An Object-Oriented Representation of Pitch-Classes, Intervals, Scales and Chords

François Pachet,
LAFORIA - Institut Blaise Pascal, Boite 169,
4, Place Jussieu, 75252 Paris Cedex 05, France
E-mail : pachet@laforia.ibp.fr

Abstract
The MusES system is intended to provide an explicit representation of musical knowledge involved in tonal music chord sequences analysis. We describe in this paper the first layer of the system, which provides an operational representation of pitch classes and their algebra, as well as standard calculus on scales, intervals and chords. The proposed representation takes enharmonic spelling into account, i.e differentiates between equivalent pitch-classes (e.g. C# and D*b*). This first layer is intended to provide a solid foundation for musical symbolic knowledge-based systems. As such, it provides an ontology to describe the basic units of harmony. This first layer of the MusES system may also be used as a pedagogical example for those wishing to apply object-oriented techniques to musical knowledge representation. A document describing the system in full details is available on request.

Résumé
Le système MusES a comme objectif de représenter les connaissances musicales nécessaires à l'analyse harmonique de séquences d'accords en musique tonale. Nous décrivons ici la première couche du système qui propose une représentation opérationnelle des notes et de leur algèbre, ainsi que des intervalles, gammes et accords. Cette représentation a comme particularité de prendre en compte les problèmes d'enharmonie, i.e. de différencier les notes équivalentes comme Do# et Ré*b*. Cette première couche est utilisée pour l'étude de mécanismes d'analyse harmonique et peut être considérée comme une ontologie des concepts de base de l'harmonie. Le but de ce document est aussi de proposer un exemple non trivial d'application de Smalltalk-80 à l'usage des musiciens désirant se lancer dans la programmation par objets. Un document plus détaillé sur le système est disponible sur demande.

Mots-clés
Programmation par objets, représentation de connaissances, analyse harmonique

1. Introduction

Musical Analysis is an ideal field for testing knowledge representation techniques. It involves complex knowledge which is well documented, and many examples are available. Lots of research have been devoted to complex harmonic problems, such as performing complete harmonic analyses of tonal pieces or extracting deep structures in jazz chord sequences.
We focus here on a remarkably simple problem, which, to our knowledge, has yet never been fully addressed. The problem is simply to provide a "good" representation of the algebra of pitch classes, including the notion of "enharmonic spelling", which is so vital to tonal harmony, and a representation of intervals, scales and chords to serve as a foundation for implementing various types of harmonic analysis mechanisms. This problem may be considered trivial compared with more complex problems such as computing Shenkerian analysis of Debussy's pieces, but it has always been solved in ad hoc ways (usually in Lisp), using idiosyncratic representation techniques. For instance, [Winograd 93] emphasises the importance of taking enharmonic spelling into account, but proposes an ad hoc representation

of chords as (Lisp) dotted lists. Similarily, [Steedman 84] proposes a solution for performing harmonic analysis of chords sequences but, considers all the entities (chords, intervals or notes) as Prolog-like constants and is interested only in higher level properties of sequences deduced from the mere ordering of their elements.

Our goal here is not only to write a program that solves the problems mentioned above, but also to explicitly represent the underlying mechanisms of pitch-class calculus. This representation is claimed to be natural, and the mechanisms that implement the operations on pitch classes are considered isomorphic to human operations. Pushing this idea to its limit (which we occasionally find ourselves doing), the system described here may be considered as a *substitute for* a first-year text-book of introduction to the basics of harmony. Indeed, many of the mysteries of music notation are explicitly solved here simply because the basic entities and mechanisms of music notation are given an operational status.

We will first spend some time on defining precisely the algebra of alterations in pitch-classes and interval computations (parts 2, 3, 4). These parts are important because they are the foundation of all the system, but they are not the most thrilling. Parts 5 and 6 deal with computations on scales, chords, and scale-tone chords, and should be much more exciting to the reader. Finally we show how the system can be easily extended, e.g. to take into account exotic tonalities.

2. Algebra of pitch classes

We are interested in representing *pitch classes*, i.e. octave-independent notes and their relations. For example, *pitch class* C refers to the set of all possible C's (C1, C2 and so on, hence the name pitch-class). In order to avoid confusion - because the word class is very polysemic - we will use the word *note* to refer to pitch classes. This convention is also needed in our context, since we will be speaking of "classes" (in the sense of object-oriented programming) of such notes, and want to avoid talking about *pitch class classes*.
For example, *note* C will actually refer to *pitch class* C, i.e. the set of all Cs (modulo 12). We will not consider *actual notes*, with actual pitch (such as Midi pitches in [1 .. 127]) in this presentation. The extension of our theory to represent actual notes - which may be thought of as "instances" of pitch classes - will be discussed in conclusion, and is rather straightforward once the theory of pitch classes is correctly set.

Here is a wish-list of what a good representation of notes (read pitch-class) should take into account :

-A note has a unique name. There are conceptually 35 different notes : 7 naturals, 7 flats, 7 sharps, 7 double sharps and 7 double flats. The unicity of notes is actually very important. There is only one occurrence of each note (in our octave-independent context). Practically, this means that, for example, the minor second of B (C) is *physically* the *same* note as the minor seventh of D, and so on.

