Technical Report No: 2003/21

Using Aspects to Help Composers

Patrick Hill
Simon Holland
Robin C. Laney

15th December 2003

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

http://computing.open.ac.uk

Using Aspects to Help Composers

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

Current AOP and related research has largely focussed on the development of technologies that assist software engineering practitioners in the separation and composition of various dimensions of concern across a range of software engineering tasks. In this paper we argue that the principles of AOP might also be usefully applied in supporting user interaction with software systems that aim to support multidimensional, non-linear, creative processes such as music composition. We support our argument with two concrete examples of AOP approaches applied to a musical context.

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 *AspectJ^{TM}, Hyper/J^{TM}*; D.3.3 **[Programming Languages]:** Language Constructs and Features - *Aspects, Hyperspaces;* J.5 **[Computer Applications]:** Performing Arts - *Music*

General Terms

Design, Languages, Human Factors.

Keywords

Aspect-oriented programming, music composition.

1. INTRODUCTION

Aspect oriented programming (AOP) and related technologies such as Multidimensional Separation of Concerns (MDSoC) have been proposed as methods of managing the re-composition of separately specified dimensions of concern in computer software. This approach has been motivated by the observation, dubbed the 'tyranny of the dominant composition' [13], that the encapsulation of concerns through a single decomposition method leads to crosscutting resulting in tangled and / or scattered implementation [8].

Current AOP research has focussed on the varied interests of the software engineering practitioner in managing scattering and tangling at various levels of interest. Programming language extensions such as AspectJ [23], and related systems such as AspectS [7], address crosscutting at the code level by 'weaving' together concerns at well defined points, *joinpoints*, in a programs execution.

The concept of Hyperspaces and the Hyper/J^{TM} tool [13] address the composition of software from separately identified concerns, described in a *concern mapping*. This composition is performed at build-time by defining a *hypermodule* that acts as a 'recipe' for composing program functionality from separately implemented program code.

Other approaches such as Adaptive Programming [10] and component systems such as JAsCo [20] address issues in the separation of structural concerns from functional concerns.

These technologies are aimed at resolving scattering and tangling of user and developer requirements 'under the hood'. We believe that the underlying principles of some of these *compositional* techniques and related *non-compositional* technologies, such as Visual Separation of Concerns [4] may also be usefully applied at the *end-user level*, in systems that support application domains in which tangling and scattering across multiple dimensions of concern are present.

This paper focuses on the potential use of aspects and related technologies in the domain of music composition. Music composition is a creative process in which 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.

Although the processes of software engineering and music composition are clearly different, we believe that there exist certain parallels and analogies between two. The usability of musical notation systems, for example, can be assessed in the same way as the usability of programming languages [3]. We can 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, these outputs are, largely, simply the result of the composition of higher-level abstractions and multiple dimensions of concern, and both are broadly 'performance instructions'. In the case of a software engineering process, this transformation is often performed automatically through compilers and other 'software build' tools that leave the source artefacts intact. However, the musical composer, even with computer assistance, often has to express high-level musical ideas in terms of varying degrees of tangled, low-level detail in which higher-level abstractions are typically poorly represented and compositional intent is often lost [22]. These issues appear to mirror those that are addressed in the software domain by aspect technologies.

In this paper we examine the parallels more closely in order to gain a detailed idea of how the principles of AOP might be applied to music composition. We support our argument using two concrete examples, and note some of the implications of this approach.

2. SEPARATION AND COMPOSITION OF CONCERNS IN MUSIC

While many software systems exist that assist in particular elements and tasks of musical composition, each system implements its own partial musical ontology that maps to its areas of interest. For example, it is possible to consider the perceived *musical surface*, purely in terms of the *principal perceptual dimensions (PPDs)* of *pitch, duration, loudness* and *timbre* [11]. Many musical representation systems, including scoring systems and the ubiquitous Musical Instruments Digital Interface (MIDI) adopt this particular ontology. By contrast, other approaches to music representation, such as Balaban's Structured Music Pieces [1], consider music in terms of its temporal and structural relationships, but largely ignore other musical dimensions such as melody, harmony and orchestration. The related, but more detailed ADTs of Smaill et al [18] include PPD information with the hierarchy, while Lerdahl and Jackendoff [9] attempt to describe music in terms of a formal grammar. However, no current musical representation completely represents all musical dimensions at the same time.

