intercession [16]. Nevertheless, Java-based dynamic AOP systems that utilise metaprogramming do exist. A hybrid solution for Java, which uses static weaving and a reflective Java environment is presented in [9], while Handi-Wrap [2] uses Java extensions (implemented using Maya [3]) to describe dynamic wrappers that are realised using compile-time reflection..

Metaprogramming appears to be a key enabler of run-time dynamic AOP systems, indeed certain MOP based AOP implementations may be considered as disciplined metaprogramming [16]. We also note that, many musical research systems utilise highly reflective languages; key examples include. Open Music [33] and Symbolic Composer [34] written in LISP, and MODE [25], DMix [22] written in Smalltalk.

In the context of music composition systems, one approach might therefore be to synthesise a dynamic AOP music composition system from existing music systems, such as MODE, in combination with a dynamic AOP system such as AspectS or Andrew. We can also imagine scenarios where introspection into the musical structure itself might be valuable. Consider the example of 'stretching' the length of a section of music. This operation, termed *augmentation*, typically involves multiplying the onset time and duration of sound events by some factor. Thus augmenting four consecutive notes each of 1 second duration by a factor of two yields four consecutive notes each of 2 seconds duration. However, in the case where the music being augmented is, for example, a 'drum roll', then instead of multiplying durations, it is necessary to preserve durations, but add additional notes to fill the augmented duration. Thus augmenting a 'drum roll' of four consecutive notes each of 1 second duration yields a sequence of 8 consecutive notes, each of one second duration. Using introspection, an 'augmentation' aspect could identify the kind of musical structure that was being augmented and invoke the correct behaviour. This *behavioural abstraction* is one of the motivating factors behind the LISP-based Nyquist [35] music system.

Like the use of JVMDI, however, MOP and reflection typically impose a performance overhead [16], which may make them unsuitable for realtime applications.

4.3 Aspectual Components

In principle, it is possible to design aspects that have a more general application than the specific application in which they are first defined. However, aspect systems such as AspectJ and AspectS require that information relating to the static class structure of an application be encoded into aspect definitions. In AspectJ, for example, pointcuts must refer to specific class and method names. Thus aspects may only be re-used in applications that include a class subgraph that matches that referenced by the pointcut definitions. Further, it has been observed that undisciplined construction of aspects in AspectJ prevents aspects from being extended and reused [13].

In the spirit of structure-shy Adaptive Programming [18], Aspectual Components [19] permit aspect binding to be deferred by introducing a level of abstraction that divorces the aspect definition from an particular class structure or method protocol. This is achieved by defining an aspect, termed a *component*, in terms of its own class structure or Participant Graph (PG) that represents an abstract slice through a set of possible concrete class graphs (CGs). Aspects are subsequently bound to a CG using *connectors* that map both classes and methods to the PG. Thus aspects may be defined as discrete crosscutting components that may be applied to any given application class graph through separate connector specifications. It is unclear, however, how ACs can be made to handle dynamic context, and thus avoid Jumping or Vanishing aspects.

A practical implementation of ACs is provided by the JAsCo system [32]. In JAsCo, the standard java component metaphor, JavaBeans, is extended to form *aspect beans*. Aspect beans correspond to the component structure of ACs and encapsulate the behavioural properties of an aspect, providing an abstract interface through which these behaviours may be invoked. Like ACs, JAsCo utilises the concept of a *connector* to establish a relationship between aspect behaviour and a concrete class graph. JAsCo also supports the dynamic insertion and removal of aspect beans.

Aspectual Components form the basis of technologies that enable aspects to be defined as 'plug-ins' across a range of applications and as such, promote re-use. In a musical context, the use of Aspectual Components would permit the definition of a range of crosscutting musical concerns that could then be applied to multiple composition projects.

5. CONCLUSIONS

In this paper we have outlined that music composition is a domain in which multidimensional scattering and tangling is very much evident. We have described that the traditional role of the music composer is to manually weave together these

dimensions to form the musical surface that is perceived by listeners.

We have suggested that aspects could be used to help in the construction of musical composition systems that enable composers to express musical intent, and that perform low-level weaving of musical data based upon higher level musical descriptions. We note, however, that the relationships between musical dimensions, even over common high-level concepts, such as crescendo, are not static, and depend both upon the composer's wishes and musical context. We believe that dynamic aspects offer a way to manage these dynamic crosscutting concerns in the provision of AOP-based interactive music composition tools.

We have briefly outlined some of the current dynamic aspect technologies and indicated their various strengths and weaknesses and possible uses in relation to musical composition applications.

Our future research will consider ways in which dynamic aspects may be implemented and used in the development of an interactive computer based music composition system. Key areas of interest are the development of a dynamic aspects system that supports the requirements of music composition, in terms of the modelling and implementation of dynamic musical relationships, and the extension of the AOP paradigm to the user-level. In particular we wish the user to be able to interactively and dynamically define, apply, and modify crosscutting relationships and to store them for future application in other musical composition projects. As an interactive system, we require dynamic aspects that are responsive, and with the potential for them to support music generation in realtime. As an end-user application, we also require stability and simplicity.