- There is a non trivial algebra for notes. The notes are linked to each other half-tone or tone wise, and form a circular list. But some notes are *pitch-equivalent*, (e.g. A# and B*b* , or C##, D and E*bb*). Although the ability to differentiate between equivalent notes may not seem important at this point, it becomes a crucial point when doing harmonic computations. There is a subtle difference between C# and D*b* which actually appears only when *scales* come into play : for instance, the major scale built from C# contains no C, whereas the major scale built from D*b* does contain a C. Stated differently, the names of notes contain condensed harmonic information that are required by harmonic analysis techniques. Our theory should be able to interpret this information.

- There is an non trivial algebra of alterations, which includes the following equations:

o *b* = *b* o # = identity.
For any x in (#, *b*, natural), x o natural = natural.

This algebra is non trivial because not everything is allowed, e.g. triple sharps.

- Notes are linked by the notion of *interval*, which, in a way, *preserves* this algebra. For instance, the diminished fifth of C is not the same note as the augmented fourth of C, but the two notes are equivalent.

- Certain intervals are forbidden for certain notes : for example, the diminished seventh of C*b* does not exist (it would be B *bbb* !).

- Certain scales do not exist, by virtue of the preceding remarks : G# major is impossible (because it would contain a F## in its signature). The same holds for D*b* harmonic minor, and so on.

Although it is certainly possible to write a global algorithm in any procedural language (such as Pascal or Lisp) that takes all these cases into account, there is clearly here a better solution. This solution is based in *abstract data types*, and consists in considering all these 35 notes as a collection of *instances* of various *types*, each type having its own structure and set of *operations*. This approach not only yields a simple implementation, but also provides us with a clear understanding of the operations on pitch classes.

3. Notes as abstract data types

The main idea underlying our representation paradigm is to model the world as a collection of *abstract data types*, i.e. we do not separate operations on one hand and data structures on the other, but rather try to define types (or classes in object-oriented programming) which gather structures and operations. The theory of *algebraic data types* [Mahr 80] gives a formal framework to represent abstract data types and the formal properties of relations. Abstract data types and object-oriented programming are particularily well suited to represent musical knowledge (Cf. for instance [Smaill&Wiggins 90] who use abstract data types to represent "constituents" useful for analysis, or [Pope 91] who use Smalltalk for sound editing and real-time algorithmic composition). As it turned out, the problem of representing notes and their algebra is a prototypical example as it fits nicely in this formalism.

This approaches leads to us to considering the notes as follows :

- All (35) notes are not equal. Some operations are permitted on some notes and not on others. There are 5 different types of notes : NaturalNotes, SharpNotes, FlatNotes, DoubleFlatNotes and DoubleSharpNotes. It is interesting to distinguish different types of notes because its gives a precise definition to alterations : the #, *b* , and natural, may then be seen as *polymorphic functional operations* on types.
For example, the # operation maps the `NaturalNotes` to `SharpNotes` : A# is then seen as the result of operation # to note A (instance of `NaturalNote`), which yields an instance of `SharpNote`, i.e. :

sharp : `NaturalNote` -------> `SharpNote`
sharp(x) is written x#.

This operation is polymorphic because there are actually several distinct sharp operations, depending on the type of the argument. An other # operation maps `SharpNotes` to `DoubleSharpNotes` (e.g. A## = sharp (A#)), and an other one maps `FlatNotes` to `NaturalNotes` (A*b* # = sharp (A*b*) = A), and `DoubleFlatNotes` to `FlatNotes`.

Some operations are common to all note types (e.g. the operation natural), other are specific to one type of note (e.g. the operation *followingPitch* that links C to D, D to E and so on, is valid only for natural notes) and other to a group of note types (e.g. the *sharp* operation is valid for all note types except `doubleSharpNotes`).

Similarily, the *natural* operation is simply identity when applied to NaturalNotes (A natural = A), but is quite different when applied to SharpNotes (A # natural = A) and still different when applied to double SharpNotes (A ## natural = A). This polymorphism of the natural operation is naturally captured by abstract data types.

3.1. From Abstract Data Types to Object-Oriented Programming

Although the theory of abstract data types sheds a new light on the algebra of pitch-class, it does not allow us to write a completely *operational* specification of the mechanisms. We could write out all the axioms of the algebra of pitch-classes and intervals but this is not what we will do now. We will consider a variant/descendant of this formalism, namely object-oriented programming and Smalltalk-80. Object-oriented programming is based on this very idea of defining abstract entities that gather structure and operations in the context in programming languages.
The vocabulary here is a little bit different : Types are called *classes*. Classes define structure in terms of *instance variables* (or slots, attributes). Each class also has a set of *methods*, which are the operations understood by its *instances*. Polymorphism in object-oriented languages is naturally present, since several classes may have different methods having the same name. An important feature of object-oriented programming is the *inheritance* mechanism between classes, that allows factoring common structure and behavior (Cf. [Perrot 92] for a general introduction to object-oriented programming and representation).

In this document, methods will be written with the following format :

! aClassName methodsFor: aProtocol!
aMethodName and its arguments
the text of the method.

Where `aProtocol` is simply a set of related methods for a particular class. The text of the method is a set of expressions. Each expression is a message sent of the form: `object messageSelector arguments` (Cf. [Goldberg&Robson 89] for further details about Smalltalk). We will describe the main methods of the system, but not all of them. The reader whishing to try the system out may obtain the Smalltalk source code by e-mail.

