Charles Fox
Clare College, Cambridge
May 2000

Genetic Hierarchical Music Structures

or

"The Tree beneath the Bach"

Computer Science Tripos
Part II Project

Proforma

Name:	Charles Fox
College:	Clare
Title:	Genetic Hierarchical Music Structures
Tripos:	Computer Science Part II
Year:	2000
Word count:	11,297
Project Originator:	Charles Fox
Project Supervisor:	William Clocksin

Original Aims:

To design and implement a system, *MusicGenie*, for semi-automated composition of meaningful and aesthetically pleasing music.

This will be done by means of hierarchical structural representation together with genetic cross-over and mutation operators, with a human as fitness function.

In addition, the structures should be human readable and editable to allow use of the system as a tool for pure human composition.

Work completed:

MusicGenie achieves its aims. Meaningfulness and aesthetics of its evolution algorithms were demonstrated by creating variations on a Bach fugue. *MusicGenie*'s selective-breeding system was used to evolve a meaningful, aesthetic composition.

MusicGenie enables composers to compile a human-readable language, SARAH, into genetic hierarchical structures. A real composer used it to write a short suite, commenting that it was easy to use and significantly speeded up his composition process.

The evolution and human composition aspects are integrated to increase the power of each. Easy-to-use GUI and web interfaces are provided.

MusicGenie successfully crossbred Bach and the Spice Girls!

Special Difficulties Encountered:

None.

Declaration of Originality:

I Charles Fox of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

"The fugues are bound by strict rules of counterpoint, in which a subject is announced, to be answered in imitation ... the subject itself may appear in inversion, upside-down, or in augmentation, with longer notes, or diminution, with shorter and quicker note values. True art is to conceal art, and this Bach, as always, achieves in music that is never merely subservient to technical requirements."

- Jenõ Jandó, on Bach's *Well Tempered Clavier*

"Even when pop aims to be lyrical, melody is synthesized from standardized units, which could be rearranged in any order without losing or gaining effect. It is not that such music is tuneless: rather it that the tune comes from elsewhere, like food from the supermarket shelf, to be heated in the microwave."

- Roger Scruton

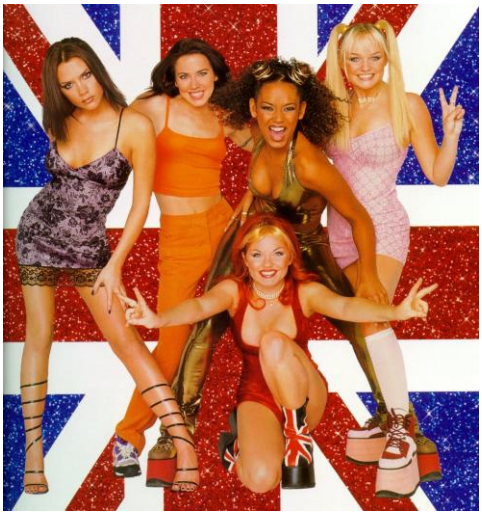
Does music, like poetry
Cry from one's core
Or is it just splicings of licks
And no more?

- Douglas Hofstadter

Contents

1 Introduction

1.1 Hierarchical music structures



The 'Spice Girls' and other modern pop bands are often criticized for being over-simplistic and formulaic. In fact, the majority of pop songs share an identical top-level structure (or simple variants of it):

```
POP SONG -> { INTRO, VERSE, CHORUS, VERSE, CHORUS, MIDDLE 8, CHORUS, OUTRO }
```

Defendants of music with unoriginal forms would argue that rather than this structure being clichéd and predictable, it enables the listener to feel more at home with new pieces in the same style; once a listener has 'learned' a song form, he knows what to expect in new songs which share that form. Sociologists would point out that shared knowledge of such structures is one element of a society's culture.

A lesser-known fact is that most music contains similar structure at lower levels. For example, a Spice Girls `VERSE` may well be decomposable as follows: ('Riff' means 'a short recognizable fragment of tune', such as a repeated line of a song.)

```
VERSE -> { RIFF-A, RIFF-B, RIFF-A, RIFF-B }
```

A `RIFF` might then be broken down into the parts for different instruments (played simultaneously):

```
RIFF -> [ VOCAL-TUNE, BASS-LINE, GUITAR-CHORDS ]
```

And so on, maybe even down to the level of individual notes:

```
VOCAL-TUNE -> { G, G, A, A, C#, A, F#, G }
```

We could imagine describing a complete composition using production rules such as those above – in which case we would be able to represent the composition by a tree structure.

Transformations, such as playing structures in *retrograde* (backwards), in *inversion* (upside-down), or at double speed, are also common. Such hierarchical structures and transforms are easily found in the music of 'mathematical' composers such as Bach, Schöenberg and Reich.

1.2 Music, Meaning and Creativity

Structure as Meaning

Minsky [1] suggested that this structure of music is its very *meaning*; to understand and appreciate a composition is to comprehend its underlying structures. Schenker [2] proposed a system for hierarchically *parsing* compositions, in which basic patterns are first identified, then progressively larger structures constructed above them. Reis [3] and other computer scientists have attempted to automate this parsing. If computer compositions are to be meaningful, then, it seems that we must perform the reverse of this structural process in generating them.

The Grammars of Music

The fundamental difference between parsing music and a natural language is grammar. A language like English has a (reasonably) fixed grammar, and parsing a text is a task of finding a structure for it which fits that grammar. It would appear that there is no universal grammar for the 'language' of music – although it has been suggested that grammars exist to describe the languages of particular composers and styles. Students and computers can easily compose music 'in the same grammar' as famous composers, which may sound as good as the original works, but the genius of composing appears to be in creating the grammar in the first place. So to generate real *new* meaningful music, it does not suffice to use existing grammars – creating the grammar must be the task of the machine.

The need for Human expression

Perhaps the most common attack on the area of automated composition is that a machine has no emotions or aesthetics, so cannot know when it has produced something beautiful or something ugly. Minsky's view would perhaps avoid the question: emotions do not affect the structure of the piece so do not affect its meaning. But if our aim is to produce meaningful and *aesthetically-pleasing* music then we must address this problem.

From these ideas we can see that to produce meaningful and aesthetically-pleasing music we need a system for building new grammars, but we also need that magical ingredient of human emotion.

1.3 Review of previous work in the area

A good review of the whole area of Algorithmic Composition can be found in [9]. Papadopolos and Wiggins divide the work area into the following main models:

- *Mathematical* (eg. Markov Chains, with probabilities gleaned from existing pieces)
- *Knowledge based* (eg. Constraint-satisfaction for harmonisation problems)
- *Grammars* (eg. EMI, discussed below)
- *Evolutionary* (eg. Genetic algorithms, either with heuristic or human fitness function)
- *Learning* (eg. Reis' *Maestro*, discussed below)
- *Hybrid* (combinations of the above)

There have been many previous attempts at automated composition, the best of which are documented in their paper. Those listed below, however, deserve special mention as they have influenced my project to various extents:

Reis's Maestro

Ben Reis' Cambridge PhD Thesis [3] was the starting point for this project. His program, *Maestro*, uses agents to guess the basic patterns in given pieces of music. These patterns are then used to build a parse tree of the whole piece. The program aquires knowledge about grammar, and can be trained on particular styles of music. Unusually, Reis was able to quantifiably test the results of the program: rather than use it to generate new compositions, he fed it with incomplete scores of existing pieces and asked it to predict the next note.

EMI and Mozart's 42nd Symphony

Perhaps the most (in)famous example of automated composition to date is David *Cope's Experiments in Musical Intelligence*. A full description of EMI can be found in [8], but its basic idea is: a large number of existing compositions by a composer is fed in as data. EMI identifies and extracts short patterns, which are then reassembled using two principals - *Local Flow* means that fragments are placed so their joins are as seamless as possible; *Global Positioning* means that fragments are placed in 'analagous' locations to their original home. EMI received media attention for creating a new 'Mozart' symphony based on fragments of old ones, fooling many non-specialists into thinking it was real Mozart.

The obvious criticisms from the above philosophical views are that:

- EMI does not create new *styles* - it mechanically re-uses existing grammars
  (it creates 'real Mozart' rather than 'real EMI')
- EMI has no notion of human emotion, other than the abstract emotional content of the grammars

Brown's 'Brute Force'

Silas Brown coded rules of classical harmony as heuristics[5] which rejected most of billions of randomly generated note sequences by back-tracking. This knowledge-based system produced results which could probably just pass an A-Level harmony exam. Aside from its gratuitous use of processor time, the main objections to this method are the same as for EMI – with 'the rules of harmony' taken to mean 'the grammar created over many years by human genius composers'.

Holtzman's GCDL

Not an automated composition program, but a tool to aid human composers in writing structured compositions, GCDL was described in Holzman's PhD Thesis[6] (and later, his book, *Digital Mantras*[7]). GCDL is a programming language which allows the user to specify:

- a non-deterministic type-2 grammar for the composition
- functions for selecting productions from the grammar

Much of the SARAH language used in this project is based on GCDL, and Holtzman's Thesis will be referred to throughout the design of this project.

Grammidity

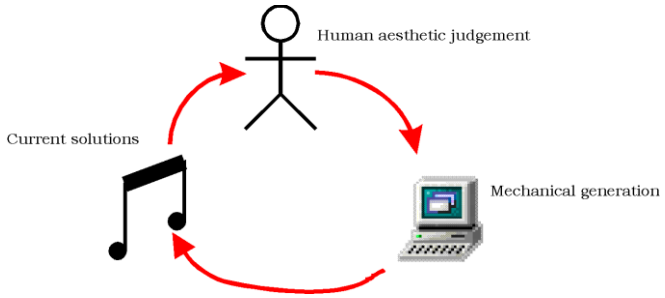
This is a web-based *evolutionary* composer[10], which turns short strings into music. All strings are legal, so any kind of mutation and crossover is permitted. Each user hears variations of a piece then selects the best one, which is used as the parent of the next generation. However, due to the randomness of the string mutations, almost all the pieces consist of just a few unstructured bleeps, and evolution is painfully slow as the human is the only fitness function. However, it does satisfy our philosophical requirement of having human expression – since every human that votes is using their own aesthetic judgement to guide the creation of the final piece.

2 Preparation

2.1 Requirements analysis

Our aim is to build a system for semi-automated composition of short pieces of music that are *meaningful* and *aesthetically pleasing*. We have seen that to achieve this, we must provide a method of generating music structurally – and that re-using *existing* grammars is cheating to some extent. Furthermore, some injection of human emotion and aesthetic judgement is required.

Using the wonderfully postmodern idea of 'listener as composer', we can use a human source of creativity and aesthetic judgement, whilst leaving the actual structural composing to the machine. By allowing the listener to guide the composition process at a high level, we can tailor each composition to his taste:



Thus, an evolutionary approach is the most natural for this project – with a human as the fitness function. Bearing in mind the problems with the *Grammidity* program (evolution is too slow; almost all pieces are short uninteresting beeps), we will aim to ensure that *all* generated pieces are meaningful and aesthetically pleasing, to at least some extent, and allow the listener to choose the very best one at each generation.

Considering the 'failing' of serialist music (that it does not sound aesthetically pleasing to most listeners), we will encourage western diatonic tonality. Purists might argue that an evolutionary approach should not constrain the generation of music in this way, but it is necessary if we are to avoid the problems of *Grammidity*. (Damage should be limited if we *provide* but *discourage* facilities for other tonal systems.)

A further requirement: In addition to the automated composition, I would like the program to be of use purely as a human composition tool, similar to Holzman's GCDL. This seems like a simple extra requirement to implement (it just requires that our grammars be editable and human readable) and has many benefits: it will allow the creation and performance of wonderful new structural music (this requires the editing facilities to be simple enough for a composer, so HCI is an issue here); it will allow initial pieces to be written from which to begin the evolution process; and will be useful for debugging the evolution methods. By integrating the editing and evolving aspects, the power of each in increased – human composers waiting for the muse can try a few mutations of their piece to get them started; whilst evolver listeners faced with an almost perfect piece can hand-edit the final touches.

A problem with human fitness functions is the large amount of human time needed to produce good results. We can greatly improve our results by providing a web interface to the system, allowing multiple users to contribute to the evolution process. This is specified as an optional requirement.

The requirements of the program are therefore:

- provide a system of evolving grammars which generate compositions
- the grammars must be human readable and editable
- compositions must sound aesthetically pleasing, at least to a limited extent

- the concept of tonality will be built in
- editing must be simple and powerful enough for a composer to use
- provide a web interface to the evolution system (optional)

2.2 Grammars and L-systems

To meet the requirement that the grammar itself must be capable of evolving, I investigated various systems of grammars and production rules. My starting point here was Holtzman's GCDL Thesis.

2.2.1 The GCDL system of Grammars and Functions

As mentioned earlier, GCDL is a composition aid which allows the user to define a non-deterministic type-2 grammar together with a set of functions, which describe which production to apply at each stage of generation. If no function for a rule is specified for a re-write, then a random selection is made. Here is a sample of GCDL, for generating Shöenberg-like canons:

```
"SERIES -> <SELECT-NOTE> obj1 . obj2 . obj3 . obj4 . obj5 … obj12"
[COMPOSITION -> CANON ]
[CANON -> voice1, STRUCTURE, voice2, STRUCTURE]
[STRUCTURE -> VERSION-TYPE(SERIES)]
[VERSION-TYPE (_) -> @I(_) . @B(_), #6#(_)]

function SELECT-NOTE
        n = n+1;
        if n <= 6 then result = 0; finish;
        else result = n; finish;
end
```

The symbols `obj1` etc. refer to actual, sounding notes. The transformations of inversion and retrograde (discussed earlier) are represented by `@I` and `@B` respectively. Note that in `STRUCTURE`, `SERIES` is passed as an argument to `VERSION-TYPE`.

The dots (.) indicate that a choice of RHSs is to be made. In `SERIES`, the function `SELECT-NOTE` is called to determine which choice to make, whilst in `VERSION-TYPE` the choice is left to chance.

The problem with using GCDL for our evolutionary approach is that while mutating the grammar may be reasonably straight forward, it would be very hard to mutate the functions and still have them make sense every time. We want each piece to be deterministic, so leaving each choice to chance is not a good plan either.

2.2.2 L-systems

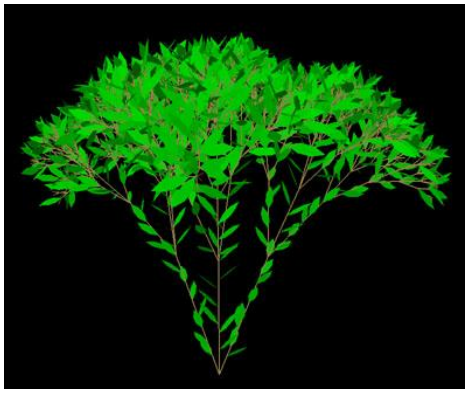
L-Systems were discovered by Lindenmayer[11] and are (at their simplest) deterministic type-2 grammars with transformations. They consist of an initial symbol, and a series of rules for replacing individual symbols with new sets of symbols (which may have transforms applied to them). At each stage in generation, each symbol in the current list is replaced using its rewrite rule (if one exists). When no more rewrites can be performed, generation is finished. For example, applying the following L-system to the symbol A: (where `(t`*n*`)` are transforms)

```
A -> (t1)B,C,(t2)B,C
B -> (t2)c, (t3)c
C -> (t4)D
```

Gives: (with each stage of rewriting on a new line)

```
A
(t1)B, C, (t2)B, C
(t1)(t2)C, (t1)(t3)C, C, (t2)(t2)C, (t2)(t3), C, C
(t1)(t2)(t4)D,(t1)(t3)(t4)D,(t4)D,(t2)(t2)(t4)D,(t2)(t3)(t4)D,(t4)D,D,(t4)D
```

A common demonstration[11] of L-systems is to generate simulations of plants, by specifying rules for how the stems and leaves are constructed.



Some simulations allow plants to compete for rays of sunlight, with successful ones reproducing - and evolving by making small changes to their childrens' L-system grammar. I claim that an analogous process can be used to evolve music – by picturing plants as Schenkerian structure; whose leaves represents notes; and whose 3D space represents musical time and pitch. The beauty of L-systems is that they are deterministic, so have no need of complex function languages for generation (as in GCDL). Hence evolution is much easier.