It is common experience that music is not merely a random stream of *sound events.* Dannenberg suggests that the musical surface may be considered as the *result* of the composers weaving together of a 'tangled web' [5] of musical structures and dimensions. For example, the musical gesture of 'getting louder' (*crescendo*), might be realised by simply instructing the performers to play louder; increasing the 'loudness' dimension. Another alternative is that the crescendo might be realised by the gradual introduction of additional instruments or use of higher / lower pitches. Thus, in this and other ways, the task of orchestrating a musical work becomes an inseparable from its composition [15]. In this example, the orchestration becomes tangled with the general 'loudness' dimension and the 'crescendo' concern is scattered among the instruments and their individual loudness dimensions.

In any given musical composition, composers tend to use limited musical resources and manipulate them in various ways to form a logical and coherent whole [16]. For example, a composer might wish to reuse a rhythm introduced in one area of a composition in a different context somewhere else. Systems whose ontologies do not permit explicit identification and separation of musical concerns, such as rhythm, force the composer to work in a detailed 'note-list' fashion, rather than a more natural expression of intent. Oppenheim [12] argues that it is unreasonable to expect a composer to work in this way, and the dichotomy between natural and formal musical expression continues to be one of the principal issues in computer music composition.

Various authors have written on the subject of exactly *how* a composer composes music. Composition does not appear to be a linear process; rather the composer originates and refines musical ideas that may then be incorporated into the composition at hand. In [14], it is suggested that the composition process requires hierarchic representations and that varying degrees of scattering and tangling exist both within these representations and within the cognitive processes of composition. In an illuminating account, Spiegel [19] describes her own preference for a paper-based system because of the freedom it offers in moving around between different hierarchic levels and musical dimensions, filling in complete or partial ideas, and composing in an iterative fashion. Speigel also identifies that sometimes the form of a composition evolves rather than being predetermined from the outset, and that musical ideas which are formed may be discarded from the current composition, or reworked into new compositions.

It has also been observed [17][14][19] that composers do not necessarily fully complete aspects of their compositions (such as structure, harmony, melody, orchestration etc.) before moving on to others, but rather move between them. Sloboda [17] argues that this may be, in part at least, because the composer records only those aspects that are important, and serve as aides' memoirs to help recover the un-notated aspects.

Both Schoenberg[16] and Hindemith[17] subscribe to the view of a 'vision' of an overall composition, wherein the compositional process reduces to simply filling in the detail of this vision. Other accounts [17] suggest a more methodical approach. Nonetheless, an important point that arises is the need to capture the essence of the composition, across whatever dimensions this 'essence' might exist, for example a formal plan that specifies the overall structure of the piece, or a melodic line and so forth.

The underlying principles of AOP and MDSoC could be used to provide a way to 'meld' or 'weave' together separately described musical elements that express musical *intent* and provide support for an extensible ontology that can operate at any level of abstraction determined by the composer. This weaving could result in the automated production of a 'score' or other notation, freeing the composer from the tedium and possible inaccuracies of transcription. An AOP-based musical creation environment could enable the composer to work in an iterative and experimental fashion, defining and extending a musical ontology that suits their purpose, in ways analogous to those in which AOP and MDSoC free the software engineer from the "tyranny of the dominant decomposition" [13].

The remainder of this paper contains two concrete examples that illustrate some of the ways in which musical dimensions interact, and how their separation and subsequent re-composition may be achieved in an AOP/MDSoC fashion. Rather than attempting to compose new pieces, our approach in this section is to show how aspects may be used to 'compose' existing pieces.

3. EXAMPLE 1: METRICAL AND GROUPING STRUCTURES

In this example we consider the use of an aspect to weave together two separate musical dimensions, namely metrical structures and grouping structures.