3.2. The hierarchy of notes

In order to represent notes according to these requirements, we define a hierarchy of classes as follows. Each class has its set of instance variables and operations :

1. `Note` represents the root of all classes representing note. It is an abstract class and has no instance variables.

2. `NaturalNote` represents natural notes. There are 7 instances of this class, representing the 7 natural notes (A, B, C, D, E, F, G). Natural notes form the core of the system :

-They have a *name*, (actually they have two names, to allow French terminology : A = La, B = Si, etc...). The name is used for global access and printing.
- They are linked to each other according to the order (A, B, C, D, E, F, G). This is represented by two instance variables : *following* and *preceding*, that point respectively to the following and preceding natural note,
- Moreover, in order to represent the various intervals between notes, we assign to each natural note an arbitrary *semiToneCount*, so that, e.g. semiToneCount(A) = 1, semiToneCount(B) = 3, .., semiToneCount(G) = 11. This semiToneCount is used for interval computations (Cf. method `alterate:toReach`).
- Finally, there are two pointers towards the *sharp* and *flat* notes generated by the natural notes. They represent the function sharp (resp. flat), which maps `NaturalNotes` -> `SharpNotes` (resp. `FlatNotes`). These notes are instances of `SharpNote` (resp. `FlatNote`) (Cf. below).

The structure of class `NaturalNote` is therefore :

```
Note subclass: #NaturalNote
    instanceVariableNames: 'name nom semiToneCount following preceding sharp flat'
```

Class `NaturalNote` defines methods to access following, preceding, sharp and flat notes. These 4 methods are simple accessing methods : their result is the value of the corresponding note. These values are assigned once, at initialization time (Cf. initialization of notes). For instance, the method sharp is defined as :

```
!NaturalNote methodsFor: 'accessing'!
sharp
    ^sharp
```

3. `AlteredNote` is the root of the classes representing altered (and doubly altered) notes. It is an abstract class. It defines only one instance variable called *natural* pointing back to the natural note it comes from. For instance, A#, A##, A*b* , and A*bb* all have A as their *natural*.

4. Finally, there are 4 subclasses of `AlteredNote` for representing respectively sharp, flat, doubleSharp and doubleFlat notes. These classes implement the methods sharp, flat and double flat so as to respect the natural algebra of alterations. For instance, class `FlatNote` implements the following `sharp` method :

```
!FlatNote methodsFor: 'accessing'!
sharp
"my sharp is simply my natural note"
    ^natural
```

Conversely, for sharp notes, the flat operation is defined as the natural operation :

```
!SharpNote methodsFor: 'accessing'!
flat
"my flat is simply my natural note"
    ^natural
```

For `DoubleFlat`, the `sharp` method will consist in delegating the message to the natural note :

```
!DoubleFlatNote methodsFor: 'accessing'!
sharp
"x bb # = x b"
    ^natural flat
```

Method `flat` in `DoubleSharpNote` is similar.

Finally, we need to represent the functional link between a flat (resp. sharp) note and its corresponding doubleFlat (resp. doubleSharp). This is realized by defining an instance variable in class `FlatNote` pointing to the corresponding doubleFlat note (and idem for sharp). Thus, the method flat is implemented as a simple access method for `FlatNote` (idem for sharp in class `SharpNote`).

To conclude, here is the list of all the implementations of the flat method (the same mechanism applies for the `sharp` operations) :

```
!NaturalNote methodsFor: 'alterations'!
flat
    ^flat
```

```
!FlatNote methodsFor: 'alterations'!
flat
    ^flat
```

```
!SharpNote methodsFor: 'alterations'!
flat
    ^natural
```

```
!DoubleSharpNote methodsFor: 'alterations'!
flat
    ^natural sharp
```

Note that the flat operation is intentionaly not defined for class `DoubleFlatNote`. The flat message sent to a `DoubleFlatNote` will raise an error, which is conform to our philosophy. Idem for method sharp in class `DoubleSharpNote`.

3.3. Equivalence of pitches

Last, we introduce a method for testing the equivalence of pitches. This method, called `pitchEquals:` tests the semiToneCount, and allows to represent the equivalence of certain notes. This method is implemented as follows :

```
!Note methodsFor: 'testing'!
pitchEquals: aNote
        ^self semiToneCount = aNote semiToneCount
```

To implement method semiToneCount, we will once again use polymorphism. The method is defined as follows in the 5 classes :

```
!NaturalNote methodsFor: 'access'!
semiToneCount
"a simple acess method"
    ^semiToneCount
```

```
!FlatNote methodsFor: 'access'!
semiToneCount
    ^natural semiToneCount - 1
```

```
!SharpNote methodsFor: 'access'!
semiToneCount
    ^natural semiToneCount + 1
```

```
!DoubleSharpNote methodsFor: 'access'!
semiToneCount
    ^natural semiToneCount + 2
```

```
!DoubleFlatNote methodsFor: 'access'!
semiToneCount
    ^natural semiToneCount - 2
```

Now all the note classes have been defined, and the algebra of pitch is correctly represented. The note classes form the following inheritance tree (instance variables are between parenthesis, and inheritance is represented by indentation) :

```
Object ()
    Note ()
        NaturalNote (name following preceding sharp flat semiToneCount)
        AlteredNote (natural)
            SharpNote (sharp)
            FlatNote (flat)
            DoubleFlatNote ()
            DoubleSharpNote ()
```

Figure 1 represents the class hierarchy as well as the instances A, B, A#, A##, A*b*, and A*bb*, and their relationships.