2.3 Genetic Algorithms

GAs are a form of hill-climbing method for finding near-optimal solutions to problems. The GA method is based on Darwin's theory of evolution, which he described as having "…no limit to this power of slowly and beautifully adapting each form to the most complex relations of life…". A full description of GA theory can be found in [14] (and a simple introduction in [15]), but the basic components are:

- A mapping from lists of symbols (the *genotype* or *genome*) to a form (the *phenotype*)
- An *initial population* of genomes
- A *fitness function* which ranks phenotypes
- A set of *operators* which generate new genomes from a set of selected genomes

Most previous work in musical GAs has focussed on solving harmonization problems or generating melodies over existing chords, rather than generating whole compositions. This is because it is possible to mechanically quantify their 'correctness' with reference to the rules of harmony. Noteable examples include McIntyre's generation of Baroque harmonies[16] and Biles' Jazz solo generator [17]. Biles, like *Grammidity*, and as we intend to, uses a human as fitness function rather than mechanical heuristics. A detailed review of musical GAs can be found in [18].

Most music GAs use human-illegible strings for the genome. This has the advantage that *all* strings can be mapped to a musical form, which greatly simplifies the operators. Typical operators are *crossover* and *mutation*. Crossover creates a new genome from two or more parents by splicing randomly selected chunks of their genomes, *eg.*:

```
Parent 1:       AJFGIRHJ83JDKS34KDS
Parent 2:       JHKUI58SJC8C32KS9CK
Child:          AJFGIR8SJC8C3234KDS
```

Mutation operates on a single genome, by making small random changes to it. (GAs using mutation alone are analogous to hill-climbing algorithms). *eg.*:

```
Parent 1:       AJFGIRHJ83JDKS34KDS
Child:          BJFGIRIJ82JDKS34LDS
```

Dawkins[15] points out that crossover alone cannot find optimal solutions as it is a *subtractive* operator: it removes bad genetic material, but never *creates* anything new (unless new genomes are brought into the pool). Hence after several generations without mutation the genome material will *stagnate*, (i.e. all genomes will be almost identical). Mutation alone will converge to local optima, but suffers from getting trapped there. Hence a combination of mutation and crossover is a good solution method (and natural evolution is proof of its efficiency).

A simple way to use GAs with L-systems to generate our compositions would be to invent an injective function from strings to sets of L-system productions. The user would then select her favorite phenotypes, whose genomes would then be subjected to crossover and mutation to produce the next generation. Unfortunately, there are two problems with this simple approach:

- there may be no guarantee that small changes in the genome would produce equally small changes in the composition – they could completely alter the resulting L-system.
- allowing *all* possible strings to have meaning would conflict directly with our requirement that our compositions be easily readable and editable by a non-technical user.

A better solution is to use the L-system itself as the genome. This will admittedly require operators more complex than simple string mutation and crossover, but will ensure that small genome changes mean small composition changes; and will keep the genome readable.

An elegant implementation for crossover is to treat each production rule in an L-system as a *gene*. A complete L-system forms the *genome*. Crossover between two genomes is now a matter of splicing their genes. In this way, the child of a Bach Fugue and a Spice Girls song could inherit the small-scale melodies of the former and large scale structure of the latter (the curly and square brackets mean 'play in series' and 'play in parallel'):

```
Parent 1:   SPICE-SONG -> { VERSE, CHORUS, VERSE, CHORUS, MIDDLE-8, CHORUS }
            VERSE -> { RIFF-A, RIFF-B, RIFF-C, RIFF-B }
            CHORUS -> { RIFF-A, RIFF-A }
            MIDDLE-8 -> { sequence of notes 1}
            RIFF-A -> { sequence of notes 2 }
            …

Parent 2:   BACH-FUGUE -> [ THEME, DELAYED ]
            DELAYED -> { 2-BAR-PAUSE, COUNTER-MELODY }
            THEME -> { sequence of notes 3 }
            COUNTER-MELODY -> { sequence of notes 4}
            2-BAR-PAUSE -> { sequence of notes 5 }

Child:      SPICE-SONG -> { DELAYED, THEME, DELAYED, MIDDLE-8, THEME}
            DELAYED -> { 2-BAR-PAUSE, MIDDLE-8 }
            THEME -> { sequence of notes }
            MIDDLE-8 -> { sequence of notes 1}
            2-BAR-PAUSE -> { sequence of notes 5 }
```

Note the *renaming* – the original LHS of each gene is preserved, but the names referred to in its RHS are 'alpha-converted' to the names of the new genes occupying equivalent positions to those in the parent.

To implement mutation we simply make 'small changes' to our child, *eg*.

```
MutantChild: SPICE-SONG -> { DELAYED, THEME, THEME, DELAYED, MIDDLE-8, THEME}
             DELAYED -> { 2-BAR-PAUSE, MIDDLE-8, 2-BAR-PAUSE }
             THEME -> { sequence of notes }
             MIDDLE-8 -> { sequence of notes 1}
             2-BAR-PAUSE -> { sequence of notes 5 }
```

The effect of such mutations on the composition is analogous to trees growing extra branches, or humans growing extra fingers. Such natural mutations are not caused by whole new chunks of DNA arising which explain exactly how to grow the new parts – instead, there is a segment of DNA which describes how to grow a single finger, which is 'called' 5 times by the DNA for the hand. So the small mutation of changing the 5 to a 6 produces a whole extra finger.

Using these two musical operators thus provides us with a beautiful system for evolving our compositions – it allows pop songs to 'sprout' extra verses and choruses, or inherit a 12-bar-blues structure from a parent to replace the sonata form from other, whilst potentially still leaving the lower-level melodies in tact.

2.4 Language decisions & Specification

From the above discussion, we arrive at a system of evolving L-systems as the basic structural mechanism, and examine the possible details for the behavior of our L-system language (which I name SARAH – an acronym for *Structured Audio for Randomly Added-to Hierarchies*). The elements characterizing L-system languages are:

- the set of transforms that can be applied to its elements
- the mapping of its terminal symbols to form (in this case, musical sounds).

2.4.1 Initial Ideas

GCDL provides multiple primitives which map directly to sounds, named `obj1`, `obj2`, `obj3` *etc.* This allows a composition to be played by mapping these symbols to *any* set of sounds. (Later in his thesis, Holtzman demonstrates how symbols can even be mapped to low-level sound-waves rather than notes.) This approach is unnecessarily complicated for our project, as we are concerned only with a note-level representation. So SARAH will use a *single* primitive, named `NOTE`. The initial idea is that `NOTE` will represent a single crotchet of middle C, with all other notes formed by applying transforms to this primitive.

The following were my initial ideas for SARAH's transforms. (The syntax is based on GCDL – '@' indicates that a string is a transform, rather than a symbol name.)

```
@#n(M)          :       play M transposed by n semitones
@R(M)           :       play M backwards (retrograde)
@I(M)           :       play inversion of M
@~a/b(M)        :       time-stretch M by a/b
@*v(M)          :       change instrument of M to voice v
@>n(M)          :       change amplitude by factor of n
```

The first line of 'Happy Birthday' could be written:

```
HAPPY-BIRTHDAY ->{NOTE, NOTE, @~2(@#2(NOTE)), NOTE, @#5(NOTE), @~2(@#4(NOTE))}
```

2.4.2 Scale-Degree model

However, the notion of semitonal transposition does not enforce, or even encourage, *tonality*, thus violating the specification. A beautiful Bach chorale could become mutated by transposing one voice up a semitone, which would most likely have disastrous results.

An elegant solution is to consider another interpretation of what notes *mean* in tonal music. Traditional music programs represent notes with quintuples of (Pitch, Time, Duration, Amplitude, Timbre) [20]. In the example above, we could imagine notes whose pitches are increased and decreased by the application of the `@#` transform. We can replace this representation by one incorporating the tonality concept: (Key, Degree, Time, Duration, Amplitude, Timbre). This idea leads us to the idea of removing the `@#` transform, and replacing it with:

```
@^n(M)          :       play M shifted by n Degrees of its Key
@&n(M)          :       shift the Key of M by n Degrees (of the current Key)
```

'Happy Birthday' (where the default key of NOTE is some major scale) now becomes: (Note the change of key on the last note)

```
BIRTHDAY -> {@^5(NOTE), @^5(NOTE), @~2(@^6(NOTE)), @^5(NOTE),
                @^8(NOTE), @~2(@&5(@^3NOTE))}
```

The beauty of this method is that mutations may now alter the *degree* of notes within their existing scale, rather than their absolute pitches. This means that mutated versions of tunes will (usually) still sound harmonically correct. Furthermore, it allows the key of passages to be changed easily: the following plays BIRTHDAY in a series of different keys:

```
BIRTHDAY-BLUES -> { BIRTHDAY, BIRTHDAY, BRITHDAY, BIRTHDAY,
                @&4(BIRTHDAY), @&4(BIRTHDAY), BIRTHDAY, BIRTHDAY,
                @&5(BIRTHDAY), @&4(BIRTHDAY), BIRTHDAY, @&5(BIRTHDAY) }
```

This system limits us to working only with major scales – but defining a Key as the tuple (Root-note, Scale-pattern) removes this limitation. The meaning of the @& transform is now to shift the Root-note, and we can invent new transforms that alter the scale pattern. I propose the transform @$*n* to mean 'change mode to that of the *n*th degree of the current scale. For example, @$5(M) where M is originally in C major, would play M in the Key of C Aoelian. (The reader unfamiliar with scale and mode theory can find an introduction in [21]). The three transforms of @^, @& and @$ are sufficient to describe simple tonality, and can all be applied by mutation to produce meaningful results. For example, a piece containing BIRTHDAY could easily mutate so that it an instance of it is played in a minor key:

```
Parent:  PIECE -> { BIRTHDAY, BIRTHDAY, BITHDAY }
         BIRTHDAY -> {@^5(NOTE), @^5(NOTE), @~2(@^6(NOTE)), @^5(NOTE),
                         @^8(NOTE), @~2(@&5(@^3NOTE))}

Child:   PIECE -> { BIRTHDAY, @$5(BIRTHDAY), BITHDAY }
         BIRTHDAY -> {@^5(NOTE), @^5(NOTE), @~2(@^6(NOTE)), @^5(NOTE),
                         @^8(NOTE), @~2(@&5(@^3NOTE))}
```

For music in non-diatonic scales, we could introduce special transforms that change the scale pattern in more specialized ways. This will probably not be necessary in this project, although the system design should be flexible enough to allow it to be added later if desired.

2.4.3 Incremental changes, not absolutes

Note that all of these transforms are *incremental* - when a sequence of transforms is applied to a phrase, the application of each of them has an effect on the final result. Had, for example, the key-change transform been defined in an *absolute* sense, such as:

```
@&n(M)          :       Change key to absolute key n
```

(for example, key 5 could represent the key of G, so @&5(M) would always play M in that particular key), then we would face the problem of *redundant* transforms: transforms added by mutation or the user may have no effect on the result. But they may build up unnoticed, and if the absolute transform is ever removed, their effects will suddenly appear. This goes against evolution's method of incremental improvement. Here is an example of such a disaster:

```
Original:        NICE-TUNE -> [ @&5(BIRTHDAY), @&5(HARMONY) ]
Child:           NICE-TUNE -> [ @&7(@&5(BIRTHDAY)), @&5(HARMONY) ]
GrandChild:      NICE-TUNE -> [ @&3(@&7(@&5(BIRTHDAY))), @&5(HARMONY) ]
GreatGrandChild: NICE-TUNE -> [ @&3(@&7(BIRTHDAY)), @&5(HARMONY) ]
```

Here the child and grandchild sound identical to the original, despite the 'bad DNA' that is allowed to build up. Only when the @&5 transform is removed are the changes made noticeable.

If the above example is used with the previous incremental definition of the key-shift transform then the problem is solved, since each successive transform shifts by an additional amount. So the root of the key of the `BIRTHDAY` in Child would be shifted by 5+7; in GrandChild by 5+7+3; and in GreatGrandChild by 7+3.

The problem of absolute transforms is, however, still present in the instrument selection transform, `@*`. A solution to this is to order the available voices in some meaningful fashion, and use relative changes in the instrument number. For example, we could order classical instruments in score order:

1. Flute
2. Clarinet
3. Oboe
4. Violin
5. Viola
6. Cello
7. Trumpet
8. Horn
9. Trombone
10. Double bass

And define a the voiceshift transform `@vn(M)` to shift the voice of `M` by *n* places in the list. This allows hill-climbing to happen in a reasonably sensible way, as new voice shifts introduced by mutation will now only create a small movement within the voice list, rather than immediately changing the voice to a completely different absolute. For example, using absolute voice set:

```
Parent:  DUET -> [ @*4(TUNE), @*8(TUNE) ]
Child:   DUET -> [ @*4(@*1(TUNE)), @*8(TUNE) ]
```

The original Duet was for Violin and Horn – in the mutation, the Violin-ness has been completely lost and replaced by a flute. If we use incremental transforms instead:

```
Parent:  DUET -> [ @v4(TUNE), @v8(TUNE) ]
Child:   DUET -> [ @v4(@v1(TUNE)), @v8(TUNE) ]
```

Now the effects of both shifts are felt, and the original Violin voice is shifted again to become a Viola – a suitably small change.

2.4.4 Nested Transforms

GCDL allows users to use nesting, *eg.*

```
[MELODY->A,B,@R(A,B,@I(A, B))]
```

Such a syntax, if used with L-systems, allows very complex structures to be defined in a single gene. There is a danger here that compositions could thus be created with a small number of highly complex genes. This would mean that the changes occurring from crossover would be very large indeed, and this would spoil our small-changes approach.

There appear to be two other possibilities for SARAH syntax schemes with regard to nesting restrictions:

The first, and most extreme, option is to define three types of gene: melody, harmony, and transform. In this scheme, the melody and harmony genes are not allowed to include transforms at all, and consist only of a list of phrases in square or curly brackets. The transform type is only allowed one phrase, but may have a series of transforms applied to it. This has the effect of forcing compositions to use the maximum number of genes – any syntax that would be nested in GCDL is now split up across different genes. The above becomes:

```
MELODY -> { A,B,NEW-GENE1 }        : melody gene
```

```
NEW-GENE1 -> @R(NEW-GENE2)          : transform gene
NEW-GENE2 -> { A,B, NEW-GENE3 }     : melody gene
NEW-GENE3 -> @I(NEW-GENE4)          : transform gene
NEW-GENE4 -> { A, B }               : melody gene
```

(an even more extreme option would be to allow only a single transform to be applied in each transform gene. So where multiple transforms on a single phrase are required, a new transform gene would be created for each one)

While this 'pure' method may well produce smoother evolution, the simple example shown above shows that it would make composing very time consuming and difficult for a human editor, due to the amount of constructions required for even this very simple melody. We can relax the restrictions somewhat by allowing transforms to be used inside melody and harmony genes, but only on single phrases. `MELODY` now becomes:

```
MELODY -> { A,B,@R(NEW-GENE1) }
NEW-GENE1 -> { A,B,@I(NEW-GENE2) }
NEW-GENE2 -> { A,B }
```

This compromise appears to provide the best trade-off between cross-over smoothness and human edibility.

2.4.5 Argument passing vs. 'Copy-mutate'

Argument passing

Passing arguments to genes would be very useful in a musical L-system, as it would allow us to describe generalised patterns, then specialise to produce variations. The first prelude from Bach's *Well-Tempered Clavier* [22] consists entirely of a single arpeggio pattern, which has various chords 'plugged in' to it, and could be described as:

```
PRELUDE  -> { ARPEGGIO(C,E,G,C',E'), ARPEGGIO(D,F,A,D',F'), ARPEGGIO(G,D,G'D'F'), …
ARPEGGIO(a,b,c,d,e,f) -> { a,b,c,d,e,d,e,f,a,b,c,d,e,d,e,f }
```

GCDL uses argument passing. There is, however, a major problem with using it in SARAH: conflict with evolution. Should genes be allowed to grow extra argument slots? Should genes which call other genes be allowed to grow extra arguments? It appears that these conditions would be necessary – but what would happen when the wrong number of arguments ended up being passed as a result?

After some thought, it seems SARAH argument passing is feasible, but would lead to some very tricky problems. Whilst admittedly losing some power from the language, it is simpler to use the following alternative:

Copymutate

So far, we have thought of mutations as small changes to existing genes. *eg.*

```
Parent:  SONG -> { VERSE, VERSE, VERSE }
         VERSE -> { RIFFA, RIFFB, RIFFA, RIFFB }

Child1:  SONG -> { VERSE, VERSE, VERSE }
         VERSE -> { RIFFA, RIFFB, RIFFA, RIFFA, RIFFB }
```

This change affects *all* of the verses in the song. But what if we want only a single verse to be changed? With argument-passing, we could have specialised the required `VERSE`.