3.1 Metrical Structures

Music is experienced in time, and in western music, the time dimension is typically dictated by a regular pulse or *beat*. Generally, short musical sections containing the same number of beats are grouped together into structures called *bars.* The number of beats contained in a bar is dictated by a *time signature*.

Musically speaking, each beat of a bar is felt to be *strong* or *weak*, depending on the time signature, and this relationship between strong and weak beats is known as *metre*. For example, a time signature of four beats in a bar implies that the first and third beats are strong, and that the second and fourth beats are weak. However, this categorisation does not account for the common experience that many listeners can correctly identify the first beat of a bar, and do not confuse it with the third. In [9], Lerdahl and Jackendoff suggest that a *metrical hierarchy* exists, with the result that the first beat is stronger than the third, and both are stronger than the second and fourth. The realisation of strong beats is termed *metrical accent*.

3.2 Grouping Structures

In [9], g*rouping structures* are described as hierarchical structures that group sequences of pitch / duration pairs, such as those that constitute a melody, according to various grammatical rules. The detail of the grouping rules is irrelevant to this example. What is important however is the observation that metrical hierarchies and grouping structures, while obviously interacting with each other, are nonetheless, separate dimensions of musical concern. The authors explicitly guard against an analytical viewpoint that attributes metrical stress to melodic groupings.

This statement may be considered as being analogous to the definition of crosscutting in software [8]. However here it is the metrical and grouping structures that need to be defined separately but be coordinated, rather than software concerns.

3.3 A Musical Example

As in [9], we use the theme of Mozart's Symphony No 40 in G minor. Figure 1 shows the high-level grouping structure of the theme itself. This structure shows only pitch and duration. There is no indication of metre, such as time signature or bar-lines; therefore all notes receive equal stress.

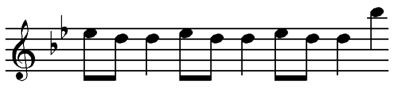

Figure 1

Mozart's original piece is in 4/4, so metrical stress is applied to the first and third beats, with beat one being stronger. Mozart starts the theme on the last beat of the bar (an *up-beat* or *anacrusis*) as shown in Figure 2.
The metrical accent is denoted by > symbols, with $\overset{>}{>}$ denoting the stronger accent.

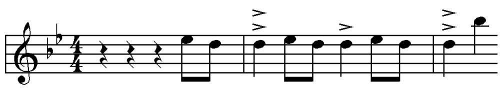

Figure 2

However, the separation of the grouping and metrical structures enables us change the metre at will, without re-specifying the melodic group.

A simple variation is to start the group on the first beat of the bar as shown in Figure 3.

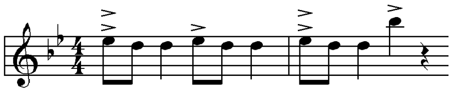

Figure 3

Although the notation looks similar, in performance the two would feel quite different, even though both the grouping structure and the metrical structure of both examples is identical. In other words, it is the composition of the two dimensions that give the music its metrical feel.

Now we can change the metre to, say, that of a 3/4 'waltz', with metrical stress now applied only to beat one, as shown in Figure 4.

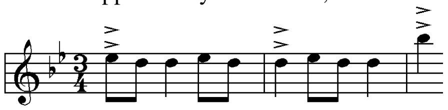

Figure 4

3.4 A Simple Sequencer

Consider a simple musical sequencing system that 'plays' the non-metrical grouping structure, with each note being represented by its principal perceptual dimensions of pitch, duration, and loudness. We ignore timbre for the purposes of this example.

One implementation might consist of the following:

- `MusicalEvent` objects:
  `MusicalEvent` objects encapsulate PPD parameters (pitch, duration and loudness) and know how to 'play' themselves.

- A `MusicalSequence` object:
  The `MusicalSequence` represents an ordered sequence of `MusicalEvent` objects. It is possible to fetch the next `MusicalEvent` in the sequence.