6. REFERENCES

[1] Aussmann, S., Haupt, M. Axon – Dynamic AOP through Runtime Inspection and Monitoring. ASARTI Workshop 2003.

[2] Baker, J., Hsieh, W. Runtime Aspect Weaving Through Metaprogramming. AOSD 2002.

[3] Baker, J., Hsieh, W. C. Maya: Multiple-Dispatch Syntax Extension in Java. In Communications of the ACM. 2002.

[4] Brant, J., Foote, B., Johnson, R. E., Roberts, D. Wrappers to the Rescue. ECOOP 1998.

[5] Brichau J., De Meuter W., De Volker K. Jumping Aspects. Workshop of Aspects and Dimensions of Concerns ECOOP. 2000.

[6] Bollert, K. On Weaving Aspects. In Proceedings of Aspect-Oriented Programming Workshop at ECOOP 1999.

[7] Brooks, F. P. No Silver Bullet: Essence and Accidents of Software Engineering. Computer, vol 20, no. 4. 1987

[8] Dannenberg, R. B., Desain, P., Honing, H. Programming Language Design for Music. In G. De Poli, A. Picialli, S. T. Pope, & C. Roads (eds.), Musical Signal Processing. 271-315. Lisse: Swets & Zeitlinger. 1997.

[9] David, P-C., Ledoux, T., Bouraqadi-Saâdani, N.M.N. Two-Step Weaving with Reflection using AspectJ.

[10] Desain, P., Honing, H. Tempo Curves Considered Harmful. Contemporary Music Review. 7(2). 1993.

[11] Douence, R., Sudholt, M. A model and a tool for Event-based Aspect-Oriented Programming (EAOP) . TR 02/l 1/INFO, lcole des Mines de Nantes, french version accepted at LMO'03, 2nd edition, Dec. 2002

[12] Gybels, K. Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure. Ph.D. Thesis 2001.

[13] Hannenberg, S., Unland, R. Using and Reusing Aspects in AspectJ. OOPSLA 2001

[14] Hill, P., Holland, S., Laney, R. C. Using Aspects to Help Composers. Technical Report TR 2003/21. Open University Dept of Computing 2003.

[15] Hirschfeld, R. Aspect-Oriented Programming with AspectS. 2002

[16] Kojarski, S., Lieberherr, K., Lorenz, D. H., Hirschfeld, R. Aspectual Reflection. AOSD SPLAT Workshop. 2003.

[17] Lerdahl, F., Jackendoff, R. A Generative Theory of Tonal Music, MIT Press, 1983.

[18] Lieberherr, K. J., Silva-Lepe I., Xiao C. Adaptive Object-Oriented Programming using Graph Customisation. College of Computer Science, Northeastern University, 1994.

[19] Lieberherr, K., Lorenz, D. and Mezini, M. Programming with Aspectual Components. Technical Report, NU-CCS-99-01, March 1999.

[20] Loy, G., Abbott, C. Programming Languages for Computer Music Synthesis, Performance and Composition. ACM Computing Surveys, Vol.17, No. 2. June 1985

[21] Oppenheim, D.V. Towards a Better Software-Design for Supporting Creative Musical Activity. ICMC 1991.

[22] Oppenheim, D. DMIX: A multi faceted environment for composing and performing computer music. Mathematics and Computers, 1996.

[23] Pearce, M., Wiggins, G. A. Aspects of a Cognitive Theory of Creativity in Musical Composition. Dept of Computing, City University, London. 2002.

[24] Piston, W. Orchestration, Gollancz 1961.

[25] Pope, S. Introduction to MODE: The Musical Object Development Environment. In The Well-Tempered

Object: Musical Applications of Object-Oriented Software Technology, S. T. Pope, ed. MIT Press. 1991.

[26] Popovici, A., Gross, T., Alonso, G. Dynamic weaving for aspect oriented programming. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, April 2002

[27] Raes, W-G. "The multitasker as an approach in musical composition" 1997 http://logosfoundation.org/g_texts/multitaskers.html

[28] Shoenberg, A. (Strang F, Stein L. eds). Fundamentals of Music composition, Faber and Faber. 1967.

[29] Sloboda, J. A. The Musical Mind. The Cognitive Psychology of Music. Oxford University Press. 1985.

[30] Speigel, L. Old Fashioned Composing from the Inside Out: On Sounding Un-Digital on the Compositional Level. Proceedings of the 8th Symposium on Small Computers in the Arts, Nov. 1988.

[31] Sullivan, G. T. Aspect-Oriented Programming using Reflection. OOPSLA 2001.

[32] Suvée, D., Vanderperren., W., Jonckers., V. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. AOSD 2003.

[33] http://www.ircam.fr/produits/logiciels/openmusic-e.html

[34] http://www.mracpublishing.com/scom/

[35] http://www-2.cs.cmu.edu/~rbd/doc/nyquist/root.html