Copymutate is allows a similar effect to occur without using argument passing, in which a new gene is created in the genome. It is a copy of the gene that we wish to specialise. It is then mutated to create the specialisation. Then

the genome is searched for references to the old gene, and some of them are randomly changed to refer to the new gene instead:

```
Child2:  SONG -> { VERSE, VERSE-COPY, VERSE }
         VERSE -> { RIFFA, RIFFB, RIFFA, RIFFB }
         VERSE-COPY -> { RIFFA, RIFFB, RIFFA, RIFFA, RIFFB }
```

Admittedly, we are throwing away the important information that `VERSE` and `VERSE-COPY` were originally related – there is now no way that new transforms can intentionally affect them both. However, the benefits gained by avoiding argument-passing problems appear to outweigh this.

2.4.6 Final Language Specification

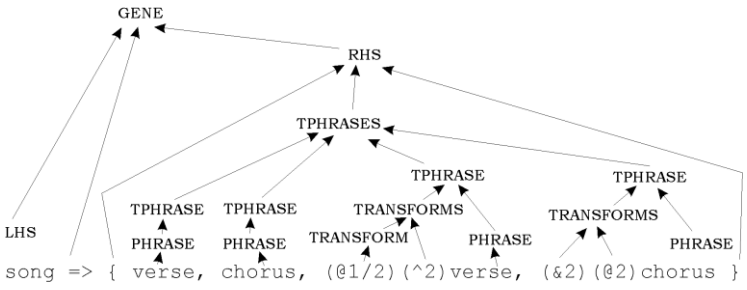
The SARAH syntax presented below has been modified from that used above. The original GCDL syntax was designed to accommodate heavy nesting such as `@R(A, @I(A,B, @R(A)))`. Now that we have prohibited this and restricted transforms to being applied to single phrases only, we can simplify the syntax by writing *transforms* in brackets, rather than their 'transformees'. Hence we can drop the '`@`' symbol, as the presence of brackets is enough to determine which symbols are transforms and which are phrases. This gives the following lambda-style syntax:

```
(^n)M          :      play M shifted by n Degrees of its Key
(&n)M          :      shift the Key of M by n Degrees (of the current Key)
(R)M           :      play M backwards (retrograde)
(I)M           :      play inversion of M
(~a/b)M        :      time-stretch M by a/b
(vn)M          :      shift instrument of M by n positions in voice list
(@a/b)M        :      change amplitude by factor of a/b
```

Each gene is written as a production, with the RHS in curly or square brackets to indicate melody (playing in series) or harmony (playing in parallel). *Phrase* means 'a reference to another gene'. A transformed phrase, or *TPhrase*, means 'a phrase with any number of transforms applied to it'. Inside the brackets of an RHS, then, can be any number of *TPhrase*s, separated by commas. (Note that SARAH uses '=>' rather than '->' in production rules, to avoid confusion with the '->' symbol used in defining SARAH itself!) Here is the formal grammar:

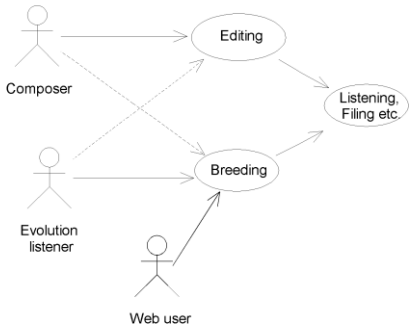
GENOME	-> GENE | GENE, GENOME
GENE	-> LHS => RHS
LHS	-> *String* (the name of the gene)
RHS	-> { TPHRASELISE } | [TPHRASELIST]
TPHRASELIST	-> TPHRASE | THPRASE, TPHRASELIST
TPHRASE	-> TRANSFORMS PHRASE | PHRASE
TRANSFORMS	-> TRANSFORM | TRANSFORM TRANSFORMS
PHRASE	-> *String* (the name of the referent)
TRANSFORM	-> (*String*) (where the string is of the form of one of the above transforms)

Below is an example gene structure:



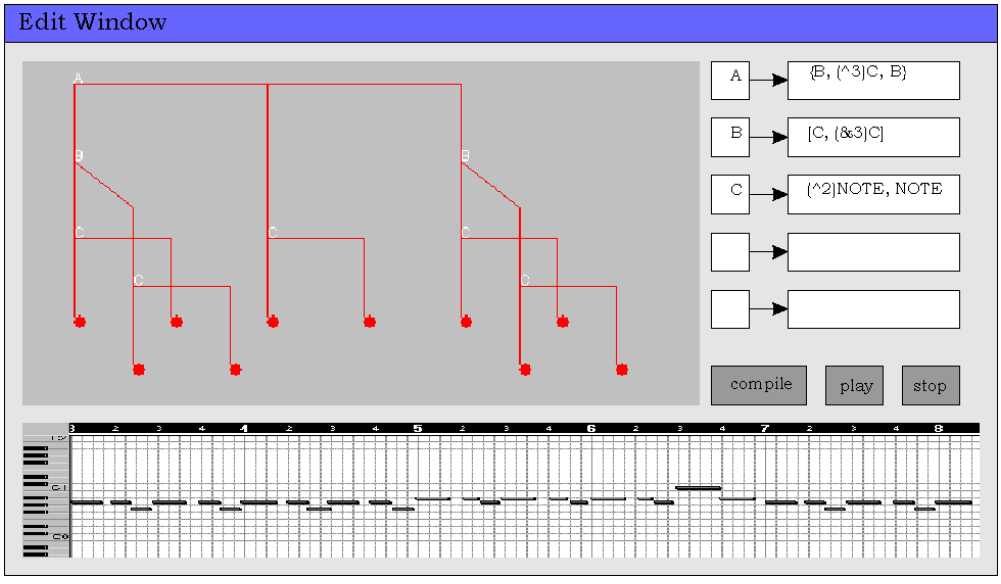
2.5 User interface Specification

There are two principal user-cases for the program: as composing *aid*, and as human-guided evolutionary *composer*. We also require the two uses to overlap so the two users may use each other's features. (In addition, we specified an optional evolution-only web interface):



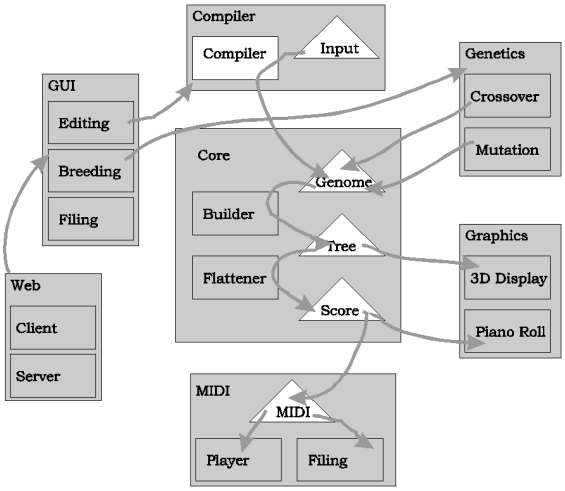
The GUI should thus provide two environments: an Editing window and an evolution ('Manager') window. The Manager should display a simple representation of the current generation and allow listening, breeding, and the opening of editing windows.

Below is the Edit Window initial mockup. Note the series of boxes provided to simplify SARAH coding. Displays show the compiled tree structure and 'piano roll' score:



2.6 Modules, schedule and strategy

From the above discussions emerges the following diagram of the main modules comprising the project. ('Module' refers not to particular implementations, but to abstract functionality areas). Full module requirements ans schedules are given in the Proposal [Appendix E]. Since the modules are mostly self-contained units, the *multiple waterfall* design model is most suitable: each module in sequence will be completely specified, designed, implemented and tested before moving on to the next module:



2.7 Technologies

Java

Java was chosen as the main language - it is elegant, object-oriented, and includes simple GUI and networking capabilities.

MIDI

We require the program to play music, and to save as MIDI – the industry standard music format. 'MIDI' has two related meanings: a file format; and a 'wire protocol' for sending real-time bits to sound generation devices. Specifications can be found in [24].

Java code could be written to create MIDI files and generate wire-protocol streams. (Sending streams to soundcards would also require *native* code). This approach is rather tedious.

A better plan was to investigate libraries. Sadly Java 1.2 had no MIDI support, so using external libraries would involve un-portable native code. Java 1.3 Beta featured a MIDI package, but I found many bugs. A stable version of this package was available in the *Java Media Framework*, though sadly only for Windows. This was the best option.

No tutorial was available for the package. Exposing bugs in the 1.3 version and learning the JMF version was thus a process of decoding the API, reading newsgroups and chat room transcripts, and occasionally emailing its designers.

Network Programming

We require a client-server system to allow multiple breeders. The client should run as a web applet for ease of use.

Sockets and RMI provide pure-Java solutions, whilst CORBA gives inter-language communication. Alternatively, HTML forms could be used to post to a CGI server scripts which could communicate with the Java program. Full descriptions are found in [24]. RMI is the elegant solution – we create a client applet connected to methods in the main program which runs as a server.

Development Environment

Free student versions of Borland's *JBuilder* and Microsoft's *J++* were available. I chose the latter since I had used *Visual C++* briefly, making J++ reasonably familiar. A student version of *Rational Rose* was available for modeling the classes.

Previous Experience

Technology	**Experience**
Java	Tripos courses and IB Project[23] only
MIDI formats, protocols and programming	None
Network programming	None
J++	None (but 5 weeks paid work with Visual C++)
Rational Rose and Java modelling	None

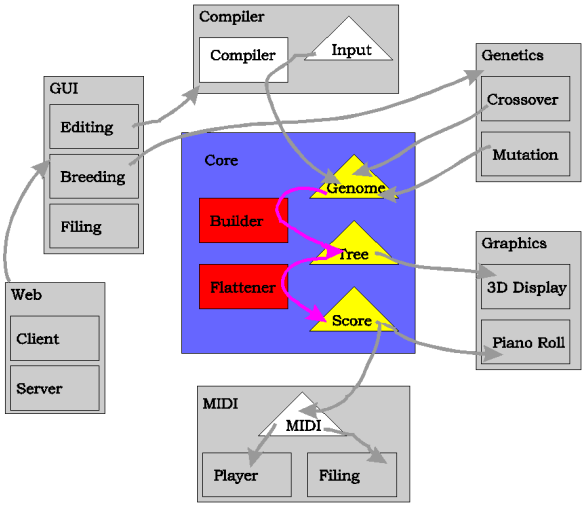
3 Implementation

This chapter describes how each module was implemented, following the chronological order of the schedule. The high-level functionality and interactions of modules were specified in advance. In sequence, the structure of each module is designed, coded, and rigorously tested before moving to the next module.

The program provides computer-aided composition facilities together with an almost magical evolution system using genes, so was named with the appalling pun, *Music Genie*.

Note that the UML diagrams presented here show greatly simplified versions of the real Java objects – their purpose is to illustrate the high-level functionality of classes rather than their entire internal workings.

3.1 Core Module



The purpose of the Core is to provide fundamental data structures for *MusicGenie*, and methods for constructing instances of the Tree and Score structures from a given Genome. These was implemented by three main complex classes corresponding to the three data structures. Here we examine these classes and their sub-components.

3.1.1 Notes and Scores

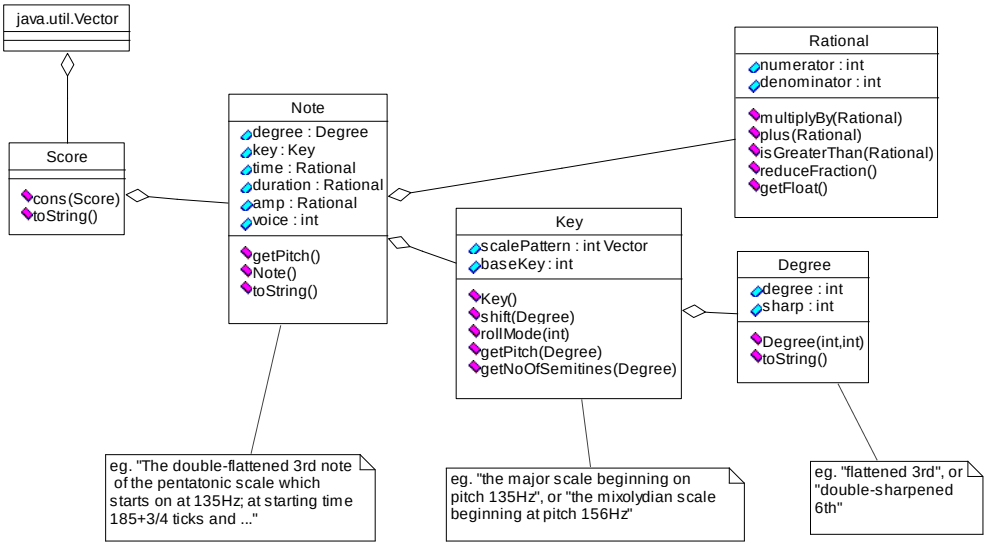
As discussed previously, Harris *et al.* [20] defined the information contained in a music note by the quintuple:

(Pitch:int, Time:Rational, Duration:Rational, Amplitude:Rational, Timbre:Instrument),

…but *MusicGenie* uses the concept of *tonality* to define notes by the sextuple:

(Degree, Key, Time, Duration, Amplitude, Timbre).

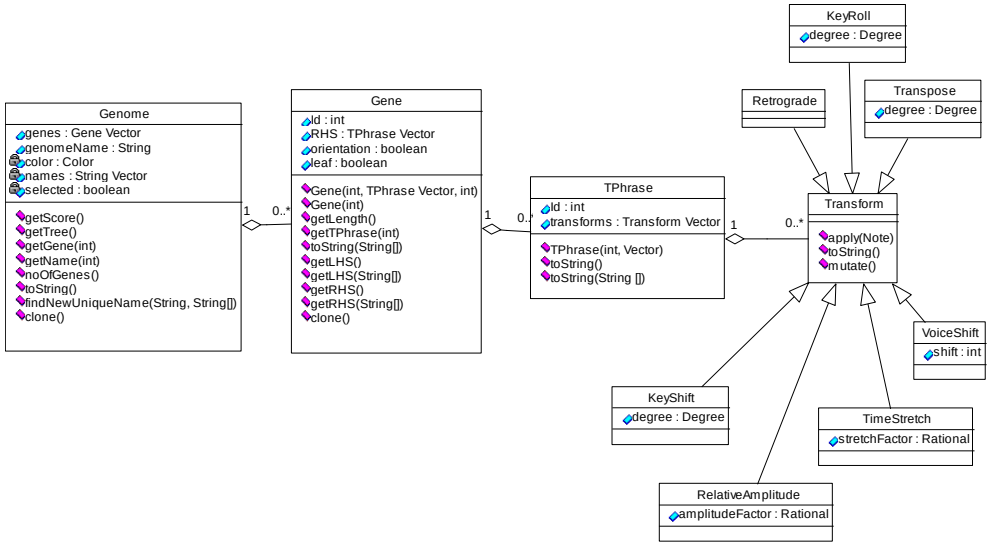
The implementation of this is shown below (Note that the Score is simply a Vector of Notes):



`Key` contains methods anticipating the key-shift and mode-shift transforms. The time-stretch, transpose, and relative-amplitude transforms can be easily applied by modifying the public attributes of `Note`. Such transforms will involve arithmetic operations on rationals, which are provided for in the `Rational` class. Euler's algorithm is implemented in `Rational.reduceFractions()` and called after each operation to ensure that sensible representations (eg. 3/4 rather than 15/20) are always in use.

3.1.2 The Genome

Recall the gene-parsing diagram from the previous chapter. A gene is named by a LHS *identifier*. Its RHS describes a {melody} or [harmony] orientation, and a list of Transformed Phrases (*TPhrases*). Alternatively, the RHS may describe a primitive `NOTE`. This state of the RHS is implemented by a boolean, named *leaf* (for reasons that will become apparent later). If the RHS is not a `NOTE` then a Vector is used to list the `TPhrase`s. A `TPhrase` such as `(^3)(&2)VERSE` is then described by the int identifier for `VERSE`, and a vector containing `Transform` objects representing `(^3)` and `(&2)` in the obvious ways.