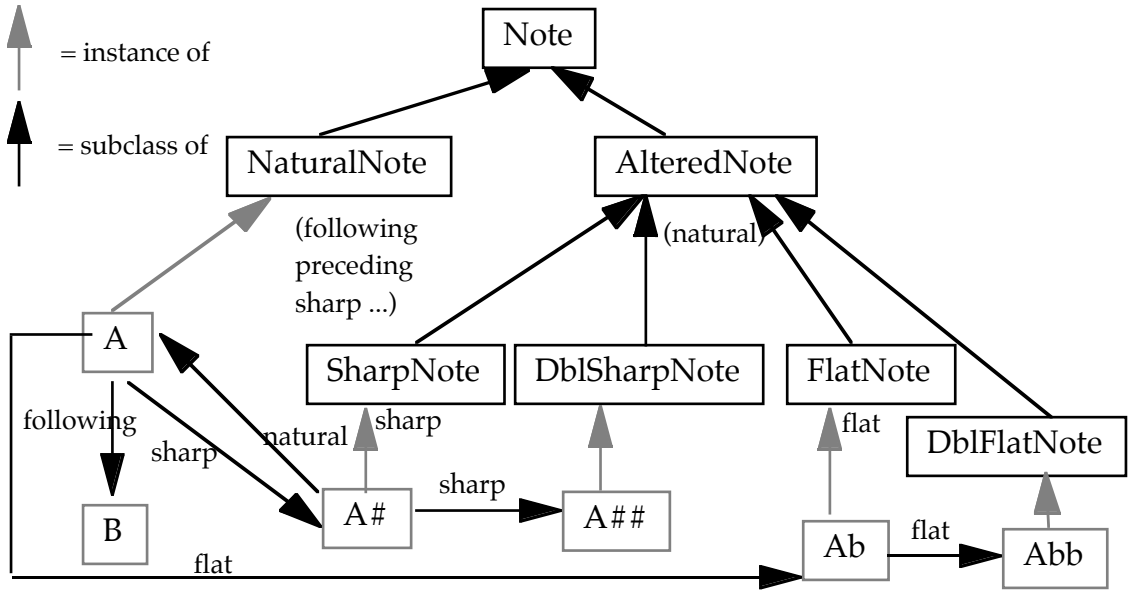

Figure 1. Relationships between several notes.

3.4. Note creation and initialization

Once these classes are defined, we define an initialization method as a class method for `Note`. This method will create the 35 instances of notes and link them according the instance variables defined above. Since notes are unique, we want to have a global access to them. This global access is realized by 7 class variables `(A` to `G)` which point to the corresponding natural notes created during the initialization phase. A set of special methods are written to access these natural notes by messages such as A, B, C (or do, re, mi). Altered notes are then accessed by sending appropriate alteration messages to natural notes.

Here is a micro session that illustrates note access[1].

Note C	-> C
Note C sharp	-> C#
Note C sharp sharp flat	-> C#
Note C flat flat flat	-> error: 'flat' not understood by class DoubleFlatNote
Note C sharp pitchEquals: Note D flat	-> true

4. Intervals

Now that notes and the algebra of alterations are correctly defined, interval computation is easy to add (and more interesting !). The same kind of requirements that hold for notes hold for intervals, namely the possibility of differentiating synonymous intervals. For instance, we want to be able to distinguish the *diminished fifth* of C (which is G*b*) from its *augmented fourth* (which is F#, a pitch-equivalent of G*b*).
There are a number of things one can do with intervals, which are :

- computing the top or bottom lacking extremity of an interval, given a note (e.g. *what is the major third of C*, or *what is the note whose major third is C*),

[1] Note that the instance of `SharpNote` that represents C# is *accessed* by sending the message sharp to the note C, but *prints itself* as C#.

- computing an interval given two notes. For example, we want to be able to answer the question : *what is the interval between C and F# ?* (the answer here is *an augmented fourth*),
- performing some computations on intervals themselves, such as :
  adding intervals (e.g. a major third + a perfect fifth = a major seventh)
  computing *reverse* intervals (the reverse of an augmented fourth is a diminished fifth).

In order to do so, we must have an explicit representation of intervals, that supports those operations. The class `Interval` is defined with the following structure :

a *type*, which indicates how many notes should be enumerated. The type is represented by an integer (e.g., 2 for a second, 3 for a third, and so forth),
a number of *semiTones*, that represents its actual width (also an integer).

These two informations are sufficient to actually compute the real name of the interval. For instance, a major third interval is represented by an instance of Interval whose *type* is 3 (for 'third'), and whose *semiTones* is 5. This is represented by the printing method of class `Interval`, that prints an interval according to the human (mysterious) terminology, that allows perfect *fifth*s ou *fourth*s, but major and minor *thirds* .

4.1. Computing interval extremities

In order to compute the note forming a given interval with a given note, we will follow the human algorithm which says that computing an interval consists in the following steps : (we will take the example of computing the diminished fifth of C*b*) :

1. getting to the natural note. In our example, C*b* yields C.
2. enumerating as many steps as the interval says. Here, a diminished fifth is a fifth, so we will enumerate five notes, starting from C : C, D, E, F, G. We get a G.
3. Adding one or two # or *b* to the resulting note (here G) to yield the right number of half-tones. In our example, we want a diminished fifth, which is 6 half-tones. From C*b* to G there are 8 half tones, so we send the message *flat flat* to the result, eventually getting G*bb*.