- A `Clock` object:
  The `Clock` object is a simple implementation of an Observer pattern [6] that supplies periodic clock signals (ticks) to a registered observer object. In this example, we use the MIDI standard of 24 clock ticks per beat.

- A `MusicSequencer` object:
  The `MusicSequencer` object is responsible for reading the `MusicalSequence` and starting and stopping the playing of `MusicalEvent` objects at appropriate times.
  `MusicSequencer` registers itself as a `Clock` observer; ticks received from the `Clock` object therefore control timing.
  Since `MusicalEvent` durations are expressed in clock ticks, `MusicSequencer` simply counts the ticks

that elapse once a MusicalEvent has started playing. When the count equals the MusicalEvent duration, MusicSequencer asks the current MusicalEvent to stop playing, and fetches a new MusicalEvent from the MusicalSequence, resets its tick counter, and asks the new event to play itself. This cycle continues until the MusicalSequence has been completed.

Appendix A shows a Java^{TM} implementation of this system.

3.5 Metrical Stress using Aspect/J^{TM}

Now we consider adding metrical stress to this simple application. We will adopt the, admittedly naïve, practice of stressing by increased loudness. Metrical stress clearly involves two distinct considerations

1) Determination of metrically strong beats.
2) Modification of the loudness parameter of a MusicalEvent that coincides with a strong beat.

Let us first consider how we could adapt the current system without using aspects. An initially attractive approach might be to use MusicalSequence to add metrical stress for us, by keeping track of beats as MusicalEvents are added, and modifying the loudness parameter of those that fall on strong beats. However, this approach clearly tangles the metrical stress concern with MusicalSequence's basic concern. Moreover, this is a destructive approach, and the MusicalSequence could not be re-used in a different metrical context. There are a number of other possible ways of modifying the given objects to implement metrical stress. However, we believe that any of these solutions will necessarily involve tangling of basic concerns and / or scattering of the metrical stress concern throughout the object model.

Let us now consider an AOP solution. In this solution we shall introduce an Aspect that is entirely responsible for the implementation of metrical stress, and requires no modification to the existing classes.

Firstly, the determination of a metrically strong beat requires the aspect to be notified of clock ticks, and to count them according to some time signature. This can be achieved with an advice that is invoked when the Clock object issues a tick.

Secondly, coincidence of a MusicalEvent with a metrically strong beat can be achieved with an advice that is invoked around the playEvent() method of MusicSequencer. The advice checks to see if the current beat count is metrically strong, and if so, constructs a copy (clone) of the MusicalEvent, and modifies its loudness parameter. It then tells the MusicSequencer that its currently playing event is this new copy. Thus when the advice proceeds, the modified MusicalEvent is used to produce the accent, leaving the original MusicalSequence intact.

The following code fragment shows an aspect that might be used with the code presented in Appendix A to implement a simple metrical accent, in real-time.

```
aspect MetricalAccent {

    // the initial values of tick,
    // and beat set the starting point
    // in the metre.
  private int tick = 0;
  private int beat = 4;
  private static final int
    beatsPerBar = 4;
  private static final int
    ticksInABeat = 24;
  private MusicalEvent evtCopy = null;

  // Static introduction of 'accent'
  // method into MusicalEvent, enables
  // the aspect to modify loudness.

  public void
    MusicalEvent.accent(int multiplier)
        { loudness *= multiplier; }

  // Metrical accent is dependent on
  // the beat-of-the-bar
  // This advice keeps track of the
  // beat-of-the-bar by counting clock
  // ticks against the offset of
  // the initial tick and beat values.

  before():
   call(void Clock.tick()) {
     tick++;
     if(tick > ticksInABeat) {
        tick = 1;
        if( beat < beatsPerBar )
          beat++;
        else
          beat = 1;
     }
   }

  // This advice 'implements' the
  // metrical accent. The advice runs
  // around the call to
  // MusicSequencer.playEvent

  pointcut
    playCut(MusicSequence seq,
       MusicalEvent evt) :
       call(void
        MusicSequencer.
        playEvent(MusicalEvent))
     && args(evt) && target(seq);

  void around(MusicSequencer seq,
       MusicalEvent evt)
       : playCut(seq,evt) {

    evtCopy =
      (MusicalEvent)evt.clone();

    if( 1 == beat && 1 == tick )
      evtCopy.accent(3);
    if( 3 == beat && 1 == tick)
      evtCopy.accent(2);

    seq.event = evtCopy;
    proceed(seq,evtCopy);

    }
}
```