Gene identifiers

An important feature of this design is that ints, rather than the original string names, are used as gene identifiers. The mapping from int IDs to string names is stored as a single array of names in the `Genome` object. The reason for

this is that, looking ahead to the crossover algorithms, it will allow genomes to be crossed by simply cutting and pasting genes between genomes. If string identifiers were used, then we would have solve difficult renaming problems. To illustrate this, compare and contrast the following crossover attempts:

Using string IDs:

```
Parent1: SONG    => {VERSE,CHORUS,VERSE,CHORUS}
         VERSE   => {RIFF, RIFF}
         CHORUS  => {RIFF2}  etc.

Parent2: RONDO   => {A,B,A,C,A}
         A       => {TUNE1}
         B=>     => {TUNE2} etc.

Child:   SONG    => {VERSE,CHORUS,VERSE,CHORUS}
         A       => {TUNE1}
         CHORUS  => {RIFF2} etc.
```

Using int IDs:

```
Parent1: 1       => {2,3,2,3}
         2       => {4, 4}
         3       => {5}  etc.

Parent2: 1       => {2,3,2,4,2}
         2       => {5}
         3       => {6} etc.

Child:   1       => {2,3,2,3}
         2       => {5}
         3       => {5} etc.
```

The child genome in the String example is illegal as it now contains references to names of genes which are no longer defined, such as `VERSE`. Using int IDs this problem does not occur, as we know that genes in any two genomes will always contain the same set of int IDs. In this case, `VERSE` is represented by `2`, and the child genome is legal. ('Pretty printing' is then achieved by passing the array of names to the gene, hence the `Gene.toString(String[])` method.)

When we cross two genomes, then, we can cut and paste the genes in this fashion, then separately cut and paste the array of names so as to preserve the LHS of each inherited gene. Eg:

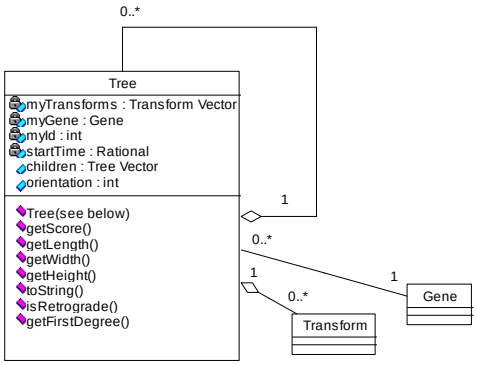
Parent1 names:	(SONG, VERSE, CHORUS, RIFF, RIFF2 …)
Parent2 names:	(RONDO, A, B, C, TUNE1, TUNE2 …)
Child names:	(SONG, A, CHORUS, …)

This does, however, leave the problem of naming conflicts: what if 'A' in Parent2 had been called 'CHORUS'? Then the child would have two different genes with the same name. To solve this potential problem, the method `Genome.findUniqueName(String, String[])` is provided. This method takes a gene name and an array of current gene names. If the name is not already used in the list, it returns the name unchanged. If it is already in use, it appends numbers to the name so as to make it unique. So it would change the second instance of CHORUS to something like CHORUS2. The provision of this method will simplify the mutation methods to be added later.

3.1.3 The Tree

The `Tree` structure is the most complex of the three classes, as it does most of the work. It provides a constructor which builds the `Tree` from a given `Genome`; and a `getScore()` method which builds a `Score` from a `Tree`. It is

defined recursively, of course – a `Tree` essentially consists of a list of child `Tree`s. Terminal, or *leaf*, nodes are indicated by the absence of any children in the `children` Vector:



In addition to its children, a `Tree` stores the list of current `Transform`s which are in place at its node; (a reference to) the gene that it its genotype; its {melody} or [harmony] orientation; and the time at which it appears in the piece.

Building Trees from Genomes

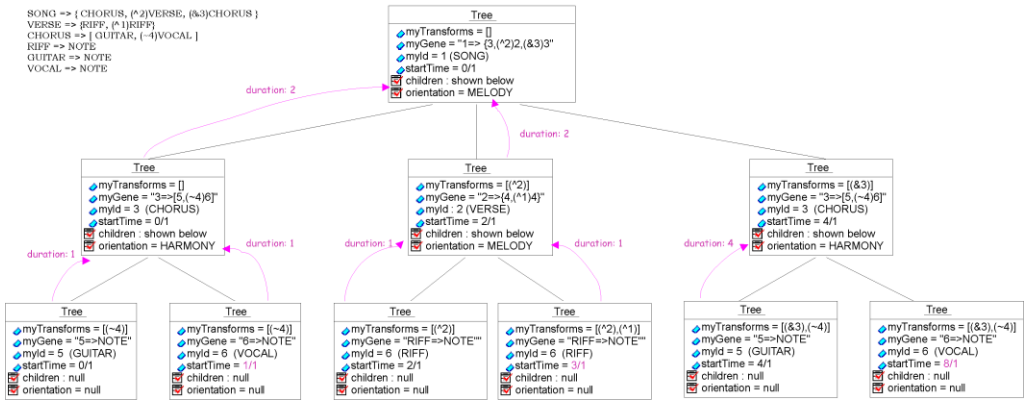
The constructor function takes a `Genome`, along with information about any transforms that are being applied or have already been applied, and the number of the gene whose tree we wish to build:

```
public Tree(int myId,
        Vector myInheritedTransforms,
        Vector myNewTransforms,
        Genome theGenome,
        Rational startTime) throws BadGenomeException
```

Hence the first constructor for the whole tree is called as follows:

```
Tree theTree = new Tree(  1,
                          new Vector()
                          new Vector(),
                          theGenome,
                          new Rational(0,1) );
```

This diagram shows an example of this taking place for a simple genome:



Note that construction happens depth-first, to allow the `startTime` of each tree to be found with no need for backtracking.

The code for Tree construction can be found in Appendix B. (Before building the tree, checks are made for recursive gene definitions and references to undefined genes. In these events, the `BadGenomeException` is thrown.)

Non-Distributive Transforms

A feature of interest in the code is the implementation of Retrograde and Inversion, which differ from other transforms by being *non-distributive*. The others are implemented by allowing lists of them to accumulate down the tree, so each terminal node has its own list of transforms. (As in the above diagram). When we later build the notes of the piece from these terminal nodes, we apply each transform in their lists to the note in order. This, however, does not work for non-distributive transforms – their actions must instead be performed *during* the construction of the tree:

Retrograde's function is to change the *structure* of the tree, rather than the sound of individual notes. Its action thus takes place during the construction of sub-trees. The simple way to implement this is to count the number of Retrogrades in the transform list of each node just before it constructs its child vector. If there is an odd number of Retrogrades, then we simply construct the child vector backwards.

Inversion, when applied to a note sequence, leaves the first note in tact, and flips the following notes about its axis. Here is a melody and it's inversion:

Inversion is implemented just after a tree's children have been created, by two recursive functions on trees. We call `getFirstPitch()` on the first child tree to find the 'pivot' note, then call `invertOnPitch(int)` on each child.

Building Scores from Trees

As mentioned above, this is a simple matter of 'picking the apples' from the tree, and is implemented in `Tree.getScore()`. For each terminal node, we create a new Note with default values (a middle C crotchet in the key of C major with amplitude 1/2). We then set its time to the `startTime` in the node, and successively apply the list of transforms to it (Using `Transform.apply(Note)`). The notes are consed together to form a score.

3.1.4 Testing the Core

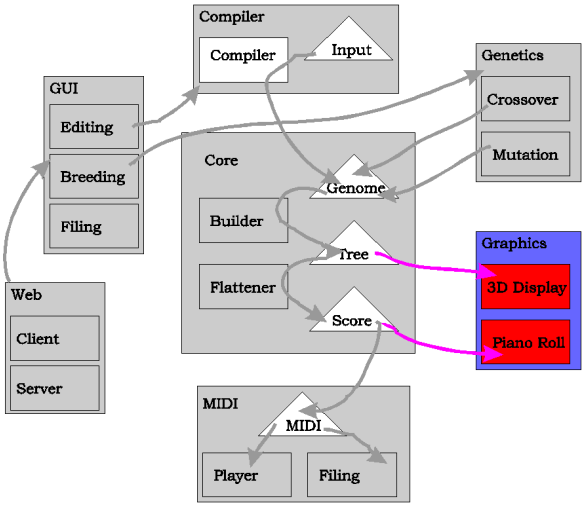
Testing and debugging were performed extensively at this point; initially by creating hard-coded instances of each class and using its `toString()` method to check that data was stored correctly. Hard-coded genomes were then input to the Tree constructor and `getScore()` methods, and again the `toString()` methods were used to test that all was functioning as intended. These genomes were chosen to include all possible features of SARAH. Attention was paid to unusual and perverse cases – attempting to break the code. Here is a sample output from tree building and apple picking (with hand-added indentation):

```
GENOME:
  A => NOTE
  B => {A, (R)(^2)A}
  C => {(^3)B, (~1/2)B}

TREE: ID 3 ori MELODY time 0
  children <
    TREE: ID 2 ori MELODY time 0
      (^3)
      children <
        TREE: LEAF
        TREE: LEAF
    END of tree 2>
    TREE: ID 2 ori MELODY time 2
      (~1/2)
      children <
        TREE: LEAF
        TREE: LEAF
    END of tree 2>
END of tree 3>

SCORE: <
  NOTE: d:2s:0   t:       d:1      a:+1/2   k: Key:0 ( 0 2 4 5 7 9 11 ),
  NOTE: d:3s:0   t:1      d:1      a:+1/2   k: Key:0 ( 0 2 4 5 7 9 11 ),
  NOTE: d:0s:0   t:2      d:+1/2   a:+1/2   k: Key:0 ( 0 2 4 5 7 9 11 ),
  NOTE: d:1s:0   t:2+1/2  d:+1/2   a:+1/2   k: Key:0 ( 0 2 4 5 7 9 11 ),
>
```

3.2 Graphics Module



As can be seen from the above results, the program's output at this stage is rather complex and difficult to verify – even for relatively small genomes. Anticipating that writing the compiler will involve much debugging, a slight schedule change is now made. Investing some time creating friendly graphical displays for Trees and Scores *before* writing the compiler will increase both efficiency and morale during its debugging.

Pre-built graphics functions are provided in the `Java2D` and `Java3D` packages. However, I had always wanted to write my own graphics package (for use with various future programs), and writing this as part of this project is a good way of killing two birds with one stone.

3.2.1 Requirements for Graphics Module

Purpose: The purpose of this module is to provide two displays - for scores and trees – to aid comprehension and understanding of the music structures as the user composes and evolves her compositions. The score is essentially two dimensional, existing in the space of time and pitch (additional 'dimensions' of timbre and amplitude can be displayed by varying the Hue and Brightness of notes). The tree, however, is best presented in three dimensions – height against time shows basic monophonic music trees, and depth can then be used to show multiple phrases played simultaneously. So we require classes to display objects in two- and three- dimensional spaces.

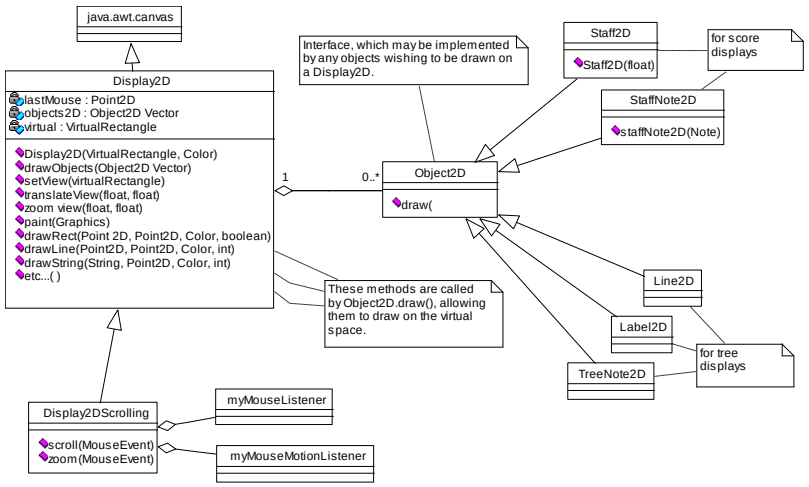
HCI: The displays should be as user-friendly as possible – specifically I would the user to move around the display using *direct* mouse movements; rather than scroll bars, magnification tools *etc* as with most other music programs. Note that no editing is performed in the displays, so we may use both mouse buttons purely for navigation.

Extension: As mentioned above, the displays should be multi-purpose components, flexible enough to accommodate new features of future applications. Specifically, new visual objects should be easy to add.

Abstraction: To allow easy creation of visual objects containing code to draw themselves, we should provide a higher-level drawing mechanism than the AWT's screen-coordinate system. Objects should be concerned only with drawing themselves in *virtual space*; whilst the Display should allow the user to navigate this virtual space by zooming, translating and rotating his view.

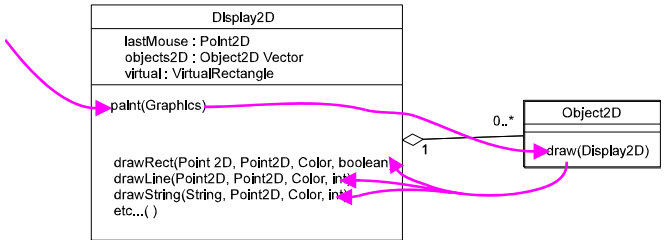
The plan is to first build the 2D display, then extend to 3D by providing methods projecting 3D objects to simple 2D objects.

3.2.2 Display2D and Score graphics

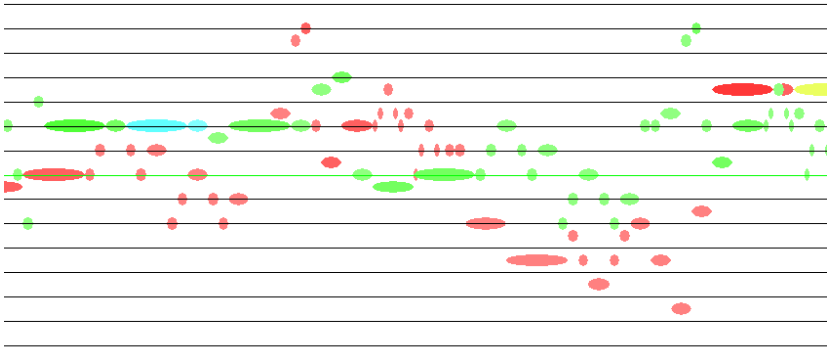


`Display2D` is a visual component, providing a 'window on virtual space' referred to here as a *view*. `Point2D` represents points in virtual space, whilst `virtualRectangle` represents the current view dimensions. Methods such as `drawRect` and `drawLine` take virtual space arguments, which are 'translated' into screen coordinates and used to call the more primitive drawing methods in `java.awt.Graphics`.

The display contains a vector of `Object2D`s currently existing in virtual space. (`Object2D` is an interface – enabling future visual objects to be easily added). `drawObjects` allows this vector to be set and updated. To draw these objects, the display calls their `draw` methods in sequence. Each object then draws itself onto the display by calling back its virtual space drawing methods:



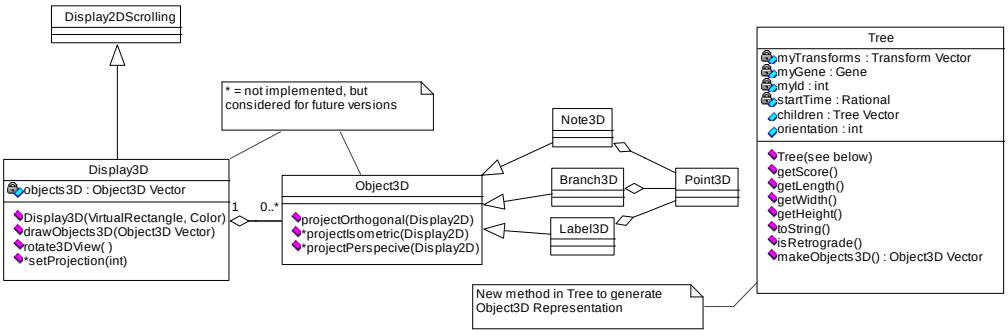
To provide a display of our `Score`s, two classes of `Object2D` are used. The `Staff2D` is a musical staff of a given length (i.e. the set of horizontal lines from the beginning to the end of the time dimension). `StaffNote2D` is the visual representation for a note, which calculates its location and color from a given `Note` object. (The other classes of `Object2D` are for use with the Tree, explained in the next section.) Here is an example of a score being displayed:



`translateView` and `zoomView` change the position and scaling. Interactivity is provided by extending `Display2D` to `Display2DScrolling`, which uses inner `mouseListener` classes to monitor mouse actions and translate them into calls to the aforementioned methods.

The user may now move around the space by dragging with the left button, and zoom (in independent *x* and *y* dimensions) by dragging with the right button. Double buffering is used to provide smooth interactivity.