Here is the corresponding method, which computes the diminished fifth of a note. It is defined in the root class of notes (`Note`).

```
!Note methodsFor: 'intervals'!
diminishedFifth
    ^Interval diminishedFifth topIfBottomIs: self
```

The main method is `topIfBottomIs:`, which is defined in class `Interval` as follows :

```
!Interval methodsFor: 'computing'!
topIfBottomIs: aNote
    "yields the note making the interval self with aNote"
    ^aNote alterate: (aNote nthFollowing: type - 1) toReach: semiTones
```

This method of class `Interval` uses two methods defined in class `Note` : `nthFollowing:` and `alterate:toReach:`. Method `nthFollowing:` simply yields the nth following note, in the natural ordering :

```
!Note methodsFor: 'intervals'!
nthFollowing: i
    | result |
    result := self natural.
```

```
    i timesRepeat: [result := result following].
    ^result
```

Now the main method is actually the method `alterate:toReach:`, which takes two arguments : a naturalNote *n*, and a number of semiTones *s*. The method sends the right number of sharp or flat messages to the natural note to reach an interval with *s* semiTones.
It is important here to note that this method may be sent to any type of note. The action to perform depends on the type of the note so we actually define 5 such methods.
The first one deals with natural notes. The computation is based on the difference between semiToneCounts of its extremities. Depending on this difference, the messages sharp and flat are sent to the note passed in parameter.

```
!NaturalNote methodsFor: 'intervals'!
alterate: note toReach: s
    | delta |
    delta := (self semiTonesWithNaturalNote: note) - s.
    delta = 0 ifTrue: [^note].
    delta = 1 ifTrue: [^note flat].
    delta = -1 ifTrue: [^note sharp].
    delta = 2 ifTrue: [^note flat flat].
    delta = -2 ifTrue: [^note sharp sharp].
    ^self error: 'illegal interval'
```

The method `semiTonesWithNaturalNote:` is defined simply as a difference of semiToneCounts mod 12 :

```
!NaturalNote methodsFor: 'intervals'!
semiTonesWithNaturalNote: aNote
        ^aNote semiToneCount - semiToneCount \\ 12
```

Now what happens to non natural notes ? The answer is simple. For `SharpNotes` for instance, the computation consists in delegating the result to the corresponding natural note, and then sending a sharp message to the result, as follows :

```
!SharpNote methodsFor: 'intervals'!
alterate: note toReach: s
    ^(natural alterate: note toReach: s) sharp
```

Similarily, the same mechanism holds for Flat, DoubleFlat and DoubleSharp notes.

The dual problem, i.e. finding the "bottom" of an interval, given its top, is now easily defined as follows, by using the "reverse" of an interval :

```
!Interval methodsFor: 'computing'!
bottomIfTopIs: aNote
    "yields the note from which aNote yields interval self"
    ^self reverse topIfBottomIs: aNote!
```

4.2. Computations on intervals

The reverse of an interval is trivially defined by computing the complement to 9 for type, and to 12 for semiTones :

```
!Interval methodsFor: 'reverse'!
reverse
    ^self class type: (9 - type) semiTones: (12 - semiTones)
```

Here is a micro-session that exemplifies interval computations :

```
Note C flatFifth                          ->    Gb
Note C augmentedFourth                    ->    F#
Note C majorThird majorThird              ->    G#
Note C flat minorSeventh                  ->    Bbb
Note C flat diminishedSeventh             ->    error: illegal interval

Interval diminishedFifth bottomIfTopIs: (Note F sharp)    -> C
Interval diminishedFifth bottomIfTopIs: (Note G flat)     -> Dbb

Interval majorThird reverse                               -> minor sixth
Interval perfectFifth + Interval majorSecond              -> majorSixth

(Note C diminishedFifth) pitchEquals: (Note F minorSecond)  -> true
```

4.3. Computing intervals from its extremities

Finally, computing an interval from two notes is simple, and implemented by only one method in class `Note` :

```
!Note methodsFor: 'intervals'!
intervalWith: aNote
    | b b2 type |
    type := 1.
    b := self natural.
    b2 := aNote natural.
    [b2 = b] whileFalse:  [b := b following. type := type + 1].
    ^Interval type: type semiTones: (self numberOfSemiTonesWith: aNote)
```

The method `numberOfSemiTonesWith:` is implemented as follows in class `Note`, by cutting the job in three pieces :

```
!Note methodsFor: 'intervals'!
numberOfSemiTonesWith: aNote
        ^self semiTonesWithNatural +
        (self natural semiTonesWithNaturalNote: aNote natural) -
        aNote semiTonesWithNatural
```

The methods `semiTonesWithNatural` and `semiTonesWithNatural:` are implemented respectively in each subclass to yield the correct result, once again using polymorphism. This method may be used as follows :

```
Note C intervalWith: Note F sharp                 ->    augmented fourth
Note C sharp intervalWith: Note G                 ->    diminished fifth

(Note C intervalWith: Note G) =
(Note D sharp intervalWith: Note A sharp)         ->    true
```

5. Scales

Let us now proceed with much more exciting matter : scales and chords. Strangely, these are extremely simple to represent, once the foundation is set (and solid!). Here are some of the things we want to do with scales, in the context of harmonic amalysis :

- Find all the scales that contains n given notes,
- Find the signatures of scales (number of sharps and flats),

- Find the notes a of scale,
- Know that certain scales are forbidden,
- Generate scale-tone chords from a scale.