4 EXAMPLE 2: MULTIDIMENSIONAL COMPOSITION

The previous example shows how two crosscutting dimensions, grouping and metrical, might be 'woven' together using aspects. The example we present here is more complex, and involves multiple dimensions.

The piece of music that we consider in this example is Widor's "Toccata" from his Organ Symphony No 5 [21], a well-known organ piece. We focus our attention solely on the construction of the right and left hand parts of just the opening four bars of the piece, and we have made slight simplifications of the actual piece in order that the discussion is not overly complicated.

In order to identify some of the dimensions that appear to exist, we offer the following informal analysis of this short musical extract. We will explain only as much musical theory as is required in order to understand the example.

1. Metre

The metre of the piece is 4/2. For the sake of this discussion, this is metrically equivalent to 4/4, as discussed in section 3.1.

2. Tonal Plan

The western tonal system consists of an *octave,* being the frequency space between two pitches at *f*Hz and 2*f*Hz, and its subdivision into twelve *pitch classes.* To illustrate this, Figure 5 shows a complete octave of a piano keyboard.

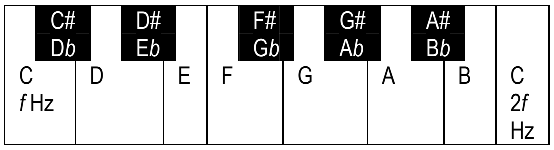

Figure 5

The distance between consecutive keys is called a *semitone*, and a distance of two semitones is called a *tone.* Sequences of tone and semitone intervals may be used to traverse a complete octave, forming a *scale*. In western tonal music, two types of scale, *major* and *minor* are common. This example considers only Major scales, formed by the sequence

Tone, Tone, Semitone, Tone, Tone, Tone, Semitone.

Thus, as illustrated in Figure 5, the major scale starting on C consists of

C D E F G A B

This example uses the major scale starting on F, which consists of

F G A Bb C D E

The starting note of a scale is called its *tonic*, while the fifth note is called the *dominant*. The names of other notes of the scale, or scale *degrees*, are not relevant to this example.

The tonal plan of the opening bars is two bars in the tonic, followed by two bars in the dominant major; specifically, two bars using the notes of the F major scale, followed by two bars using the notes of the C major scale. The implementation of the Tonal Plan crosscuts the metre, because although we have specified each section in terms of *bars*, the tonal plan is independent of what a bar actually *is*.

3. Harmonic Progression

Notes can be played simultaneously forming *chords.* Western tonal harmony is based on the concept of the triad. The notes of the triad consist of successive *thirds* with a third being either 3 semitones (*minor third*) or 4 semitones (*major third).*

Thus a triad formed on F in F major consists of the notes

F A C

Typically, triads are notated in roman numerals representing the scale degree of the lowest note (the *root*). So the F major triad in F major is chord I, and the C major triad in F major is V. The notation may be extended to show additional notes. Thus, the notation I7, indicates a triad formed on the first degree of the scale, but with an additional note which is a major 7th above the root.

Thus, in F major, the chord I7 consists of the notes

F A C E

This Roman numeral notation is independent of key.

Sequences of chords or *harmonic progressions* may be constructed. The following harmonic progression is observed in the Toccata.

I, I7, I6, I7, I, I7, I6,V of V

The notation V of V indicates chord V of a scale formed on the fifth degree of the current scale. So in F Major, V of V is the fifth chord of C major (i.e. G major).

4. Harmonic Rhythm

Harmonic rhythm describes the rhythm of change of harmony. In the toccata, the harmonic rhythm changes chord on each beat of the bar. Since there are eight chords in the progression, this means that the Harmonic Progression extends over two bars.