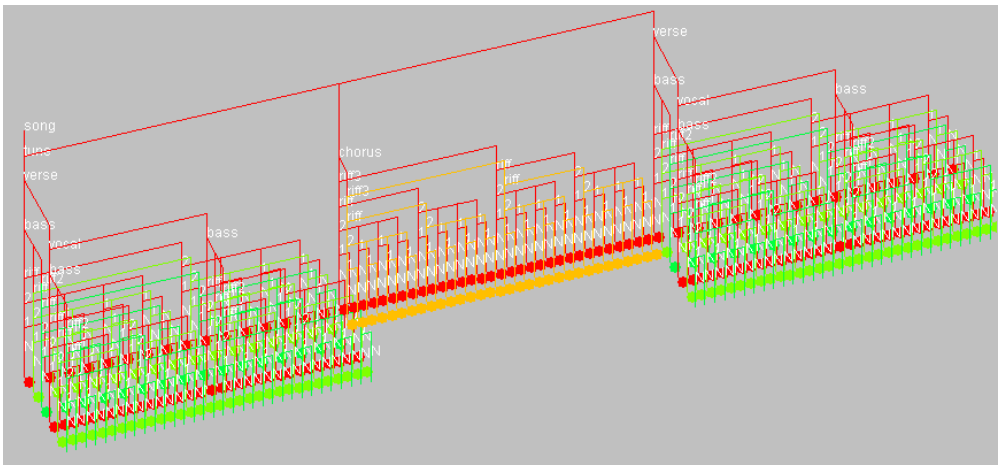
3.2.3 Display3D and Tree Graphics



`Display3D` and `Object3D` function similarly to the above 2D classes. Except that rather than asking each `Object3D` to draw itself, we ask its projection method to cons an `Object2D Vector` representation of itself onto

the `Display3D`'s `objects2D` vector (inherited from `Display2D`). The `Display3D` then functions as a `Display2D` to draw these projections. (The `Object2D` classes used for projection are `Line2D`, `TreeNote2D` and `Label2D`.)

`makeObjects3D` has been added to the Tree class, and returns a 3D representation of the tree complete with text labels on the nodes:

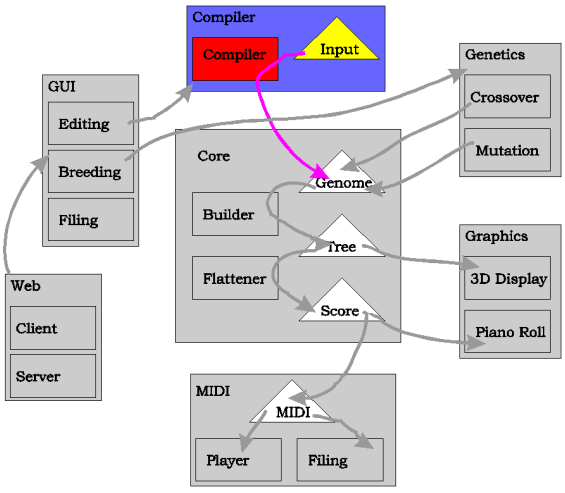


The `Display3D` listens for user mouse drags where both buttons are held down together, and rotates the tree around in 3D space. (This is in addition to the inherited `Display2D` scaling methods).

3.2.4 Testing the Graphics Module

The hard-code genomes from the Core tests were reused to test the graphics. The visual displays were matched against the previous string versions of trees and scores to ensure that the module acted as planned.

3.3 Compiler Module

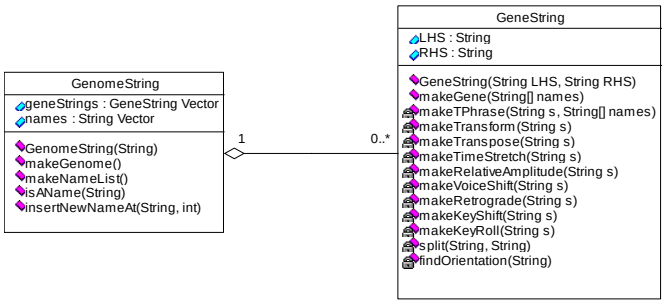


The purpose of this module is to build Genomes from SARAH strings.

3.3.1 Parsing method

The bulk of compiling work is performed by the `GeneString` class. This is constructed from two strings, corresponding to the LHS and RHS of a required gene. (eg. LHS="Song", RHS="Verse, Chorus, Verse, Chorus"). The method `makeGene` then uses these strings, along with the ordered list of names from the whole genome, and

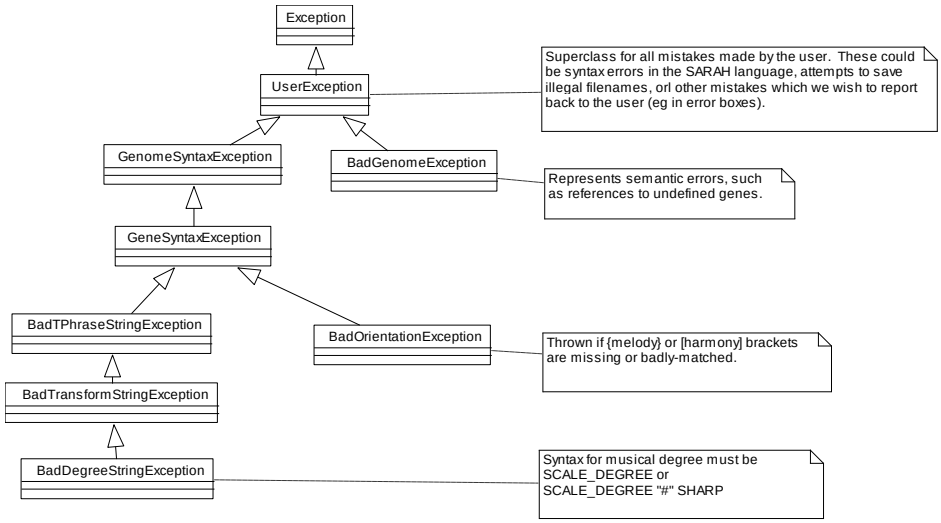
builds the required Gene object. This is achieved by parsing the RHS recursively – an RHS is made up of several `TPhrases`, which in turn are made up of a Phrase and several `Transforms`, which are made up of… etc.



Some of the code for this process is shown in Appendix C.

`GenomeString`'s constructor does the initial work of splitting a complete string of SARAH first into lines, then into LHSs and RHSs. `GenomeString.MakeGenome()` builds the array of names from the LHSs, then creates and calls `GeneString`s to build each gene. The resulting `Gene`s are consed to create the required `Genome` object.

3.3.2 Reporting syntax errors



A detailed hierarchy of Exceptions is used to report syntax errors during compilation. This allows errors in small-scale structures to have the additional types of all their higher-level parents. For example, the `BadTransformStringException` thrown from parsing the ill-formed gene `SONG=>{VERSE, (£1)VERSE}` is also a `GeneSyntaxException`, a `GenomeSyntaxException`, and a `UserException`. This hierarchy allows the error to be reported to the user as a meaningful stack:

```
Compilation failed.
Syntax error in genome:
  Syntax error in definition of  "SONG":
    Syntax error in "(£1)VERSE":
      Symbol "£" is not recognized in transform (£1).
```

3.3.3 Inverses

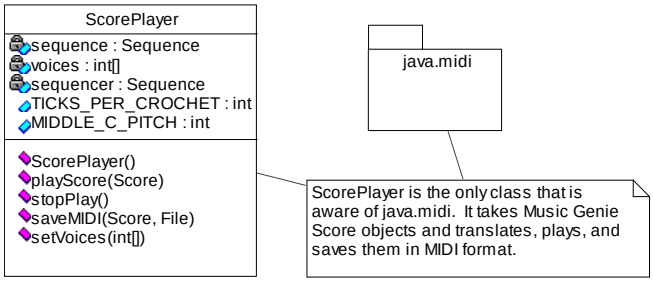
Note that the compilation process and the `Genome.toString()` method are inverses – we can now translate back and forth between `String`, `GenomeString`, and `Genome` representations of genomes. (This property will be exploited later by clever shortcuts in the mutation module.)

3.3.4 Testing the Compiler

The `toString` outputs of the previous hard-coded test genomes were input to the compiler. The `toString` outputs of the new compiled genomes were then checked against the original inputs, and found to be identical. A simple GUI was created for interactive compilation and graphical display of user-input. This allowed a rapid input/output cycle so many perverse language constructions could be tested and debugged.

3.4 MIDI Module

The purpose of this module is to allow Scores to be played through standard MIDI output, and to be saved in standard MIDI file format. This functionality was implemented by single new class:



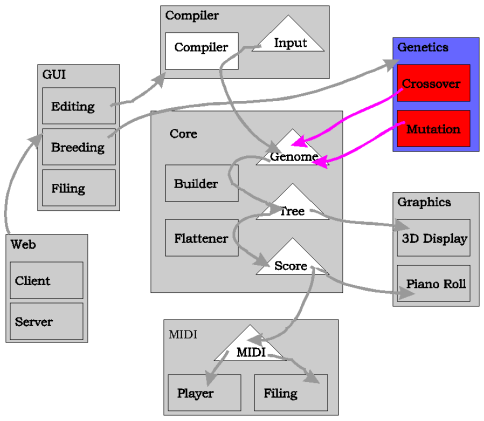
The `java.midi` package provides a low-level API to the MIDI system – only a single layer of abstraction above creating individual MIDI bytes. The main benefit is that it will (when ported to systems other than Windows) provide an OS-independent system for sending these bytes to the computer's sound card. In this typical example, a MIDI 'note-on' event is added to the Sequence:

```
ShortMessage on = new ShortMessage();
on.setMessage(144+track, pitch, velocity);          // see MIDI spec[12] for explanation of
seq.getTracks()[track].add(new MidiEvent(on, time));// the 'magic numbers' 144,128 and 192!
```

3.4.1 Testing the MIDI Module

The `ScorePlayer` was plugged in to the previously used prototype GUI to allow rapid editing, compiling and playing. This time, the *graphical* score output was compared with the *audio* output. A composer (Chris Ramshaw) was let loose on the system, briefed to break it in any way he could! These fresh eyes were also useful in picking up several obscure bugs remaining in the compiler.

3.5 Evolution Module

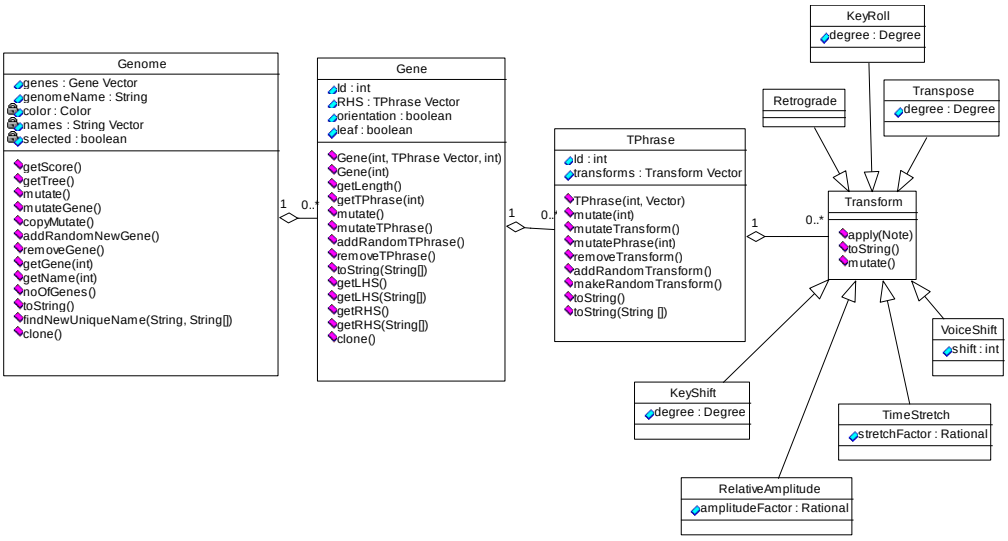


The purpose of this module is to produce new (mostly) meaningful and aesthetically pleasing genomes, from single or multiple parents. Ideally, this should enable a selective-breeding process to converge on the 'perfect'

composition. As discussed during the preparation stage, the two operators of mutation and cross-over (if applied correctly) are sufficient to enable convergence. Mutation operates on single genomes to inflict 'small changes' on them, whilst cross-over produces a child genome by combining elements from two or more parents.

3.5.1 Mutation

The code for mutation takes the form of new methods added to the existing classes in the Core:



Here we see the various kinds of mutation that can be performed at the different levels of the Genome structure:

Mutating Transforms

`Transform.mutate()` alters the argument of a single transform by a Gaussian deviation. (Recall that `Transform` is an interface, so each implementation contains its own mutation code). *eg.*

`(^5#1)` *could mutate to any of:* `(^3#1), (^4#1), (^6#1), (^5#2), (^5)`

Mutating TPhrases

`TPhrase.mutate()` randomly selects one of the four possible mutations for `TPhrases` – one of these is to mutate a randomly chosen transform using the above `Transform.mutate()`. Other possibilities include changing the Phrase (the name of the gene being transformed); removing a randomly chosen transform; or adding a completely new transform with randomly chosen argument. These four possibilities are illustrated (in order) below:

`(^5#1)(&4)(@1/2)VERSE` *could mutate to any of:*
`(^4#1)(&4)(@1/2)VERSE`
`(^5#1)(&4)(@1/2)CHORUS`
`(^5#1)(@1/2)VERSE`
`(^5#1)(&4)(&5)(@1/2)VERSE`

Mutating Genes

Similarly, but on the next level up, `Gene.mutate()` randomly selects from mutateRandomTPhrase() (which selects a random `TPhrase` then calls the above `TPhrase.mutate()`); `addRandomNewPhrase()`; and removeRandomTPhrase(); (the names of the latter two are self-explanatory):

`SONG => {(^5#1)VERSE, CHORUS}` " "
`SONG => {(^5#1)(&5)VERSE, CHORUS}`
`SONG => {(^5#1)VERSE, VERSE, CHORUS}`
`SONG => {CHORUS}`

Mutating Genomes

This now-familiar pattern is repeated yet again in the top-level Genome mutations – `Genome.mutate()` selects a random mutation from `mutateRandomGene()`, `addRandomNewGene()`, `removeRandomGene()` – and this time also `copyMutate()`. Recall from the Preparation that the Copy-Mutate operation makes a copy of a gene, mutates the copy, then changes some of the 'references to the old gene' to 'references to the new gene'. Examples of the first three of these mutations should not be necessary, and an example of Copy-Mutate in action was given in section 2.4.5.

Notes on Implementation

Some of the mutations were easily implemented by operating directly on `Genome` objects – for example, the simple methods for mutating transforms or adding random `TPhrases` to `Genes`.

Other mutations, such as Copy-Mutate and removing random genes, require substantial changes to be made to the genome structure. Specifically, when we change the number of genes, the internal integer `ID`s of all the genes in the genome need to be updated to preserve their ordering from 0 to *n* (where *n* is the number of genes). This is not just a matter of updating the single `ID` of each gene – we must also update the `IDs` on the RHS of each gene which refer to its children. This could have been implemented using a complex renumbering algorithm on the genome. But a much cleverer method is to use `Genome.toString()` to recover the SARAH description of the genome. We can then build a `GenomeString` object and make the required changes to this `GenomeString` without having to worry about internal `ID`s. We then rebuild the `Genome` from this to give the required result.

For example, Copy-mutate thus functions as follows: we obtain the `GenomeString` of the `Genome`. We then make a copy of a randomly `GeneString`. We find change the LHS of this `GeneString` to a new unique name, using the method described in Section 3.1.2. This `GeneString` is inserted in to the `GenomeString`, which is then used to rebuild the new `Genome`.

3.5.2 Cross-breeding

The method for cross-over has already been discussed in section 3.1.2, and essentially consists of splicing genes from two or more parents, followed by renaming the genes to restore consistency in its string representation. As this operates on multiple genomes, the code is stored in a new `Manager` class. This class will eventually become the 'master' class for the final application, and will be responsible for managing the set of genomes currently in play, and performing user-selected actions on them. For now, though, we are concerned only with the single action `breedGenome`, which takes a vector of parent genomes and returns a new child produced from them.

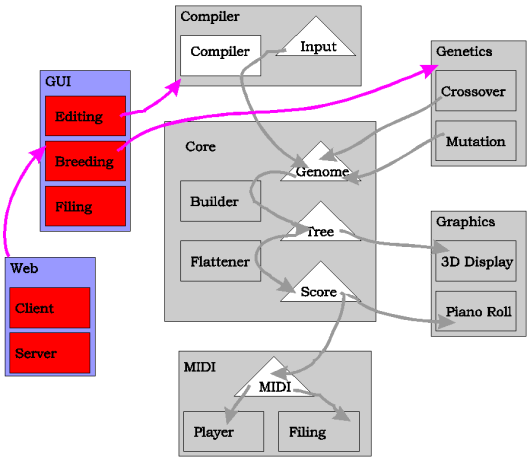
The self-explanatory code for this is shown in Appendix D.