5.1. Definition and creation of scales

We actually have all we need to represent scales : a scale is an ordered list of intervals, starting on a given root note. The class `Scale` is defined with the following instance variables :

a *root* that points to the root note,
a list of *notes* of the scale[2]. This list of notes may be deduced from the root and type as we will see.

Now there are different types of scale : major scales, harmonic minor scales, melodic minor scales[3]. The type of the scale could be represented by yet an other instance variable. But there is a better solution that allows us to benefit, once more, from the advantages of polymorphism. This solution consists in creating subclasses of `Scale` to represent the various possible types. In this scheme, the class `Scale` is an abstract class, i.e. does not have any instance, but serves as a root for subclasses, which will implement the actual definitions (here, the series of intervals) of particular types of scales.

Here is how it works. The main creation method is defined as follows, with one argument : the root of the scale. This creation method is also in charge of computing the list of notes and testing the validity of the scale.

```
!Scale class methodsFor: 'creation'!
root: aNote
    |s|
    s := self new root: aNote; computeNotes.
    ^s isValid ifTrue: [s] ifFalse: [self error: 'invalid scale']
```

Now the 2 important methods are `computeNotes` and `isValid`, and are defined as follows :

```
!Scale methodsFor: 'computing notes'!
computeNotes
"intervalList depends on the type of the scale. It is defined in each subclass of Scale"
    notes := self intervalList collect: [:s | root perform: s]!
```[4]

The actual interval list is defined in each particular subclass of `Scale`. This is the only method needed to define a subclass of Scale. For instance, here are the definition of `Major`, `HarmonicMinor` and `MelodicMinor` scales by their intervalList definition :

```
!MajorScale methodsFor: 'interval list'!
intervalList
    ^#(yourself second majorThird fourth fifth majorSixth majorSeventh)

!HarmonicMinorScale methodsFor: 'interval list'!
intervalList
    ^#(yourself second minorThird fourth fifth minorSixth majorSeventh)
```

---

2 At this point, we do not consider the problem of finding the scale corresponding to a set of notes, as this is handled by successive layers of the system.
3 These 3 types of scales are sufficient to describe most of standard be-bop Jazz music, but new ones could be added to cope with exotic tonalities such as the ones founded in some Jazz-Rock tunes (Cf. § on genericity)). These scales are also called *synthetic modes* in some literature.
4 Note the "smart" use of `perform:` to compute the intervals using the interval computation methods.

```
!MelodicMinorScale methodsFor: 'interval list'!
intervalList
    ^#(yourself second minorThird fourth fifth majorSixth majorSeventh)
```

The validation test consists in checking the absence of any double altered note. The creation of scales is defined as follows in class `Note` by sending the a creation message to the corresponding Scale class with self as the root parameter :

```
!Note methodsFor: 'scales'!
majorScale
    ^MajorScale root: self
```

Here is a micro-session for scales :

```
Note A flat majorScale                -> Ab major
Note A flat majorScale notes          -> (Ab Bb C Db Eb F G)
Note C harmonicMinorScale notes       -> (C D Eb F G Ab B)
Note G sharp majorScale               -> error: 'invalid scale'
```

6. Chords

6.1. Definition and creation of chords

Let us proceed with the core of harmonic analysis : chords. We propose here a representation of chords that is based on the representation of notes, intervals and scales defined above, which allows to make various computations such as :

- finding the name of a chord given a set of notes,
- finding the set of notes given a chord name,
- finding all the possible harmonic analysis of a chord, in various scales.

Chords are represented by a class with two main instance variables : a `root`, which is a note, and a `structure`, which is a list of symbols. Chords may be created by sending a message to the root, with the structure as argument, or by sending a message to class `Chord` with the complete string as argument, such as :

```
Note C sharp chordFromString: 'min'     -> C#  min
Note D chordFromString: ''              -> D

Chord newFromString: 'A min 7 9'        -> A min 7 9
```

6.2. Creating the structure from the list of notes

One of the problems with chords is to find a stable and consensual terminology. Here, this problem is trivially represented by a method that takes successively each note, compares it to the root, and deduce the corresponding piece of structure. We do not include this method for reason of space.