Tables 1a and 1b summarise the interaction of these four dimensions over the first four bars of the work. The harmonic rhythm is notated as •. The resulting actual harmonic progression, with harmonic rhythm, shown on the bottom row, is the product of 'weaving' the Tonal Plan, Harmonic Rhythm, and Harmonic Progression concerns.

Table 1a

Bar	1				2			
Beat	1	2	3	4	1	2	3	4
Tonal Plan	Tonic (I)							
Harmonic Rhythm	•	•	•	•	•	•	•	•
Harmonic Progression	I	I7	I6	I7	I	I7	I6	VofV
Actual Chord	**F**	**F7**	**F6**	**F7**	**F**	**F7**	**F6**	**G**

Table 1b

Bar	3				4			
Beat	1	2	3	4	1	2	3	4
Tonal Plan	Dominant (V)							
Harmonic Rhythm	•	•	•	•	•	•	•	•
Harmonic Progression	I	I7	I6	I7	I	I7	I6	VofV
Actual Chord	**C**	**C7**	**C6**	**C7**	**C**	**C7**	**C6**	**D**

4.2 The Left Hand Part

The left-hand part consists of a repeating, one-bar rhythmic sequence that is superimposed over the harmonic progression and harmonic rhythm. Tables 2a and 2b represents the rhythmic sequence. Each vertical division represents a 32nd of the bar, or 1/8 of a beat. Beat numbers are marked, for ease of reference. Grey boxes represent sound events while black boxes represent rests. The width of each box represents its duration. For clarity, the onset of a sound event is marked with •.

Table 2

1								2							
•		•		■	■	•	•	•		•		■	■	•	

Table 2b

3								4							
•		•		■	■	•		•		•		■	■	•	

We can define a 'weaving' operation that brings together this rhythmic dimension, and the harmonic rhythm derived above, by simply playing the chords that coincide with the LH rhythm. This produces the four bars shown in tables 3a-3h. Here, the actual chord is notated instead of •.

Table 3a

1								2							
F		F		■	■	F	F	F7		F7		■	■	F7	

Table 3b

3								4							
F6		F6		■	■	F6		F7		F7		■	■	F7	

Table 3c

1								2							
F		F		■	■	F	F	F7		F7		■	■	F7	

Table 3d

3								4							
F6		F6		■	■	F6		G		G		■	■	G	

Table 3e

1								2							
C		C		■	■	C	C	C7		C7		■	■	C7	

Table 3f

3								4							
C6		C6		■	■	C6	C6	C7		C7		■	■	C7	

Table 3g

1								2							
C		C		■	■	C	C	C7		C7		■	■	C7	

Table 3h

3								4							
C6		C6		■	■	C6	C6	D		D		■	■	D	

This represents exactly the harmonic and rhythmic elements of the left-hand part of the piece.

4.3 The Right Hand Part

The right-hand part of the first four bars of the Toccata is composed of the notes of a chord played singly rather than in unison; an *arpeggio* figure. An example, showing the first two beats worth of the right hand part, is shown in Common Practice Notation in Figure 6.

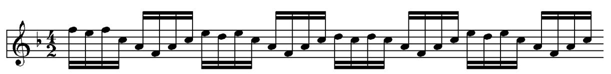
Figure 6

The arpeggiated chord is dictated by the combination of the same tonal plan, harmonic rhythm and harmonic progression as was used in the left-hand part. However, the right hand rhythm is a so-called *moto perpetuo*, with eight notes of equal duration in a beat. The arrangement of notes that constitute the arpeggio may be determined algorithmically, as a function of the underlying chord.

4.4 An MDSoC Expression of the Toccata.

From the foregoing, it is apparent that multiple dimensions of concern are in operation in the composition of this work. Informally we can consider the Left Hand part to be composed of the Metre, Tonal Plan, Harmonic Progression, Harmonic Rhythm and Left Hand Rhythm concerns. The Right Hand part can be considered to be composed of the Metre, Tonal Plan, Harmonic Progression, Harmonic Rhythm, Right Hand Rhythm and Right Hand Arpeggiation concerns.