3.5.3 Testing the Evolution Module

After rigorously testing that the mutation code functioned as intended (using the prototype GUI for rapidly editing, and checking), attention was paid to the aesthetic aspects of the mutations. In particular, the probabilities of for selecting different mutations were adjusted to encourage more favorable effects. For example, transposition by integer degrees of the scale is encouraged, but transposition by sharpened and flattened intervals is discouraged (though not entirely prohibited). Transforms with lesser-noticable effects such as amplitude change were made more common than those with large effects such as key change. (The probabilities are stored as private static variables in the appropriate classes).

The crossover code is relatively simple, so a short series of test runs on genomes was enough to ensure correct functionality.

3.6 User Interface



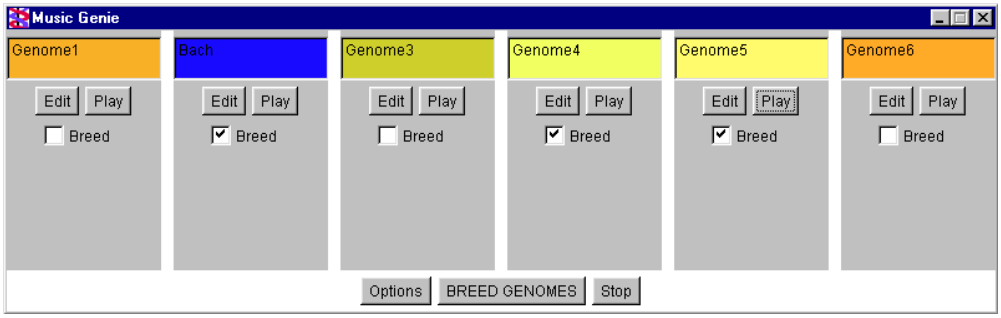
The UI functionality was specified in section 2.5. We require three main windows to be designed:

- A 'Manager', displaying the current generation of genomes for breeding.
- 'Editors', which can be opened from the Manager for editing individual genomes.
- A Web Client, similar to the Manager (but without the editing option).

AWT components were used as the GUI basis (learning Swing was unnecessary, as the interface is reasonably simple.) Low-level AWT components are not discussed here, but are easily inferred from the pictures and any Java book [24].

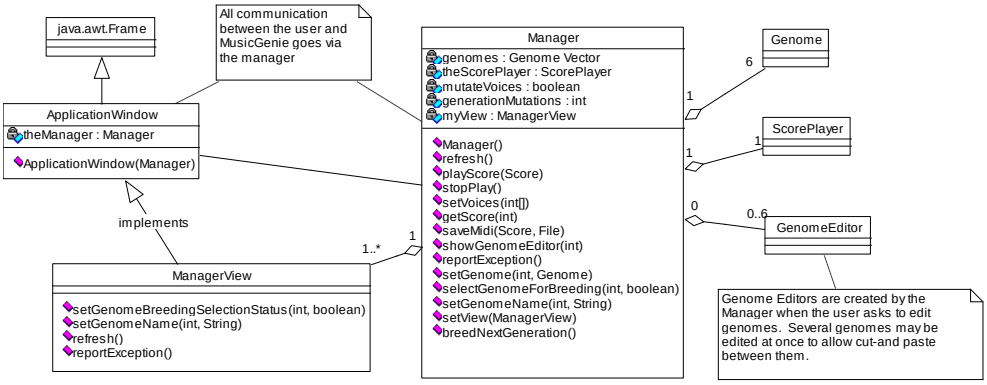
3.6.1 Manager Window

Below, we see panels representing a generation of genomes. Six was chosen for the generation size it is difficult to hold seven or more objects in one's head at once – which is essential if one is to compare all the genomes with each other to choose the best ones. Each genome has a color associated with it, to emphasize genome changes. (The RGB vales of parent colors are crossed-over and mutated to produce child values):



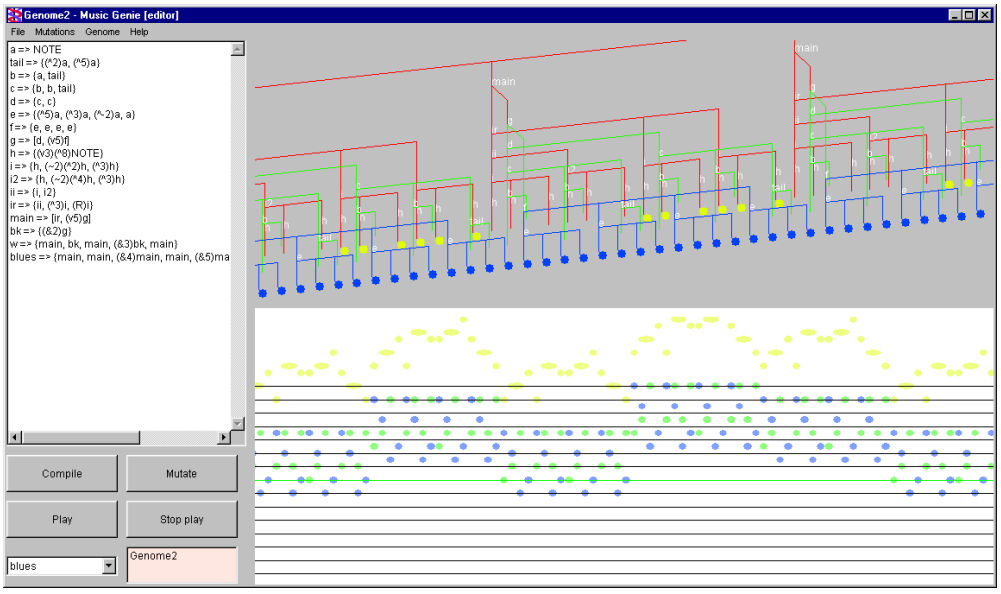
Underneath, the architecture has been designed with extension in mind. The `Manager` class, which provides top level actions on genomes, has no graphical components – just a set of public methods to perform its actions. An interface, `ManagerView`, is used to specify methods that any displays associated with a `Manager` must provide. Finally, the graphical `ApplicationWindow` class implements this interface, and is constructed with a reference to a `Manager`. Thus the window can respond to user actions by calling `Manager` methods, and the `Manager` can tell the window to display the results using the window's own methods. (Note that the window itself does not have access to any other objects in the system – all of its actions must be performed via the `Manager`.) This system has the

advantage that it is now easy to add new front-ends to the `Manager`. For example, a CLI could be provided to for experienced UNIX users; or the `Manager` could become an invisible plug-in for use by other music applications:

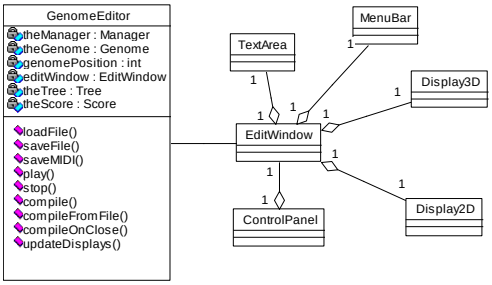


3.6.2 Editor Window

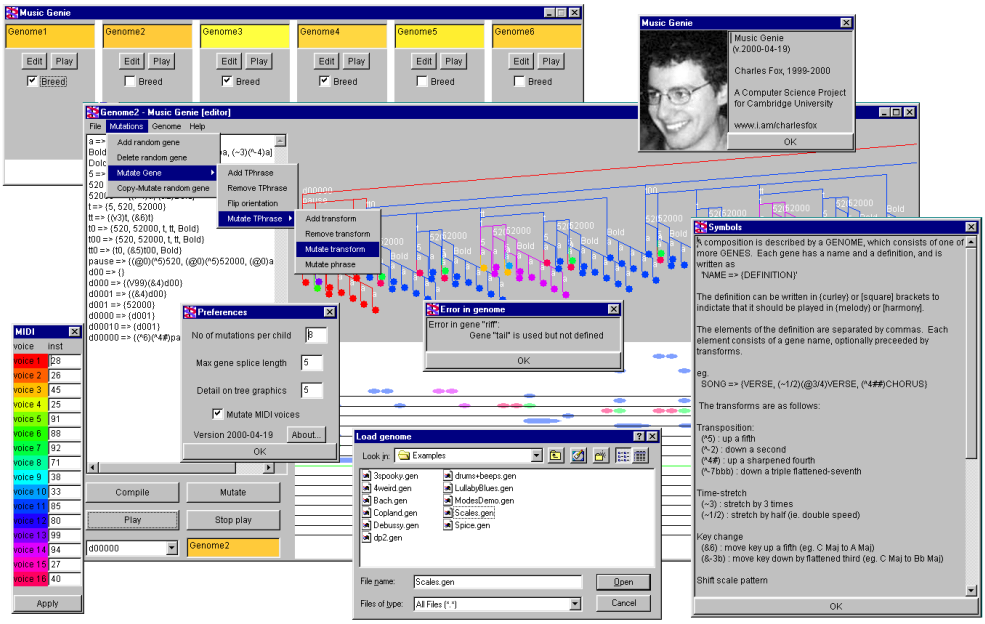
Editor Windows for genomes are opened by clicking one of the 'Edit' buttons in the Manager Window. The panels included in the Editor Window include the smoothly interactive scrolling displays of the tree and score, and code editing area (with cutting and pasting). Large buttons are provided for the most common functions (large buttons take less time to point to than small buttons), while lesser-used features are available in drop-down menus.



The editing system architecture is similar to the Manager's – an 'invisible' `GenomeEditor`, does the work, whist the graphical `EditWindow` calls its methods in response to user actions:



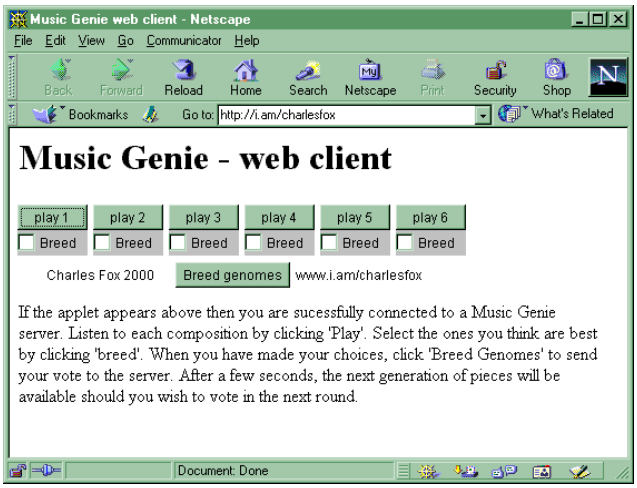
In addition to the Manager and Editor windows, various other dialogs are provided – these are shown in the following montage:



3.6.3 Web client

Using RMI, the `Manager` methods `getScore`, `selectGenomeForBreeding` and `breedNextGeneration` were provided as remote services. A thin client applet was then constructed. The client knows nothing about Genomes; but does include the `Score` and `ScorePlayer` classes. It allows the user to play the current scores, transmitted from the server; and to remotely select and breed the server's genomes. (The architecture is not described here, as its standard RMI techniques can be inferred from its functionality and any Java book [24]).

Unfortunately, no browsers at the time of writing yet support `java.midi`, although the client does work with Sun's Java2 v1.3 *Appletviewer*. (The picture shows the applet running with `ScorePlayer` replaced by a dummy.) A working applet will hopefully be provided on the forthcoming *MusicGenie* website when suitable browsers are released!



3.6.4 Testing the GUI

The editing interface was designed with the help of two composers, who used a series of 'beta' versions of *MusicGenie* to write short pieces. After each composition, the they made suggestions as to how usability could be improved, and these were implemented in the next 'release'.

An important feature which resulted from this process is the selection box in the bottom left of the window (displaying 'blues' in section 3.6.2). This allows the composer to compile subtrees of the whole genome, whilst leaving the code for the whole genome in tact. This is useful in long pieces, when he wants to rapidly edit individual sections of the composition without having to play the whole piece. (In the above window, for example, he could choose to compile a 'main' rather than the whole 'blues').

The original Editor design (as shown in 2.5) featured a 'friendly' system for entering SARAH, where the LHS and RHS of each gene appeared in separate text areas. This was quickly found to be quite tedious, as it was slow to navigate around these boxes. They were quickly dropped and replaced by the large single text field shown here.

4 Evaluation

This chapter presents examples of use and criticism of the program *as a whole*, and suggests areas for future work. The reader is referred back to sections 3.1.4, 3.2.4, 3.3.4, 3.4.1, 3.5.3 and 3.6.4 for details of testing individual modules.

The following should be read in conjunction with the accompanying audio CD:

4.1 Evolution

4.1.1 Mutation and crossover demonstrations (CD Track 2)

Composer Chris Hutchings used *MusicGenie* to encode this short extract from *The Well Tempered Clavier*. It was used to demonstrate the power of the mutation and crossover algorithms. (And its existence shows that *MusicGenie* is expressive enough to represent real music.)

The CD first presents three mutations of the Bach. (These were selected by a human from around 20 such mutations produced by *MusicGenie*.)

Whilst *MusicGenie* was not specified to breed children from radically different parents (as will be explained in section 4.1.5), it is entertaining to try cross-breeding Bach with the Spice Girls! This was achieved by deliberately encoding the Spice Girls so that the ordering of its genes was 'analogous' to the ordering of the Bach genes. The result of this cultural monstrosity is presented at the end of the CD track. Again, the crossover was performed around 20 times, with a human choosing the best result.

4.1.2 Evolution User Walkthrough (CD Track 3)

MusicGenie usually produces 6 compositions per generation, but the number has been limited to four in this brief demonstration of a user walking though two generations of selective breeding.

4.1.3 Long term evolution result (CD Track 4)

This is an example of a composition evolved by around 100 generations of selective breeding. The genetic material used to start the evolution process was made by crossbreeding simple representations of *Fanfare for the Common Man* and a minimalist blues progression. However, the genome here contains almost no resemblance to any of the initial genetic material, as a result of its long evolution.

4.1.4 Client usability

Web surfers' attention spans are typically low, so it is important that the web interface is simple to use. To mimic this, I asked an untrained eight-year-old for his comments:

> "I think that it is very good but it can be improved by making some of the tunes longer. I think it is very easy to use and user friendly."

4.1.5 Future evolution improvements

Heuristics

The main problem with the evolution system is that of the 6 genomes per generation, there are typically only two or three 'pleasing' pieces. The unpleasing pieces are almost always due to one of the following problems:

- long silences appear in pieces (when a 'zero-amplitude' transform is applied to a high-level gene)
- long single-voice sections appear (when harmony genes refer to genes of very different durations)
- many pieces are very short (caused by the top level gene referring to a very low-level gene)

Whilst purists may argue that a GA should not attempt to artificially restrict the evolution process, we could greatly improve the quality of each generation by generating more than 6 genomes, and using simple heuristics (such as tests for the above problems) to select the 6 best ones.

Crossover system and Creative Analogies

The crossover algorithm performs brilliantly on parents which are structurally similar. Since the genomes in a current generation all come from the same parents, their structures are usually similar and there are no problems.

Future work could consider how to perform crossover on radically different parents. Recall that crossover works by assuming that genes *at the same level* in different parents perform similar functions, so can be meaningfully substituted. If the parents are very different, we cannot make this assumption. We must then use other methods to find 'analogous' genes in pairs of parents.

This leads us to the idea of Creative Analogy-Making, famously researched by Hofstadter *et al.*[19]. Such analogies would allow us to meaningfully crossbreed *any* set of parents and would hopefully produce results comparable with Cope's EMI. However, Creative Analogy-Making is notoriously fuzzy (but fun!) – hence the careful use of inverted commas around 'analogous' throughout this project.

Cognitive Conservatism

Minksy[1] described music appreciation as a learning process – claiming that humans find familiar structures more aesthetically pleasing than new ones. (Hence the 'music as cultural knowledge' argument in section 1.1).

This bias towards familiarity is potentially very dangerous for our human-as-fitness-function system. It means that rather than judging each composition on its own merits, listeners are likely to produce 'conservative' judgements, favoring new pieces that sound the most like old ones. Future cognitive work could investigate methods to reduce the effects of this problem.

4.3 Human Composition

4.3.2 Example composition (CD track 5)

To demonstrate *MusicGenie's* power as a composing tool, minimalist composer Chris Ramshaw kindly wrote this short piece (taking only 15 minutes to do so!), and recorded some of his thoughts on *MusicGenie*.