6.3. Creating the list of notes from the structure

The reverse problem consists in finding the list of notes given a particular structure. To garanty a unique and non ambiguous terminology we systematically format chords using the

two preceding methods. Since the problem of chord identification would necessitate a whole report, so the corresponding methods are not described here (teh interested reader may obtain the full report with all the details on this aspect).

```
(Chord newFromString: 'C min dim5 aug9') notes          ->          (C Eb Gb Bb D#)
```

6.4. Computing chords from scales

An extremely important and interesting feature of scales is their ability to generate the so-called *scale-tone chords*. In a way, the whole mechanic of harmonic analysis is based on this principle (in the other way round, Cf. below).
Generating chords from a scale is an operation that takes two arguments : a number of polyphony *p*, and an interval *i*. The generation of chords consists simply in building (7) sets of notes. Each set of notes (a chord) is built by taking successively each note of the scale, and iteratively (p times) getting its *i* th following note. The classical case is when i = 3, and the chords are built by successive thirds. The method that implements this latter case is `generateChordsPoly:`, which only needs the polyphony parameter :

Here is a micro-session that generates chords :

```
Note C majorScale generateChordsPoly: 7    ->

OrderedCollection (C maj7 9 11 13, D min 7 9 11 13, E min 7 dim9 11 13, F maj7 9
aug11 13, G 7 9 11 13, A min 7 9 11 dim13, B min dim5 11 dim13)

Note D harmonicMinorScale generateChordPoly: 3  ->
OrderedCollection (D min,  E min flat5,  F aug5,  G min, A,  Bb,  C# min flat5)
```

6.5. Computing possible analysis

Now that we know how to generate scale-tone chords from a given scale, we are, of course, also interested in the reverse operation, which is the at the heart of harmonic analysis : knowing, for a given chord, what analysis it can "support", i.e. what are the scales from which is may be generated, and, for each of these possible scale, what is the *degree* of the chord.

Let us first represent explicitly the notion of `HarmonicAnalysis`, with a trivial representation by two instance variables :

```
Object subclass: #HarmonicAnalysis
    instanceVariableNames: 'scale degree'
```

`HarmonicAnalysis` defines a printing method to print itself between brackets {}, and with roman literals. Now the method that computes all possible analysis for a given chord is naturally defined in class `Chord` by adding all the possible analysis in a given scale (i.e. a subclass of Scale), for all possible scales :

```
!Chord methodsFor: 'computing tonalities'!
possibleTonalites
    | result |
    result := OrderedCollection new.
    Scale allSubclasses do:
      [:aScaleClass | result addAll: self possibleTonalitiesInScaleClass: aScaleClass].
    ^result

possibleTonalitesInScaleClass: aScaleClass
```

```
| ana scale chords  possibleTonalities|
self format.
possibleTonalities := OrderedCollection new.
scale := aScaleClass root: Note C.
chords := scale generateChordsPoly: notes size.
chords do: [:c | (c matchWith: self) ifTrue:
   [ana := Analysis new degree: (scale degreeOfChord: c).
   ana scale: (aScaleClass root:
                    (self root transposeOf: (aScaleClass root intervalWith: scale root))).
   possibleTonalities add: ana]].
^possibleTonalities
```

Here is the corresponding micro-session :

```
(Chord new fromString: 'C maj') possibleTonalities          ->
 OrderedCollection (
  {IV of G MelodicMinor}     {V of F MelodicMinor}  {I of C Major}
  {IV of G Major}            {V of F Major}         {V of F HarmonicMinor}
  {VI of E HarmonicMinor} )

(Chord new fromString: 'D min 7 dim5') possibleTonalities  ->
OrderedCollection (
  {IV of F MelodicMinor}     {VII of Eb MelodicMinor}
  {VII of Eb Major}          {II of C HarmonicMinor} )
```

6.6. Genericity and Reusability

One of the main advantages of our approach, besides the clarification it brings to the overall algebra of alterations, intervals and scales, is the fact that all the mechanisms may be extended very easily, mainly by subclassing. For instance, our representation of scales makes it straightforward to add new types of scales, using inheritance. Introducing a new type of scale consists simply in creating a new subclass of `Scale`, and defining its interval list. The new class is then ready to use.

For instance, we can define the `HungarianMinor` scale as a subclass of class Scale and the following method :

```
!HungarianMinor methodsFor: 'interval list'!
intervalList
    "example : (C D Eb F# G Ab B)"
    ^#(yourself second minorThird augmentedFourth fifth minorSixth majorSeventh)
```

We can then right away use all the preceding methods without any modification. For instance, we can compute the new (exotic) set of possible chords generated by this scale as :

```
(HungarianMinor root: Note C) generateChordPoly: 4          ->
OrderedCollection (C min maj7,  D dim5 7,  Eb aug5 maj7,  F# dim5 dim7,  G maj7,
Abmaj7, B min dim7)
```

Of course, we will be also able to use this scale for performing exotic analysis, in the successive layers, at a minimal cost ! Here are for example, the possible analysis of a chord, in this new tonality :

```
(Chord new fromString: 'C maj') possibleTonalities  ->
 OrderedCollection (
{V of F HungarianMinor}     {VI of E HungarianMinor}
{IV of G MelodicMinor}      {V of F MelodicMinor}
```

```
{I of C Major}                  {IV of G Major}
{V of F Major}                  {V of F HarmonicMinor}
{VI of E HarmonicMinor} )

(Chord new fromString: 'D min') possibleTonalitiesIn: HungarianMinor ->
OrderedCollection ( {I of D HungarianMinor }  {VII of Eb HungarianMinor })
```

As John McLaughlin (one of the inventor of Jazz-rock, who, among other things, introduced sophisticated and hard-to-analyse harmonic progressions in Jazz) writes in the foreword of [Mahavishnu 76] : *"... one can find so much hidden within [synthetic modes], particularily in the extraction of their scale-tone chords"*. Well, the extraction and study of these exotic scale-tone chords and their interactions is now a child's play :

```
!NeapolitanMinor methodsFor: 'interval list'!
intervalList
    "example : (C Db Eb F G Ab B)"
    ^#(yourself minorSecond minorThird perfectFourth fifth minorSixth majorSeventh)

!DoubleHarmonic methodsFor: 'interval list'!
intervalList
    "example : (C Db E F G Ab B)"
    ^#(yourself minorSecond majorThird fourth fifth minorSixth majorSeventh)
```

... and so on : McLaughlin gives 16 synthetic modes, which can be all represented similarily. We can now have the full possible analysis for any chord in any scale, and study them by appropriate queries to MusES.