This analysis bears some resemblance to the *hypermodule* specification of Hyper/J^{TM} [13], and we can apply the examples given in [13] to this musical context.

On-Demand Remodularisation

Enables the composer to remove or reconfigure the way in which musical concerns are composed. For example, we might choose to omit the Tonal Plan concern.

Adding additional concerns

Enables the composer to perhaps refine his work. An example might be to add in a metrical stress concern as explained in example 1.

Retrofitting crosscutting concerns.

An example might be adding an orchestration concern to the music. This necessarily crosscuts the right-hand and left-hand dimensions.

5. CONCLUSIONS

We have shown that multidimensional tangling and scattering exists within the structure, representation and manipulation of musical data, and also in the cognitive processes of musical composition.

Listeners generally perceive a musical composition in terms of the audio signals that constitute its musical surface. We have outlined that the musical surface of a composition may be expressed through the principal perceptual dimensions of pitch, rhythm, loudness and the distinctive audio qualities of particular sound sources that fall under the umbrella term of ‘timbre’. We have shown that at a higher level, music consists of various structures, both hierarchical and non-hierarchical, and that the musical surface is overlaid by a tangle of such structures and musical dimensions. The tangling that exists between musical dimensions is not always static. For example, we cannot say that an increase in loudness (*crescendo*) is always effected by simply playing louder. Rather, the composer might realise a crescendo by increasing the number or type of instruments that are playing (orchestration), using alternative chords (harmony) and so on. Thus the implementation of, in this case, a crescendo might be scattered among other musical dimensions.

We have also described some of the ways in which the cognitive processes of composition are tangled, and that composers might wish to move between possibly incomplete and tangled dimensions throughout the compositional process. Finally we have broadly described the use of AOP in two specific examples showing how various musical dimensions may be composed to form a musical work.. We have noted, for example, that orchestration is inseparable from the compositional task [15], but that a composer might choose to orchestrate incomplete musical sections.

To our knowledge, no current computer music representation directly addresses the tangling of musical dimensions. We do not suggest that all music can necessarily be represented in terms of the separation and composition of its various dimensions. However, we think that the ability to explicitly express the interrelationships that exist between various musical dimensions, and manipulate and compose these dimensions separately would represent a significant contribution to computer music systems, both from the perspective of the composer and the musical analyst. In particular we feel that aspects / MDSoC go some way to addressing two particular issues found in computer music systems. Firstly it seems likely that, certain composers at least, already think in terms of multiple dimensions [14][17][19], and that aspects / MDSoC could provide a mechanism for capturing or analysing these ideas, simultaneously adding support for natural expression and supporting the cognitive processes of composition. Secondly, since the separation and re-composition of concerns is entirely flexible, we think that such an approach might free the composer from the “tyranny of the dominant ontology” and enable the analyst to experiment with different analytical perspectives.

6. REFERENCES

[1] Balaban, M. The Music Structures Approach in Knowledge Representation in Music Processing. Computer Music Journal (Summer 1996).Vol 20 (2).

[2] Belkin, A. A Practical Guide to Musical Composition. http://www.musique.umontreal.ca/personnel/Belkin/bk/. 1995-1999.

[3] Blackwell, A.F., Green, T.R.G., Nunn, D.J.E. Cognitive Dimensions and Musical Notation Systems. ICMC 2000.

[4] Chu-Carroll, M.C., Wright, J., Ying, A.T.T. Visual Separation of Concerns through Multidimensional Program Storage. In proceedings AOSD. 2003.

[5] 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.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

[7] Hirschfeld. R. Aspect-Oriented Programming with AspectS. DoCoMo Communications Laboratories Europe, 2002.

[8] Kiczales, G. et al. Aspect Oriented Programming. In proceedings of ECOOP. June 1997.

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

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

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

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

[13] Ossher, H., Tarr., P. Hyper/J™ User and Installation Manual, IBM Corporation. 2000.