4.3.3 Future language work

Automated parsing

Inputting a SARAH piece from scratch is rather time consuming. A better method would be to parse existing serieses of notes to create the tree and genome structures automatically. In this way, a composer could play in his initial ideas, then use the structural facilities to edit and improve them. Such a system would also allow for easy cross-breeding of existing compositions, if coupled with the AI Analogy techniques above. This work would be similar to Reis's *Meastro* thesis[3] – in fact, a promising project would be to integrate *MusicGenie* and *Meastro*.

Argument passing

Composer Jonathan Warrell suggested that allowing arguments to be passed to genes would be the single most useful improvement to the language. The difficulties of this method (its conflicts with evolution) were discussed in section 2.4.5 – but future work in solving this problem could greatly increase the power of the system.

Non-western tonalities

Recall that the original specification required that we support but discourage non-western tonality. The current design of the scale-degree system allows non-scale notes to be represented by sharpening and flattening intervals. However, should we wish to allow whole compositions in non-western scales, the current architecture allows for the easy extension of adding new transforms, for example, to change the scale patterns. Future work could investigate ways of integrating custom tonality into the composition and evolution processes.

4.5 Performance

A serious concern with the Java language is its speed – especially for large tasks such as compiling long compositions. It was noted that the evolution-only uses of *MusicGenie* retained usability for much larger genomes than uses involving the edit window – due to the large amount of processing involved in building and displaying the 3D tree model.

Code was inserted into the compiler to record it's times for:

- compiling the SARAH code into a playable `Score` (via the `Tree` structure)
- creating the visual tree and score objects (performed only once, after compiling in Edit window)
- displaying these visual objects (done whenever displays updates are requested, eg many times per second during user rotating and zooming)

…for genomes of various representative sizes. (These tests were performed on an AMD-K6III 450MHz/ 64Mb/ Win98 PC, using Sun's JRE 1.2.2):

No of notes	*Playback duration*	*Compile (s)*	*Visual Objects (s)*	*Display (s)*
426	1'15"	0.16	0.44	0.13
1,278	4'22"	0.55	1.21	0.39
2,848	3'38"	1.54	2.20	0.88
5,112	7'45"	2.75	3.51	1.70
10,224	14'45"	7.53	6.92	2.81
17,088	21'52"	15.99	56.80	60.59
20,448	26'15"	73.05	174.82	103.59
28,480	18'13"	289.07	More than 20 mins	N/A (crashes)

The results show that the interactive displays break down for pieces more than about 5 minutes in length. The composer's edit-compile-listen cycle becomes impractical for pieces of around 20 minutes. Evolution usability becomes impractical for 25 minute compositions.

A large amount of hard disc activity was noticed during the final test – suggesting that the computer's memory was in full use and page faults were occurring. This explains the explosion in Visual Object and Display times.

It is interesting that the graphics break down before the 'real work' of the compiler. Whilst the program was specified to generate 'short pieces of music', we can see from these results that should we wish to upgrade the program to deal with larger-scale compositions, then we should concentrate our optimization efforts on the graphics bottleneck. The best way to do this would be to remove all of the current graphics components and replace them by commercial graphics packages such as `Java2D` and `Java3D`.

5 Conclusion

MusicGenie enables human-guided evolutionary composition, and also provides a useful development environment for structural composers. It encourages meaningfulness and aesthetics in both classes of composition through the implicit ideas of hierarchies, scales and degrees in its conceptual model.

That the evolution operators preserve meaningfulness and aesthetics was demonstrated by creating pleasing mutations on the Bach fugue, and the interesting Bach/Spice Girls crossover. The success of evolution in incrementally improving compositions was demonstrated by the walkthrough and long-term evolution result recordings.

The system was used by several human composers, both to input existing works and to create their own new music. One composer commented that encoding existing work is a fascinating exercise in itself, which forces one to think very clearly about the structure and meaning of the music, perhaps in ways not noticed before. Another composer mentioned that the SARAH language mirrors his minimalist 'conceptual framework' well, allowing him to concentrate on rapid musical planning while the machine performed the more tedious work of transforming his structures.

There are, of course, limitations to the system, and almost infinite potential for further work in improving the quality of its output. The major improvements to evolution would be to implement heuristics to automatically reject obviously uninteresting pieces; and creative analogy-making to allow meaningful crossovers between radically different parents[1]. To improve the language, a model for argument-passing would be very useful. HCI work could perhaps find a more friendly SARAH input mechanism, as one composer still commented on its bulkiness.

Three mistakes were made. Debugging the compiler from its text output was difficult, so its implementation was postponed, and the graphics module coded prematurely, to enable faster visual debugging. Secondly, there was confusion over the different versions of Java's MIDI packages – documentation was sparse, and the latest version was buggy - which delayed the MIDI module by a week. Finally, whilst designing my own graphics module was interesting and useful for my future projects, the editing environment would have supported much larger compositions had a commercial graphics package been used.

All original aims have been achieved, and apart from the changes above, the project was completed according to schedule.

[1] Would anyone like to fund my PhD for this?

Bibliography

[1] M. Minsky, 1981. *Music, Mind and Meaning*, in The Music Machine, MIT Press.
[2] A. Forte, 1926. *Introduction to Schenkerian Analysis.*
[3] B. Reis, 1998. *Simulating Music Learning with Autonomous Agents*, Cambridge PhD Thesis.
[4] D. Hofstatder, *Staring EMI in the Eye and Doing my Best Not to Flinch*, CRCC Report #122.
[5] S. Brown, 1996. *Clara Empricost*, www.flatline.org.uk/~silas.
[6] S. Holtzman, 1980. *Generative Grammars and the Computer Aided Composition of Music*, Edinburgh PhD Thesis.
[7] S. Holtzman, 1995. *Digital Mantras*, MIT Press.
[8] D. Cope, 1991. *Computers and Musical Style*, Oxford University Press.
[9] G. Papadopoulos, G. Wiggins. *AI Methods and Algorithmic Composition: A Survey, Critical View and Future Prospects*. Edinburgh University AI Report.
[10] J. Putnam, *Grammidity*, cs.eou.edu/~jefu/Grammidity.html.
[11] K. Lindenmayer, *The Algorithmic Beauty of Plants*, MIT Press.
[12] MIDI Organization, *The MIDI Standard*, www.midi.org.
[13] J. Warrell, 1997. *Cyclical Journeys*, Cambridge New Music Society *CamScores* Archive.
[14] G.F. Luger, W.A. Stubblefield, *Artificial Intelligence*, Chapter 15, Addison Wesley.
[15] R. Dawkins, 1986. *The Blind Watchmaker*, Chapter 3, Penguin.
[16] R.A. McIntyre, 1994. *Bach in a Box*, First IEEE Conference on Evolutionary Composition.
[17] J.A. Biles, 1994. *GenJam: A GA for generating Jazz Solos*. ICMS Proceedings 1994. Computer Music Association.
[18] G. Wiggins *et. al*. *Evolutionary Methods for Musical Composition*. www.dai.ed.ac.uk/groups/aimusic.
[19] D. Hofstadter *et al*, 1995. *Fluid Concepts and Creative Analogies*. Penguin.
[20] M. Harris, A. Smaill, G. Wiggins. *Representing Music Symbolically*. www.dai.ed.ac.uk/groups/aimusic.
[21] E. Taylor, *AB Guide to Music Theory Part II*. Chapter 23. Associated Board Press.
[22] J.S. Bach. *The Well Tempered Clavier*, BMV 846-869, Associated Board Press.
[23] C. Wilson *et. al*. 1998. *Animating Algorithms*, Part IB Group Project.
[24] B. Eckel, 1998. *Thinking in Java*. Prentice-Hall.

Acknowledgements

Thanks to Dr William Clocksin for his help and advice as my Supervisor for this project,

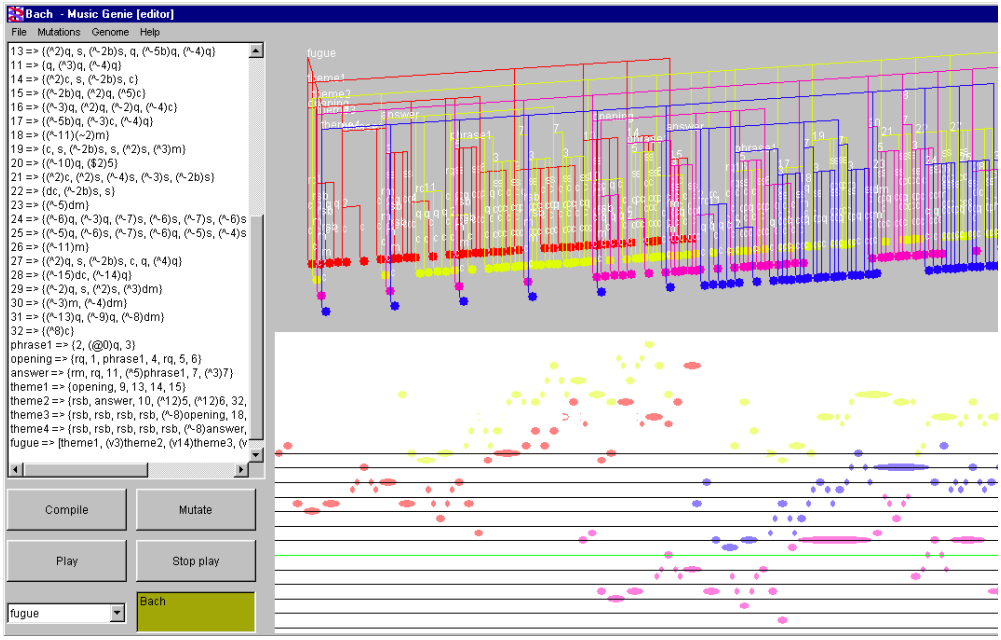
Chris Hutchings, Chris Ramshaw and Jonathan Warrell for composing with *MusicGenie*,

Catriona Massey for demonstrating the evolution system,

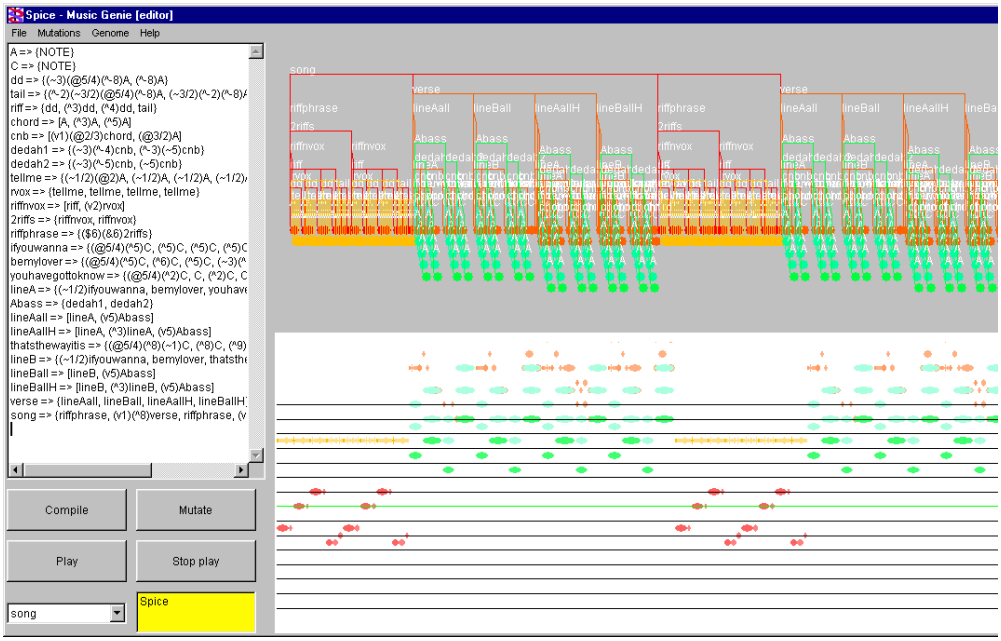
And my geneticist girlfriend, Sarah Williams, for teaching me about sex.

Appendix A: Sample Genomes

Extract from 'The Well Tempered Clavier' Fugue in G Minor (J.S. Bach)



Extract from the 'Wannabe' (The Spice Girls)



Appendix B: Tree constructor code

```
        // uk.ac.cam.cwf22.mg.core.Tree.Tree -- constructor
        public Tree(    int myId,
                        Vector myInheritedTransforms,
                        Vector myNewTransforms,
                        Genome theGenome,
                        Rational startTime
                        )
                        throws BadGenomeException
        {
             this.myId = myId; children = new Vector(); inversions = new Vector();

             //get instructions for making my children, mu orientation, and my start time
             myGene = theGenome.getGene(myId);
             this.orientation = myGene.orientation;
             this.startTime = (Rational)startTime.clone();

             //store the result of consing the transforms in mytransforms
             myTransforms = consVectors(myInheritedTransforms, myNewTransforms);

             // go through the transform list and see if I'm retrograde and/or inverted
             boolean retrograde = this.isRetrograde();
             boolean isNewlyInverted = this.isInverted(myNewTransforms);

             int numberOfChildren = myGene.getLength();

             // var used to keep track of current child's startTime
             // (beginning at my own startTime for first child)
             Rational childStartTime = (Rational)startTime.clone();

             for (int i=0; i<numberOfChildren; i++)
             {
                  //get pointer to the transformedPhrase describing this child from the gene
                  // working forwards through my child list, or backwards if I am retrograde
                  TransformedPhrase childTP;
                  if (!retrograde) childTP = myGene.getTransformedPhrase(i);
                  else childTP = myGene.getTransformedPhrase(numberOfChildren-1 - i);

                  //the number of the child
                  int childId = childTP.Id;

                  //get the new transforms from the TP
                  Vector childNewTransforms = childTP.transforms;

                  Tree child = new Tree(    childId,
                                            (Vector)(myTransforms.clone,
                                            (Vector)(childNewTransforms.clone()),
                                            theGenome,
                                            childStartTime );

                  children.addElement(child);

                  //increment next child's startTime by the duration of this child
                  // - but only if I'm a melody.  Harmonies keep the same start for every child
                  if (orientation==MELODY)
                       childStartTime = childStartTime.plus(child.getLength());
             }
             //if NEW transforms include inversion, find the first degree and invert each child
             if (isInverted(myNewTransforms))
             {
                  Degree d = this.getFirstDegree();
                  this.invert(d);
             }
        }
```