7. Extending the system

7.1. Representing actual octave-dependent notes

As we said in the beginning, our theory only takes pitch-classes into account, and does not differentiate several notes belonging to the same pitch class (*octave-dependent notes*). The first idea that comes to mind to include these actual octave-dependent notes in our system is to have our present notes (instances of the various subclasses of `Note`) *become classes*, in the sense of OOP, so that one can make instances out of them ! For instance, we would like to say that note *C3* is an instance of pitch-class C. And of course pitch-class C would still be an instance of class `NaturalNote` !
This procedure, which consists in raising all the classes and instances one step higher in the instanciation tree is technically possible[5], but raises an ontological problem : What do we want to consider global vs volatile ?
Intuitively, we would like to say that pitch-classes are global objects, but that octave-dependent notes are not. There are two arguments to support this claim : (1) Pitch classes are not too many (35), compared to actual octave-dependent notes (35 * say, 8 octaves = 280 notes !), and (2) there is no reason to decide a priori what are the limits in the octave multiplication : 8 seems a good approximation, but then we will have the problem of deciding what happens to the upper or lower bounds (would we authorize interval computations on these bounds for instance ?). This lead us to consider a representation for octave-dependent notes as instances, and pitch-classes as classes. Because of space limitation, we will not discuss these technical details here.

---

[5] But it is not trivial, since metaclasses are not really first-class objects in Smalltalk. However, small extensions to Smalltalk allow the user to have complete control on metaclasses (Cf. the ClassTalk system by [Cointe&Briot 89]).

7.2. Problems not solved

There are a couple of classical problems involving pitch class computation we did not deal with yet, such as : computing the scale from a list of notes, or : given an incomplete list of notes (of length < 7), compute the list of plausible scales. We hope that our presentation convinced the reader that these extension are trivial to add to the existing system.

7.3. Representing non trivial reasoning

The system presented here achieves is goal, which is to represent the basic harmonic entities necessary to perform sophisticated reasoning. The representation of this reasoning is the main goal of the higher levels of the MusES system, and is described in subsequent documents. The central idea of these extensions is to use a specialized forward-chaining, first-order inference mechanism (NéOpus) with which all the *reasonings* involving the objects defined here are represented. More on this can be found in [Pachet 91], the expertise is described in [Pachet 87] but represented awkwardly, and a forthcoming report will present a version of the system as an extension to the present architecture.

8. Conclusion

The first layer of the MusES system sets the foundations for the study of various harmonic analysic mechanisms. The basic entites of harmony notes, intervals, scales and chords are defined as by set of classes, having a structure and a behavior. Our approach is validated by the "friendly" feel of the overall system and the almost physical presence of the musical entities, that allow the user to think more naturally, and by the reusability of these entities, and their capacity to support extensions.

9. References

[Cointe&Briot 89] Cointe P., Briot J.-P. Programming with ObjVlisp metaclasses in Smalltalk-80, OOPSLA '89, New Orleans, USA.

[Ebcioglu 92] Ebcioglu K. An expert system for harmonizing chorales in the style of Bach. In Understanding Music with A.I. AAAI Press/ MIT Press, 1992. Ed. by Balaban M., Ebcioglu K., Laske O.

[Goldberg&Robson 89] Goldberg A., Robson D. Smalltalk-80 : the language and its implementation. Addison-Wesley 1989 (revised edition).

[MacLaughlin 76] J. McLaughlin. John McLaughlin and the Mahavishnu Orchestra. Warner-Tamerlane publishing, Warner Bros. Publishing, New York,1976.

[Mahr 80] Mahr Ehrig. Fundamentals of Algebraic Specifications. Vol 1.

[Pachet 87] F. Pachet. Vers un système expert de suivi d'improvisation. Rapport de DEA IARFA, IRCAM/Paris 6, September 1987.

[Pachet 91] F. Pachet. A meta-level architecture for analysing jazz chord sequences. Proceedings of ICMC 91, Montréal.

[Perrot 92] J.-F. Perrot. Langages à objets, Programmation par Objets. Rapport LAFORIA n° 92/34. Novembre 1992.

[Pope 9] Pope Steven. The Well-Tempered Object. MIT Press, 1991.

[Smaill&Wiggins 90] Smaill Alan, Wiggins Geraint. Hierarchical music representation for composition and analysis. In Colloque International "Musique et assistance informatique", pp. 261-279, Marseille, 3-6 oct. 1990.

[Steedman 84] Steedman M.J. A Generative Grammar for Jazz Chord Sequences. Music Perception, Fall 1984, Vol. n° 2, N° 1, pp. 52-77.

[Winograd 93] T. Winograd. Linguistics and the Computer Analysis of Tonal Harmony. In Machines Models of Music, Edited by S. M. Schwanauer and D.A. Levitt, MIT Press, 1993.