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

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

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

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

[18] Smaill, A., Wiggins, G., Harris, M. Hierarchical Music Representation for Composition and Analysis. 1993.

[19] 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.

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

[21] Widor., C. M. Organ Symphony N^{o} 5. Schirmer. 1954.

[22] Wiggins, G., Miranda, E.R., Smaill, A., Harris, M. Surveying Musical Representation Systems. Computer Music Journal. Fall 1993 Vol 17(3).

[23] The AspectJ Programming Guide, Xerox Corporation. 1998-2002.

APPENDIX A – SIMPLE SEQUENCER

```
import java.util.*;

class MusicalEvent {

  String pitch;
  int durationInTicks;
  int loudness;

  MusicalEvent(String p, int d) {
    pitch = p;
    durationInTicks = d;
    loudness = 50; // Default loudness.
  }
  MusicalEvent(String p, int d, int l) {
    pitch = p;
    durationInTicks = d;
    loudness = l;
  }

  public int getDuration()
    { return durationInTicks; }

  public String getPitch()
    { return pitch; }

  public int getLoudness()
    { return loudness; }

  public String toString()
    { return "Pitch=" + pitch +
            " Duration=" + durationInTicks +
            " Loudness=" + loudness; }

  public void play() {
    System.out.println("Playing  " +
    toString());
  }

  public void stop() {
    System.out.println("Stopping" +
    toString());
  }
}

class MusicSequence {

  LinkedList sequence = new LinkedList();
  ListIterator iterator = null;

  public MusicSequence() { reset(); }
}

  public void add(MusicalEvent e)
    { sequence.add(e); }

  public void reset()
    { iterator = sequence.listIterator(); }

  public MusicalEvent getNext() {
    return  (MusicalEvent)(iterator.hasNext() ?
            iterator.next() : null);
  }
}

class MusicSequencer  {

  MusicalEvent event;
  MusicSequence sequence;
  int currentDuration = 0;
  int tickCounter = 0;

  MusicSequencer(MusicSequence seq)
   { sequence = seq; }

  private MusicalEvent  fetchNextEvent() {
    event = sequence.getNext();
    if(null == event) return null;

    currentDuration = event.getDuration();
    return event;
  }

  public void tick() throws Exception {
    if(tickCounter == currentDuration) {
      if( event != null ) event.stop();
      if(null == fetchNextEvent())
         throw new
            Exception("End of Sequence");

      tickCounter = 0;
      event.play();
    }
    tickCounter++;
  }
}

class Clock extends Thread {

  MusicSequencer sequencer = null;
  boolean eventsAvailable = true;
  float sleepMillis = 20; // Default 120bpm
  public Clock(MusicSequencer l)
    { sequencer = l; }

  public void setTempo(int bpm)
    { sleepMillis = (1000 / (bpm / 60)) / 24; }

  private void tick() throws Exception
    { sequencer.tick(); }

// run() is called by Thread.start()
   public void run() {
     while(eventsAvailable) {
       try {
         tick();
        Thread.sleep((long)sleepMillis);
       }
       catch(Exception e) {
           eventsAvailable = false;
       }
     }
   }
}
```

```
class MetricalAccentDemo {

  static public void main(String[RR] args) {
    // Construct a sequence
    MusicSequence   sequence =
      new MusicSequence();

    sequence.add(new MusicalEvent("Eb",12));
    sequence.add(new MusicalEvent("D",12));
    sequence.add(new MusicalEvent("D",24));
    sequence.add(new MusicalEvent("Eb",12));
    sequence.add(new MusicalEvent("D",12));
    sequence.add(new MusicalEvent("D",24));

    sequence.add(new MusicalEvent("Eb",12));
    sequence.add(new MusicalEvent("D",12));
    sequence.add(new MusicalEvent("D",24));
    sequence.add(new MusicalEvent("Bb",24));

    MusicSequencer player =
      new MusicSequencer(sequence);

    Clock clock = new Clock(player);

    sequence.reset();

    clock.setTempo(60);
    clock.start();
  }
}
```