Appendix C: Extracts from Compiler

```
//---------------------------------------------------------------------
// uk.ac.cam.cwf22.mg.compiler.GeneString.makeGene  :  Makes Gene from GeneString
//---------------------------------------------------------------------
public Gene makeGene(String[] names) throws GeneSyntaxException {
     try {
          int Id = 0;                         //these are the initial arguments for the new gene
          Vector tPhrases = new Vector();
          int ori = 0;

          Id = indexOf(LHS, names);           //find the int Id corresponding to the LHS string

          if (RHS.compareTo("NOTE")==0) {     //if leaf node, return a new leaf gene
               Gene leafGene = new Gene(Id);
               return leafGene;
          }
          // the following only runs if its not a leaf gene...

          String workRHS = RHS;               //make new copy of RHS to work on
          ori = findOrientation(workRHS);     //find orientation

          workRHS = workRHS.substring(1,workRHS.length()-1);  //strip brackets from ends of RHS
          Vector tPhraseStrings = split(workRHS, ",");   //and split into comma-separated chunks

          for (int i=0; i<tPhraseStrings.size(); i++) {       //for each chunk, make a tPhrase…
               String currentString = (String)tPhraseStrings.elementAt(i);
               TransformedPhrase currentTPhrase = makeTPhrase(currentString, names);
               tPhrases.addElement(currentTPhrase);
          }
          // Check for recursive gene definitions
          for (int i=0; i<tPhrases.size(); i++) {
               TransformedPhrase currentTPhrase = (TransformedPhrase)tPhrases.elementAt(i);
               if (currentTPhrase.Id >= Id) {
                    throw new GeneSyntaxException("Recursive definition in gene "+LHS);
               }
          }
          // Create and return the new gene.
          return new Gene(Id, tPhrases, ori);
     }
     catch (GeneSyntaxException e)
     {    String old = e.userReport;
          throw new GeneSyntaxException("Error in gene "+LHS+":\n\t"+old);
     }
}


//---------------------------------------------------------------------------------------------
//GeneString.makeTPhrase : takes a TPhrase String (eg. "(^3)(~1/2)VERSE") and makes its Tphrase
//---------------------------------------------------------------------------------------------
private TransformedPhrase makeTPhrase(String tPhraseString, String[] names)
          throws BadTPhraseStringException {

     String s = tPhraseString; //local copy to play with
     String name;
     int tpId = 0;
     Vector transforms = new Vector();

     if (s.length() < 1) throw new BadTPhraseStringException("Name must be at least 1 char long");

     //find the name at the right, and strip it…

     int position = s.length()-1; //position is length-1 (since start at zero)

     //start at the right, move left until a close bracket is found
     //or we run out of chars
     while( s.charAt(position) != ')' && position >0 ) { position--; }

     //position is now the position of the rightmost ')' if there are brackets
     //or is 0 if no brackets
```

```
     if (position>0) {   //if there are brackets...
          //now we know the split position, do the split
          name = s.substring(position+1, s.length());  //name
          s = s.substring(0, position+1);              //chop name off from original
     }
     else { //if no brackets...
          name = s.toString();
          s = "";
     }
     // name now holds the name
     // s now holds the transform list.

     //find int id for this name using the genome namelist
     //throw exception if a name is used which has not been defined in the genome
     try { tpId = indexOf(name, names);}
     catch (ArrayIndexOutOfBoundsException e) {
          e.printStackTrace();
          throw new BadTPhraseStringException("Gene "+name+" is used but not defined");
     }
     // No transforms case: create the Tphrase with the null vector of transforms.
     if (s.length() == 0) { return new TransformedPhrase(tpId, transforms); }

     // Otherwise, transforms exist…

     // check that there are brackets at ends of transforms string - eg. "(^2)(&3)(~1/2)"
     if (!(s.startsWith("(") && s.endsWith(")"))) {
          throw new BadTPhraseStringException("Bad brackets in transform list: "+s);
     }
     s = s.substring(1,s.length()-1);                  //strip end brackets - eg. "^2)(&3)(~1.2"

     Vector transformStrings = split(s, ")(");         //now split on ")(" to give transform list
                                                       //eg.  "^2","&3","~1/2"

     for (int i=0; i<transformStrings.size(); i++)     // transforms from each TransformString
     {
          Transform currentTransform = makeTransform((String)transformStrings.elementAt(i));
          transforms.addElement(currentTransform);
     }
     return new TransformedPhrase(tpId, transforms);   // make the Tphrase and return it
}

//-------------------------------------------------------------------------------------
// GeneString.makeTPhrase : Takes the string for a transform and returns its transform object
//-------------------------------------------------------------------------------------
private Transform makeTransform(String transformString) throws BadTransformStringException
{
     Transform transform;
     String s = transformString;                                  //local copy
     String[] symbols = {"~", "^", "@", "v", "R", "&", "$"};      //(ordered) array of symbols

     // remove first char from string and test it to find type of transform eg. "~1/2" …

     char firstChar = s.charAt(0);
     char[] a = {firstChar};                                      //to create string from char
     String type = new String(a);
     s = s.substring(1, s.length());

     // find index of this symbol in the symbol array
     int typeNo;
     try {typeNo = indexOf(type, symbols);}
     catch (Exception e) {throw new BadTransformStringException("type symbol not listed");}

     // now make the appropriate kind of transform…

     switch (typeNo) {
     case 0:   // "~"
          transform = makeTimeStretch(s);      [NB: The code is only shown for this one transform.
          break;                          The methods for the other transforms work in similar ways.]

     case 1:   // "^"
          transform = makeTranspose(s);
          break;
```

```
        case 2:   // "@"
             transform = makeRelativeAmplitude(s);
             break;
        case 3:   // "v"
             transform = makeVoiceShift(s);
             break;
        case 4:   // "R"
             transform = makeRetrograde(s);
             break;
        case 5:   // "&"
             transform = makeKeyShift(s);
             break;
        case 6:   // "$"
             transform = makeKeyRoll(s);
             break;

        default:  // throw an exception if the symbol is not recognised
             throw new BadTransformStringException("Transform "+type+" not recognised");
        }
        return transform;
}

//-------------------------------------------------------------------------------------
// GeneString.makeTimeStretch : makes TimeStretch object from String
//-------------------------------------------------------------------------------------
private TimeStretch makeTimeStretch(String s) throws BadTransformStringException

{
        // (there are two cases - either 'n' or 'a/b' eg. "~2" or "~1/2" )
        int numerator, denominator;
        Vector ratValues = split(s, "/");                       //split on "/"

        try {
             Integer i,j;

             switch(ratValues.size())
             {
             case 1:                                            //integer timeStretch
                  i = new Integer((String)ratValues.elementAt(0));
                  numerator = i.intValue();
                  denominator = 1;
                  break;
             case 2:                                            // rational timeStretch
                  i = new Integer((String)ratValues.elementAt(0));
                  j = new Integer((String)ratValues.elementAt(1));
                  numerator = i.intValue();
                  denominator = j.intValue();
                  break;
             default:                                           // if neither, throw an exception
                  throw new BadTransformStringException("Bad TimeStretch string");
             }

             // create and return the TimeStretch object
             Rational r = new Rational(numerator, denominator);
             return new TimeStretch(r);
        }
        catch (NumberFormatException e) {
             throw new BadTransformStringException("Bad TimeStretch string");}
}
```

Appendix D: Cross-breeding code

```
//---------------------------------------------------------------------
// uk.cam.ac.cwf22.mg.gui.Manager.breedGenome
// breeds a baby genome from a set of parents
//---------------------------------------------------------------------
  private Genome breedGenome(Vector parentGenomes)
  {
    //for height of new genome choose gaussian random between min and max lengths
    int babyHeight = Stats.getRandomInt(minGenomeHeight(parentGenomes),
                    1+maxGenomeHeight(parentGenomes));

    Vector babyGenes = new Vector(babyHeight);
    String[] babyNames = new String[babyHeight];

    try
    {
      int height = 0;
      //beginning at NOTE and working up...
      while (height < babyHeight)
      {
           //choose random genome
           Genome parent = (Genome)chooseRandomElement(parentGenomes);

           //choose random (potential) length of genome to splice from it
           int potential = height + Stats.getRandomInt(0, maxSpliceLength);

           //while next of these exists
           while (height < parent.genes.size() &
                   height < potential &
                   height < babyHeight)
           {
                //add it to genome
                babyGenes.addElement(parent.getGene(height).clone());

                //get potential name, and check for clashes (a-conversion)
                String testName = parent.names[height];
                babyNames[height] = Genome.findNewUniqueName(testName, babyNames);

                height++;
           }
      }
    }
    catch (Exception e)
    {
      reportException("Internal cloning Error\n(please report on the Music Genie website)");
      e.printStackTrace(); return null;
    }

    Genome baby = new Genome(babyGenes, babyNames);

    //mutate the colors and instruments
    baby.color = crossBreedColor(parentGenomes);
    baby.voices = crossBreedVoices(parentGenomes);

    //Crossover now complete.  Now add some random mutations to the baby
    for (int i=0; i<generationMutations; i++) {baby = baby.mutate();}

    return baby;


}
```

Appendix E: Project Proposal

Overview

The program will provide a representation for hierarchical music structures, for the purposes of:

(a) computer-*aided* composition, using a simple 'genome' language for users to create pieces
(b) the mechanical *evolution* of pieces, using human listeners to select 'fittest' mutations

General description

Introduction

A musical composition can be thought of as a complex tree structure – with features on many different structural levels. For example, a rock drummer may think of a song as being simply:

```
SONG -> INTRO, VERSE, CHORUS, VERSE, CHORUS, BRIDGE, CHORUS, ENDING
```

While a guitar player would think in a further level of detail – such as the chords forming each line of the verses:

```
VERSE -> (Eb)LINE1, (Eb)LINE2,   (Ab)LINE1, (Ab)LINE2
```

And the singer will have to know the actual notes forming each of the lines:

```
LINE -> (Eb)QUAVER, (Eb)QUAVER, (Db)CROCHET) , (Eb)CROTCHET …
```

Structural techniques have been used to compose music for hundreds of years – composers such as Bach, Schoenberg and Reich rely heavily on the idea of repeated structures and transformations performed on them. Eg.

```
SCHOENBERG -> SERIES, (invert)SERIES, (play backwards)SERIES, (up a fifth)SERIES
SERIES -> Eb, Ab, C, D, Db, G, B, F, Bb, Db, A, E
```

would form a simple Schoenberg-style musical fragment.

I intend to write software that will allow the human composition and mechanical evolution of music by structural rules (or 'genes') similar to those above.

Previous attempts

The idea of using grammars to generate music has been around for a long time – in particular, Steven Holtzman's 1980 PhD Thesis describes the use of generative grammars coupled with a function language to aid the human composition process. His program uses a very rich grammar to define sets of potential pieces, using rewrite rules with alternative RHS strings – and a complex function language which is used to select the RHS (as a function of its position in the piece, and previous selection form the same rule). Once a human had input a grammar from say, a Bach piece, the machine could then use different selection functions to generate more pieces 'in the style of Bach' from that grammar.

My project is inspired by Holtzman's thesis, but has several very major differences:

Evolution

I would like my program to be capable of performing mutations on the genes – and to allow users to select the best-sounding mutations (hopefully via the web) in order to let the music evolve into a 'perfect' piece. So rather than just being an 'aid to composition', the program (along with its listeners) also becomes a composer in its own right.

Simplicity

The Holtzman grammar and language are syntactically and semantically complex – and it is unlikely that a mutation process would be capable of causing the 'small but noticeable' changes required for evolution. For this reason, I have developed a much simpler system for the genes.

Determinism

The production system I will use is deterministic – there are no alternative RHS strings. Different versions of structures will be produced by making a copy of the structure and mutating it, rather than my providing multiple choices for a single production. For example, using multiple RHS, two different types of verse could be represented as:

```
SONG ->    CHORUS, VERSE, CHORUS   |  VERSE, CHORUS, VERSE
```

But with my system, a copy will be made and changed: (I call this operation 'copyMutate')

```
SONG ->  CHORUS, VERSE, CHORUS
SONGCOPY ->  VERSE, CHORUS, VERSE
```

This means that the genome defines a specific piece of music, rather than a class of pieces produced by a grammar.

Transformations

Example of transformations:

```
SONG ->  (&20)VERSE, (#5)VERSE, (BK)(>2)CHORUS
```

(A verse at 20% amplitude, a verse transposed by 5 degrees of the scale, and a backward chorus at twice the speed)

Each phrase (such as a verse) will store the following information:

Pitch
Key
Amplitude
Instrument
Tempo
Start time
Pointers to its children (the objects made by the production)
List of Lists of transformations applied to each child

The top-level SONG object will be given default values, and the values for its children are inherited and transformed appropriately, forming a tree structure. Eventually, we reach the bottom of this 'musical tree' to find the values describing the notes themselves.

Note that the notion of 'Key' is present in the nodes of the tree- this allows for transformations which work in terms of scale intervals rather than semitones, and should enable aesthetically pleasing music to be generated more easily (if the user wishes to use those transformations).

Polyphony

My musical tree will be 3-dimensional – some productions will allow two or more branches to appear simultaneously on different voices:

```
VERSE -> [GUITARVERSE|BASSVERSE|DRUMVERSE]
```

And there will be transformations which can move musical objects around to different instrument tracks. Imagine the tree as growing downwards onto a 2D plane – then the axes of the plane will represent time vs. instrument –

with the instruments roughly ordered from woodwind through strings to brass and basses so that small mutations in instrument space will make sense:

```
FANFARE -> [TRUMPETS | (*1)TROMBONES | (*2) TUBAS]
```

NB. The grammar used here is only a rough sketch – and may well bear no relation whatsoever to the final syntax)

Mutations

Rather than just changing random characters in the genes, I will write a set of 'smart' mutations – which will ensure that they always generate meaningful productions (evolution would take a long time otherwise!). When in mutation mode, the program will play the original piece and several variations to the user – who may then select their favourite version, which becomes the new basis for the next mutations. If time allows, I would like to investigate the possibility of 'cross-breeding' two or more pieces together, by writing smart mutations that will merge their genomes in 'meaningful' ways.

Implementation

The program will be written in Java, because:

- I have the most experience in this language
- The client/server system will be easy
- Java has good facilities for the GUI
- Object-oriented, so easy to carve the project up
- Java is my favourite language

The MIDI output system may have to be written in C, to obtain access to system functions. I need to research how to do this. Java 2 v.1.3 is currently in beta, and will feature full MIDI facilities – I hope that it will be released in time! The MIDI part of the project will be written in Jan/Feb 2000, so if the new Java is available then it will be used. Otherwise C will do the job.

Plan of work

The project breaks down very nicely into separate packages. Most packages are independent of all the other packages except the core, on which everything is dependent. (The only exception is that the Mutation package interacts with the Compiler package.)

For this reason, each package will be built, tested, and integrated with the core before moving on to the next package. Due to the high level of independence, there is no need for a large 'integration testing' phase at the end of the project – the only 'integration' is with the core.

It also means that the precise design of each package does not need to be finalised until that package

Michaelmas 1999 weeks 1-2 – research and design

1.Research – existing similar work, Java development environments, MIDI formats, buy a PC
2. Grammar – specify the production language
3. Interfaces – finalise function of the five packages (see below) and specify how they will interact

Michaelmas 1999 week 3-4

4. Designing, Coding and Testing the Core
  - Tree structure – classes for the nodes and leaves making up the tree
  - Simple tree display – just text to begin with, then maybe 2D graphical tree*
  - tree lineariser – will walk over the tree, collecting the leaves into a score object
  - score data structure – a score, containing notes and their positions
  - note display – simple text output
  - hard-coded tree object instances to test

*this may not be necessary, as an advanced 3D display may eventually replace it. But the text display may be insufficient for understanding of the program during debugging, in which case the 2D tree will be a useful temporary measure.

At this stage, the hard coded objects will be linearisable and the results displayed as text. (or a simple 2D tree)

Michaelmas 1999 week 5-6

5. Designing, Coding and Testing the Input system
   - user interface – a display/input panel showing the genes and allowing editing
   - parser – will take input strings from the user, and form them into a genome object
   - Builder – takes the genome and builds the tree structure
   - filing – simple loading and saving of the genome

   At this stage the user will be able to input, load and save genomes, and view the results of linearisation on them as a text representation of the notes.

Michaelmas 1999 week 7-8

6. Graphics
   - Wireframe 3D Tree representation

Christmas holiday 1999

- Complete and test all of the above.

At this stage, the program will allow user input of a genome, and display the 3D tree which is produces.

Lent 2000 – weeks 1 and 2

7. MIDI
   - (Player class – will take a score object and play it through soundcard's MIDI out)*
   - MIDI file exporter – takes score object and saves a MIDI file

   *this optional component will make use of *javax* in Java 2 v1.3, if it is released on time! (Febuary 2000). A beta version is available now, but it will be best to wait for the full version. The player class will be a very small job using *javax.*

Lent 2000 - weeks 3 aesthetic testing

This is the first time that we will actually hear any results from the program. It is possible that this may reveal aesthetic errors in the grammar system, which can now be tweaked to produce 'nicer' results in time for…

Feb 4 – Performance Review

The program will now allow fully functional computer-aided composition. The user will be able to input a genome, then hear their piece and see it represented as a tree on scree.

Lent 2000 Weeks 4-7

7. Genetics system
   - GUI – panel with mutation and selection controls
   - Various mutaion functions which take a genome and return a mutated genome
   - (this should be the fun part, so has been saved for lent fifth week!)

If work runs ahead of schedule, the following extensions may be added. Work on these may run in parallel with other work – in particular, writing the extended graphics can be classed as 'fun' for light relief during the duller coding periods!

Lent 2000 Week 8

A spare week for overspill or optional extras.

(Optional) Web interface

- Allow the Genetics package to run as a server
- Java applet client to remotely control Genetics

(Optional) Advanced graphics

- add features to 3D tree display (the main visual component), eg, rotating, zooming
- 'piano roll' display of the final notes
- maybe add simple 'traditional notation' display if time still left.
-

All coding should be finished by the end of Lent term. (Though the holiday can be used for overspill coding if an emergency arises)

Easter Holiday 2000

8. Write the whole dissertation (in draft)
9. Fine-tune mutation systems to produce 'nicer' pieces more of the time

Easter 2000

10. Finishing and tidying up dissertation.

Development environment

I would like to use a Java development/debugging environment – I used MS Visual C++ over the summer and now I really can't imagine writing anything of this scale using anything less!

I will hopefully have my own machine in a few weeks - AMD400/64Mb/12Gb/17"/Win98 networked PC. I will backup a new copy of the whole program to Pelican every day that I am working on it.

(Until the machine arrives, I can write the first few classes in emacs on Thor – the development environment is only really needed when the project starts to get large.)

Supervision arrangements

Dr Clocksin of Trinity Hall has agreed to supervise my project. He has made 2.30pm on Monday afternoons available as a shared supervision time for myself and three other students who are working on music-related projects.
The overseers are Dr Robinson and Dr Moore.